

Model-Driven Software Modernization

Concept-Based Engineering of Situation-Specific Methods



PADERBORN UNIVERSITY
The University for the Information Society

Marvin Grieger

Faculty of Computer Science, Electrical Engineering and Mathematics

Paderborn University

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

May 2016

Abstract

During a software modernization project, a legacy system is transferred into and adapted to a new environment. A transformation method guides this endeavor by prescribing activities to perform or artifacts to generate in order to realize the transition. Thereby, the method used needs to fit to the situation at hand by considering conceptual differences between source and target environment and by automating parts of the transformation whenever suitable. Otherwise, a decreased software quality or increased effort to perform the transformation may result. Although various method engineering approaches have been proposed to support the development of transformation methods, most of them do not provide sufficient flexibility in the development or fall short in guiding the endeavor. In this thesis, we address this problem by introducing a situational method engineering framework to guide the development of situation-specific transformation methods. The framework uses a method base that contains reusable building blocks of transformation methods, based on principles from the domain of model-driven engineering. The development of a method is centered around the identification of concepts within a legacy system that represent its functionality by abstracting from the technology-specific realization. Selecting predefined, model-driven transformation strategies for each concept enables assembling transformation methods that are well-suited for a software modernization scenario.

Zusammenfassung

In einem Softwaremodernisierungsprojekt wird ein Altsystem in eine neue Umgebung überführt und an diese angepasst. Eine Transformationsmethode leitet dieses Unterfangen an, indem sie auszuführende Aktivitäten oder zu generierende Artefakte beschreibt. Dabei muss die Methode an die Projektsituation angepasst sein, indem sie konzeptionelle Unterschiede zwischen der Quell- und Zielumgebung berücksichtigt und eine Automatisierung ermöglicht. Eine nicht angepasste Methode kann eine verringerte Qualität des resultierenden Systems oder einen erhöhten Aufwand für die Transformation zur Folge haben. Bestehende Ansätze zur Erstellung von Transformationsmethoden stellen entweder eine zu geringe Flexibilität in der Entwicklung bereit oder leiten diese nicht ausreichend an. In dieser Arbeit adressieren wir dieses Problem durch die Definition eines Frameworks zur Erstellung situationsspezifischer Transformationsmethoden. Dazu nutzt das Framework eine Methodenbasis, welche wiederverwendbare Bausteine von Transformationsmethoden basierend auf Prinzipien der modellgetriebenen Entwicklung beinhaltet. Im Mittelpunkt der Methodenentwicklung steht die Identifikation von Konzepten innerhalb eines Altsystems, welche dessen Funktionalität repräsentieren und von der technologiespezifischen Realisierung abstrahieren. Die Auswahl von vordefinierten, modellgetriebenen Transformationsstrategien für jedes Konzept ermöglicht den systematischen Aufbau einer Methode für die Modernisierung eines Systems.

Danksagung

Ich möchte mich zu allererst bei den vielen Personen bedanken, die direkten oder indirekten Einfluss auf die Entstehung dieser Arbeit hatten. Ohne sie wäre diese Arbeit in dieser Form nicht möglich gewesen.

Allen voran möchte ich meinem Doktorvater Prof. Dr. Gregor Engels danken. Durch seine intensive wissenschaftliche Betreuung über die Jahre, die vielen Ratschläge, Diskussionen und Denkanstöße hat er die Arbeit maßgeblich geprägt. Gleichzeitig hat sein Auge für das Detail und die Fähigkeit, komplexe Dinge auf das Wesentliche reduzieren zu können, meine Denkweise nachhaltig geprägt. Danken möchte ich auch Dr. Stefan Sauer und Prof. Dr. Jürgen Ebert für die Diskussionen über die Arbeit und für ihre Funktionen in der Promotionskommission.

Die Ausarbeitung einer solchen Arbeit ist ein mehrjähriges, anstrengendes Unterfangen. Ich hatte das Glück, während dieser Zeit fortwährende Unterstützung durch meine Familie zu erfahren. Vor allem durch meine kleine Familie, meiner Frau Stefanie und unserer Tochter Marie. Steffi hat die ganze Zeit über zu mir gehalten, mich motiviert und mir Rückhalt gegeben, dafür möchte ich ihr von ganzem Herzen danken.

Ein genauso großer Dank geht auch an meine Eltern, Sabine und Ralf. Sie haben mich zu dem gemacht, der ich heute bin und stehen mir immer mit Rat und Tat zur Seite. Dafür danke ich ihnen zutiefst. Ebenso möchte ich meiner Lieblingsschwester Danika und ihrem Mann Jan für die immerwährende Unterstützung danken. Ohne meinen Onkel, Klaus Sonnemann, wäre diese Arbeit vielleicht niemals zu Stande gekommen. Er hat damals einen der Grundsteine für mein Studium gelegt, wofür ich ihm dankbar bin.

Diese Arbeit wurde in vielen Diskussionen geschärft, wobei einige Personen besonders hervorzuheben sind. Mit Markus Klenke konnte ich einige Jahre in einem industriellen Kontext eng zusammenarbeiten. Er hat wesentlich dazu beigetragen, die Kernidee dieser Arbeit auszuarbeiten und umzusetzen, wofür ich dankbar bin. Genauso möchte ich Svetlana Arifulina und Masud Fazal-Baqaie danken, für die vielen Diskussionen, die Unterstützung und gegenseitige Motivation. Danken möchte ich auch meinem langjährigen Bürokollegen Barış Güldali, insbesondere für die Einführung in den wissenschaftlichen Kontext.

Ich habe das Arbeitsklima in der AG Engels immer als sehr gut empfunden, weshalb ich nicht zuletzt allen meinen Kollegen danken möchte.

Table of Contents

List of Figures	xiii
------------------------	-------------

List of Tables	xvii
-----------------------	-------------

I Foundations and Related Work	1
---------------------------------------	----------

1 Introduction	3
-----------------------	----------

1.1 Software Transformation Methods	6
1.2 Problem Statement	9
1.3 Requirements	11
1.4 Solution Concept	12
1.5 Overview of Publications	13
1.6 Structure of this Thesis	15

2 Foundations	17
----------------------	-----------

2.1 Model-Driven Engineering	17
2.1.1 Meta-Object Facility	18
2.1.2 Model-Driven Architecture	19
2.1.3 Architecture-Driven Modernization	20
2.2 Method Engineering	22
2.2.1 Situational Method Engineering	23
2.2.2 Software and Systems Process Engineering Metamodel	24
2.2.3 Metamodeling Layers	25
2.3 Software Reengineering	26
2.3.1 Software Migration, Transformation & Modernization	27
2.3.2 Compiler	29
2.3.3 Programming Paradigms	31
2.3.4 Concept Modeling	32

3	Scenario and Related Work	35
3.1	Modernization Scenario	35
3.1.1	Oracle Forms	36
3.1.2	Oracle ADF	38
3.2	Requirements	39
3.3	Related Work	41
3.3.1	Situational Method Engineering Approaches	41
3.3.2	Reengineering Frameworks	44
3.3.3	Evaluation	45
3.4	Summary	48
II	Solution Concept	49
4	Overview	51
4.1	The MEFiSTo Framework	51
4.1.1	Method Base	52
4.1.2	Method Engineering Process	54
4.1.3	Phases & Roles	55
4.2	Evaluation Criteria	57
4.3	Running Example	58
4.3.1	Summit	59
4.4	Summary	63
5	MEFiSTo Method Base	65
5.1	Requirements	65
5.2	Overview of the Structure	68
5.2.1	Method Fragments	68
5.2.2	Method Patterns	70
5.3	Transformation Phase Fragments	73
5.4	Tool Implementation Phase Fragments	77
5.5	Basic Transformation Patterns	79
5.5.1	Language Transformation	80
5.5.2	Conceptual Transformation	92
5.5.3	Reimplementation	101
5.5.4	Code Removal	107
5.5.5	Platform-Dependent Architecture Restructuring	112

5.6	Composed Transformation Patterns	119
5.6.1	Language Transformation-Based Reimplementation	121
5.6.2	Concept Recognition-Based Language Transformation	128
5.7	Formalization	133
5.7.1	MEFiSTo Intermediate Modeling Language (MIML)	135
5.7.2	SPEM	136
5.7.3	Transforming MIML to SPEM	139
5.8	Summary	140
6	MEFiSTo Method Engineering Process	141
6.1	Requirements	141
6.2	Overview of the Process	143
6.3	Situational Context Identification	146
6.3.1	Concept Identification	146
6.3.2	Influence Factor Identification	154
6.4	Transformation Method Construction	160
6.4.1	Method Pattern Selection & Configuration	161
6.4.2	Method Pattern Integration	173
6.4.3	Instantiation of Tool Implementation Phase Fragments	179
6.4.4	MIML To SPEM Transformation	181
6.4.5	Method Completion	181
6.5	Tool Implementation	183
6.6	Transformation	186
6.7	Summary	187
III	Evaluation and Conclusion	189
7	Feasibility Studies	191
7.1	Evaluation Criteria Revisited	191
7.2	Feasibility Study 1: Oracle Forms to Oracle ADF	192
7.2.1	Situational Context Identification	193
7.2.2	Transformation Method Construction	196
7.2.3	Tool Implementation	198
7.2.4	Transformation	201
7.3	Feasibility Study 2: Oracle Reports to Jasper Reports	205
7.3.1	Situational Context Identification	207

7.3.2	Transformation Method Construction	210
7.3.3	Tool Implementation	212
7.3.4	Transformation	212
7.4	Discussion	215
7.5	Summary	218
8	Conclusion and Future Work	219
8.1	Contributions	219
8.2	Requirements Revisited	221
8.3	Future Work	223
8.3.1	Enhancing the MEFiSTo framework	224
8.3.2	Example-Based Method Learning	226
	References	227
	Appendix A Characterization of Method Patterns	239
	Appendix B Overview of the Method Engineering Process	245
	Glossary	247
	Acronyms	257

List of Figures

1.1	Staged software lifecycle	3
1.2	Summit application in Oracle Forms and Oracle ADF	5
1.3	Core disciplines of a software modernization method and their relation to the contained transformation method	6
1.4	Conceptual representation of the horseshoe model, consisting of the consecutive phases reverse engineering, restructuring and forward engineering	7
1.5	Imperative realization of a dialog flow concept in the source environment and a declarative realization in the target environment	8
1.6	Categorization of transformation methods based on their adaptability	10
1.7	Solution concept overview	12
1.8	Overview of the publications related to this thesis	14
1.9	Thesis structure	15
2.1	Solution concept overview	17
2.2	Exemplary Meta-Object Facility-based architecture	18
2.3	Artifacts and abstraction levels of the MDA	20
2.4	Modeling languages and abstraction levels of the ADM	20
2.5	Packages and layers of the Abstract Syntax Tree Metamodel	21
2.6	Packages and layers of the Knowledge Discovery Metamodel	21
2.7	Packages and metamodel classes of SPEM	24
2.8	Metamodeling layers involved when developing and enacting a method	26
2.9	Structure of a compiler	29
2.10	Source code and its abstract syntax graph	29
2.11	Imperative realization of attribute validation rules in Oracle Forms	32
2.12	Declarative realization of attribute validation rules in Oracle ADF	32
2.13	Representing a software system as a set of concepts	33
3.1	Architecture of Oracle Forms-based systems, interpreted as MVC architecture	37
3.2	Model-View-Controller (MVC) architecture of Oracle ADF-based systems	38

3.3	Categorization of situational method engineering approaches to develop transformation methods according to their degree of controlled flexibility	42
4.1	Pattern-based development of transformation methods	53
4.2	Overview of the MEFiSTo framework	54
4.3	Core activities, associated roles and phases of MEFiSTo	56
4.4	User interface of the Summit application to view and edit order information .	59
4.5	Form Modules of the Summit application, their contained dialogs and the navigation flows between them	62
5.1	Use of a technology-independent method base in MEFiSTo, reengineering of developed methods to derive a technology-specific method base	66
5.2	Overview of the structure of the method base in MEFiSTo	69
5.3	Simplified conceptual model of the main concepts that are involved when assessing the suitability of a method pattern	72
5.4	Method fragments of the transformation phase, visualized as an integrated horseshoe model	74
5.5	Method fragments of the tool implementation phase	78
5.6	Basic transformation method patterns	80
5.7	Enacting a transformation method to transform the internal representation of database tables of the running example by using the language transformation pattern	83
5.8	Model transformation rules to transform the L-PSM into the T-PSM	86
5.9	Developing a transformation method to transform the internal representation of database tables by using the language transformation pattern	87
5.10	Enacting a transformation method to transform the attribute validation rules of the running example by using the conceptual transformation pattern	95
5.11	Enacting a transformation method to transform the view-based data access of the running example by using the reimplementation pattern	103
5.12	Potential differences between a legacy and a target system	106
5.13	Enacting a transformation method to transform the tree-based data selection of the running example by using the code removal pattern	110
5.14	Enacting a transformation method to transform the architecture of the running example by using the platform-dependent architecture restructuring pattern .	115
5.15	Composed transformation method patterns	120

5.16	Enacting a transformation method to transform the internal representation of database tables and associated attribute validation rules of the running example by using the language transformation-based reimplementation pattern	124
5.17	Resulting models when enacting a transformation method to transform the internal representation of database views of the running example by using the concept recognition-based language transformation pattern	130
5.18	Overview of packages when formalizing a method in MEFiSTo	134
5.19	Model of a transformation method in MIML	135
5.20	Model atransformation method in SPEM	137
5.21	Transforming a MIML model to SPEM using model transformations	139
6.1	Core activities of the method engineering process of MEFiSTo	143
6.2	Exemplification of the relationships between the different phases of the method engineering process of MEFiSTo	145
6.3	Situational context identification process	146
6.4	Classes of concepts involved in a software modernization scenario	147
6.5	Concept model of the running example, specified in MIML	149
6.6	Using concepts to abstract from technology-specific realizations	151
6.7	Target-driven concept identification process	153
6.8	Excerpt of the MIML metamodel to formalize concept models	154
6.9	Classification of influence factors	157
6.10	Influence factors related to the attribute validation concept of the running example, specified in MIML	157
6.11	Influence factor identification process	158
6.12	Excerpt of the MIML metamodel to formalize influence factor models	160
6.13	Transformation method construction process	161
6.14	Instantiating a fragmented transformation method specification from a situational context model	162
6.15	Concept model of the running example with selected and configured method patterns, specified in MIML	164
6.16	Horseshoe model showing customized method fragments to transform the table-based view concept of the running example, specified in MIML	165
6.17	Example for an interrupted transformation method specification, graduated enactment of an integrated transformation method	167
6.18	Excerpt of patterns and anti-patterns within a concept model	168
6.19	Removal of an anticipation anti-pattern instance within a concept model	169
6.20	Method pattern selection and configuration process	173

6.21	Integrating a fragmented transformation method specification	174
6.22	Integrated horseshoe model showing method fragments for the table-based view, attribute calculation and view relation concept, specified in MIML . . .	176
6.23	Transformation process	177
6.24	Instantiation of tool implementation phase fragments	179
6.25	Tool implementation process and model transformation rules definition process	180
6.26	Excerpt of SPEM-based transformation method specification that defines a method to transform the running example	183
7.1	Real estate management legacy system in the source environment, the trans- formed system in the target environment	192
7.2	Configured concept model for the real estate management legacy system . . .	194
7.3	Legacy and target realization of the dialog flow concept	195
7.4	Integrated horseshoe model of the real estate management system for the dialog, dialog flow and business module concepts	198
7.5	Architecture of the generic tool infrastructure	199
7.6	Henshin model transformation rule to extract flows between dialogs	200
7.7	Enacting a transformation method to transform the dialog, dialog flow and business module concepts of the real estate management legacy system	202
7.8	Result of a semi-automatic architectural restructuring of the real estate man- agement legacy system by clustering related business modules	203
7.9	Effort for transforming the real estate management legacy system, amount of feedback exchanged between involved roles	204
7.10	Legacy report in the development environment of the source environment, transformed report in the development environment of the target environment	206
7.11	Configured concept model for the reports legacy system	208
7.12	Legacy and target realization of the report layout and element concepts	209
7.13	Integrated horseshoe model for the report layout and report element concepts	211
7.14	Enacting a transformation method to transform the report layout and element concepts of the legacy report	213
B.1	Overview of the method enactment process of MEFiSTo	245
B.2	Overview of the method development process of MEFiSTo	246

List of Tables

3.1	Evaluation of selected method engineering approaches against requirements	46
5.1	Schema to characterize method patterns	71
5.2	Language transformation pattern characteristics	81
5.3	Conceptual transformation pattern characteristics	93
5.4	Reimplementation pattern characteristics	102
5.5	Code removal pattern characteristics	108
5.6	Platform-dependent architecture restructuring pattern characteristics	113
5.7	Language transformation-based reimplementation pattern characteristics	122
5.8	Concept recognition-based language transformation pattern characteristics	129
6.1	Transferring principles from the domain of architectural design decisions to the domain of MEFiSTo	155
6.2	Operations for the integration of method fragments that originate from different method pattern applications	175
A.1	Concept recognition-based reimplementation pattern characteristics	240
A.2	Concept recognition and language transformation-based reimplementation pattern characteristics	242
A.3	Platform-dependent architecture restructuring pattern characteristics	243
A.4	Architecture restructuring pattern characteristics	244

PART I

FOUNDATIONS AND RELATED WORK

“The beginning is the most important part of the work.”

– PLATO

CHAPTER 1

Introduction

Information systems have become a critical asset for most companies as daily business often depends on them. Each system has a lifecycle consisting of multiple stages, as illustrated in Figure 1.1. After a system has been developed initially, it automatically enters the *Evolution* stage [MD08, pp.1-3]. In this stage, it is feasible to modify the system in order to remove bugs, improve non-functional characteristics or change its functionality due to changed requirements.

Just as anything else, information systems are aging over time. This aging process can have various effects on the system like its architectural integrity becoming violated, its documentation becoming outdated or its technological environment becoming obsolete. In each case, the aging process ultimately results in the *loss of evolvability*, preventing the modification of the information system to meet new or changed requirements. If the system is still valuable for ongoing business, it has become a legacy system.

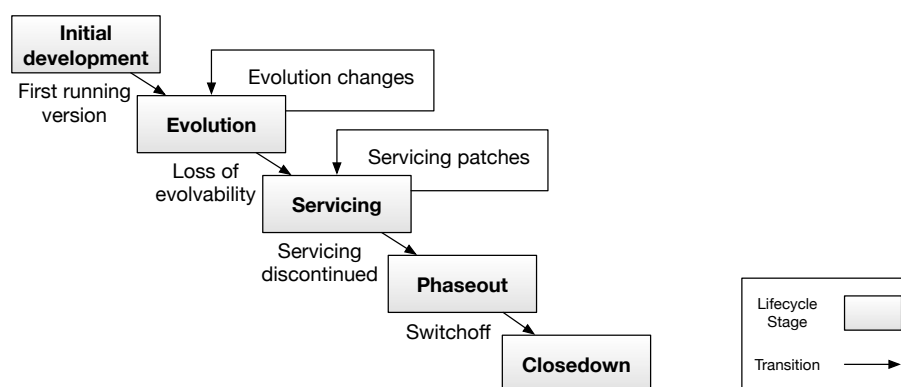


Figure 1.1 Staged software lifecycle [RB00]

As information systems becoming legacy is an established problem, four different solutions have emerged in practice [SWH10, pp.7-9]. The first solution is to overwork the legacy system to make it evolvable again. This is feasible if the loss of evolvability is caused by the degradation

of the system over time. However, it is not feasible if the loss of evolvability is caused by the degradation of the technological environment on which the legacy system depends on. In this thesis, we focus on the latter case.

The second solution is to redevelop the system from scratch. This solution can be applied if the environment of the legacy system became technologically obsolete, since it enables using state-of-the-art technologies and software architectures. Most notably, the resulting information system can be specifically designed to match changed requirements, possibly extending beyond the functionality of the legacy system.

However, due to the significant size of the information systems, redevelopment can become time-consuming and risky [Sne05]. Therefore, the third alternative solution is to buy Commercial Off-The-Shelf (COTS) software to replace it. As every COTS software needs to be adapted to match individual requirements, this solution is suitable if the effort for customization is low. However, it involves the risk of losing undocumented but business-critical functionality contained in the legacy system during the transition.

For this reason, the fourth alternative of migrating a legacy system to a new environment is seen as the commonsense solution when dealing with legacy systems [BS95, p.6]. Software migration is the transition of a legacy system to a new environment while retaining its data and functionality [Bis+99]. As in the other solutions, the information system re-enters the evolution stage after the transition has been performed and thereby regains its evolvability. Compared to redevelopment or the use of COTS software, the intention is to exclude the risk of having lost business-critical functionality after the phasing out of the legacy system. This is achieved by systematically transforming the system and thereby preserving its business-critical parts. Then, a software migration shifts the risk of failure to the transformation itself.

Software modernization is some kind of software migration that emphasizes the attempt of adapting the software system to the new environment. When modernizing a legacy system, the goal is not only to create an executable version of the system in the new environment that preserves its data and functionality but also to design the system for the new environment [Fle+07]. In this thesis, we focus on modernizing rather than just migrating legacy systems. We assume that this is essential to ensure the longevity of the system.

We use a real-world modernization scenario between platforms of the vendor Oracle as a continuous example, namely the transformation from Oracle Forms to Oracle ADF. The platform Oracle Forms, which originated in 1981¹, has been established to enable the development of information systems that consist of user interfaces which allow users to interact with an underlying database. Systems developed in Oracle Forms are written in a proprietary

¹http://www.trivadis.com/sites/default/files/downloads/Kopernikus_de.pdf (accessed March 22th, 2016)

Fourth-Generation Programming Language (4GL) and do employ a monolithic architecture. An example of a system developed in Oracle Forms can be seen in the left side of Figure 1.2.

In the right side of the figure, the same user interface is shown in the more recent platform Oracle ADF. Oracle ADF is a framework that aims to ease the development of information systems by providing infrastructure services and corresponding development tools. Systems developed in Oracle ADF are written in the object-oriented programming language Java and do employ a layered, service-oriented architecture. The framework extends the capabilities of Oracle Forms, for example, by enabling the development of systems that are optimized for mobile devices. This is one of the reasons why various companies consider transforming their Oracle Forms-based information system to Oracle ADF.

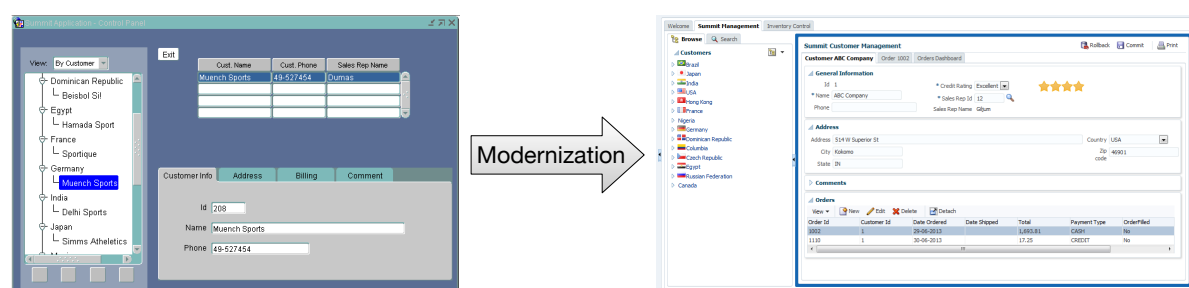


Figure 1.2 User interface of the Summit application, realized in the platform Oracle Forms (left) and in Oracle ADF (right)

An essential characteristic of each software modernization is the intended change of the environment [GW05]. This change can be diverse since it can affect every part of the technical environment used by the legacy system, examples being the hardware, operating system or runtime environment. The change of the environment then determines the required change that needs to occur in the transformation of the legacy system.

For example, a transformation from Oracle Forms to Oracle ADF will result in a change of the runtime environment, requiring the change of interfaces and the programming language used. Carefully analyzing such characteristics of a software modernization, planning and executing the transformation is the task of a software modernization project. Just like software development projects, software modernization projects are complex and therefore need to be carried out systematically. This can be achieved by enacting a *software modernization method*.

In general, a method describes all relevant aspects to guide a software engineering endeavor, such as a software modernization project [ES10]. A description comprises activities to be performed, artifacts to be generated, roles to be involved, tools to be utilized as well as relation between these concepts. For software development, the Rational Unified Process (RUP) [Kru03] or the V-Model XT [VMXe15] are examples for established methods that are enacted in corresponding projects in order to provide guidance.

For software modernization, such methods have been developed as well, an example being the Reference Migration Process (ReMiP) [SWH10, pp.85-131]. The contents of ReMiP have been derived by generalizing established methods, it can be seen as a reference method for software modernization. In Figure 1.3, an overview of its core disciplines is shown. A discipline is a categorization of similar contents within a method, e.g., activities with similar concerns are contained in the same discipline [OMG08a, p.161].

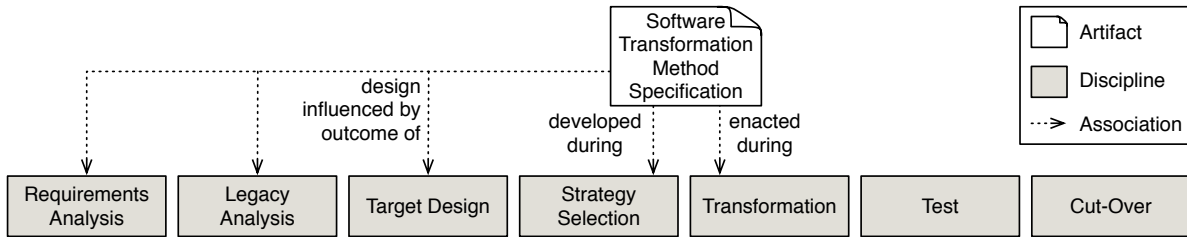


Figure 1.3 Core disciplines of a software modernization method and their relation to the development and enactment of a transformation method, based on ReMiP [SWH10]

When comparing methods to develop software with methods to modernize legacy systems, both have some disciplines in common. In both cases, requirements of the underlying project are analyzed, the target architecture of the resulting information system gets designed and the system is tested. But, there are some disciplines that are specific to software modernization methods, such as the need for analyzing the legacy system, choosing a transition strategy or performing the transformation and cut-over.

The *Transformation* discipline substitutes the *Implementation* discipline of a software development project, it comprises the use of a previously developed *Software Transformation Method*. Since a software modernization can be seen as the systematic transformation of artifacts that constitute a legacy system, a software transformation method is the part of a software modernization method which describes how to perform the transformation.

1.1 Software Transformation Methods

A software transformation method specifies for every part of the legacy system which activities to perform, roles to involve, tools to apply or artifacts to generate in order to transform the system technically. In other words, the *Transformation Method Specification* guides the technical transformation of a legacy system. Besides the transformation itself, it can also guide the development of tools that are required to automate parts of the transformation.

Transformation methods follow a software transformation strategy which describes a specific way of how to perform the transformation. Established strategies are conversion, reim-

plementation or wrapping [SWH10, pp.10-13]. A conversion strategy intends to automatically transform a legacy system by defining mappings between the programming languages involved. In contrast, when following a reimplementation strategy, a legacy system gets rewritten manually by developers. A wrapping strategy foresees to encapsulate a system by using a wrapper. The wrapper acts as a connection point to the legacy functionality in the resulting system.

Related to processes, transformation methods are instances of the established horseshoe model [KWC98]. This is due to the fact that software modernization is some kind of software reengineering, which comprises the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [CC90]. In general, reengineering methods consist of the consecutive phases *Reverse Engineering*, *Restructuring* and *Forward Engineering* which is represented by the horseshoe model shown in Figure 1.4.

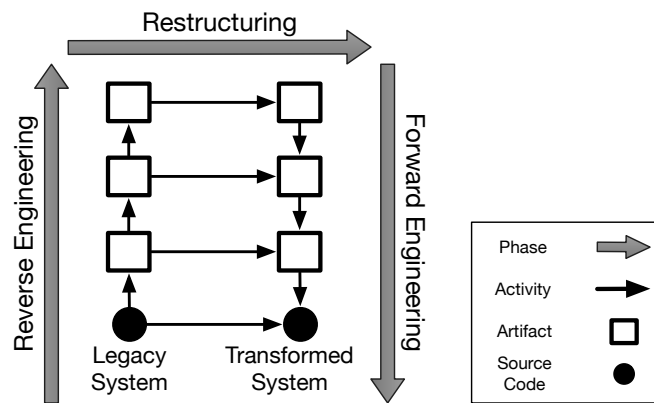


Figure 1.4 Conceptual representation of the horseshoe model, consisting of the consecutive phases reverse engineering, restructuring and forward engineering [KWC98]

During the transformation, the legacy system is represented by artifacts on different levels of abstraction, e.g., on a syntactical or architectural level. Representations on higher levels of abstraction are reverse engineered by enacting activities that apply parsing or clustering techniques. Then, the representations are restructured to adapt the system to the new environment, for example, by changing its architectural style. Finally, forward engineering is applied to concretize them in the target environment, possibly generating runnable source code.

The standardization of artifacts used by a software transformation method has been the goal of the Object Management Group (OMG). For this purpose, the Architecture-Driven Modernization (ADM) task force² has been established which aims to apply principles from the domain of Model-Driven Engineering (MDE) on the domain of software modernization. Up to the present day, several standards for these models have been defined in terms of metamodels, others are in development.

²<http://adm.omg.org> (accessed March 22th, 2016)

Method Development

During a software modernization project, a software transformation method needs to be developed. The development of such a method is a critical task, as it influences the overall efficiency and effectiveness of the software modernization project. Effectiveness in this context relates to properties of the resulting information system, e.g., its resulting non-functional properties. Efficiency relates to properties of the process that is performed to realize the transformation, e.g., the budget or time required. An efficient and effective transformation method satisfies constraints on the described properties, e.g., it minimizes the effort required while certain non-functional properties are maximized.

To be efficient and effective, the transformation method needs to take into account the situational context of the project and is then called situation-specific. This relation between the situational context and the development of a transformation method is shown in Figure 1.3. The outcomes of activities preceding the *Strategy Selection* discipline form the situational context, they represent influence factors on the development of a transformation method. This comprises goals of stakeholders, characteristics of the legacy system, the change of the environment or the designed target architecture. These influence factors need to be considered, in order to develop a situation-specific transformation method

To demonstrate the implications of a transformation method on the efficiency and effectiveness of a modernization project, we provide an example based on the transformation from Oracle Forms to Oracle ADF. In this example, consider the following two functionalities, i.e., concepts [KNE92], realized by a legacy system. First, the system contains *Dialogs*, i.e., graphical user interfaces. Second, some of these dialogs contain buttons that enable to perform a navigation flow to another dialog, i.e., a *Dialog Flow*. The imperative source code shown in the left side of Figure 1.5 realizes such a dialog flow concept. It gets executed whenever a user presses an associated button. The invocation of the platform-specific function `call_form` triggers the change to another dialog, i.e., to a dialog which can be used to manage properties.

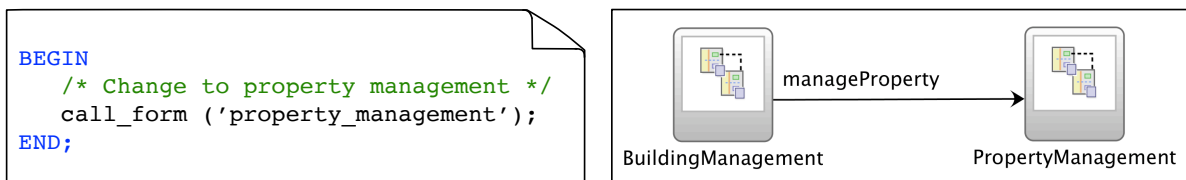


Figure 1.5 Imperative realization of a *Dialog Flow* concept in the source environment (left), declarative realization in the target environment (right)

Based on this example, we describe two situations in which the realization of the dialog flow in the target environment varies. In the first situation, we envision to realize dialog flows by

imperative source code, too. This means, it is also required to call a platform specific function in the target environment. Then, using a transformation method T_1 that specifies to perform an automatic conversion on a syntactical level can be efficient and effective. Following this strategy results in the desired realization and enables transforming large parts of the system automatically, as it only required to develop a parser, a code generator, and a mapping between the syntactic elements of the languages.

In the second situation, the realization in the target environment might be significantly different. Such a case can be seen in the right side of Figure 1.5. In this situation, dialog flows should be realized by using the provided declarative language. The language enables referencing dialogs (`BuildingManagement`, `PropertyManagement`) and to define flows between them (`manageProperty`). Then, using the same method T_1 that prescribes to perform a transformation on a syntactical level bears the risk to preserve the imperative realization in the target environment [Byr92], possibly by emulating the platform specific function. As the functionality would be preserved, but its realization would not be adapted to the target environment, we consider the method to be ineffective [Fle+07].

Using a method T_2 that prescribes to perform a transformation on a higher level of abstraction by extracting the underlying concept would increase the effectiveness of an automatic transformation. Following this strategy requires extracting contained dialogs and flows between them by interpreting the source code. But, this will influence the efficiency of the method as sophisticated program comprehension techniques are required.

If an automatic transformation is either inefficient or ineffective, a method T_3 that prescribes a guided manual or semi-automatic reimplementation can be a viable alternative.

1.2 Problem Statement

Developing a situation-specific transformation method is a challenging task, since an ill-designed transformation method can result in a modernization project becoming inefficient or ineffective and therefore raises the risk of failure. To counteract this problem, it is current practice to support the development of transformation methods by a method engineering approach. Such approaches usually provide methods that have been successfully applied in practice before or guide the development of new methods.

The applicability of any method engineering approach mainly depends on its degree of *controlled flexibility*. Thereby, *flexibility* refers to the degree of freedom given during the development of a method to adapt the method to the situation at hand. However, flexibility shall not come at the expense of decrease quality of the resulting method, wherefore the development additionally needs to be controlled. In this context, *control* refers to the degree of guidance

given during the development of the method. Without such guidance, it is not possible to ensure the result of the development, e.g., the correctness or quality of the method. In [HBO94] a categorization is introduced by which methods are categorized based on their degree of controlled flexibility. We applied this categorization on state-of-the-art software transformation methods [GFB15], shown in Figure 1.6.

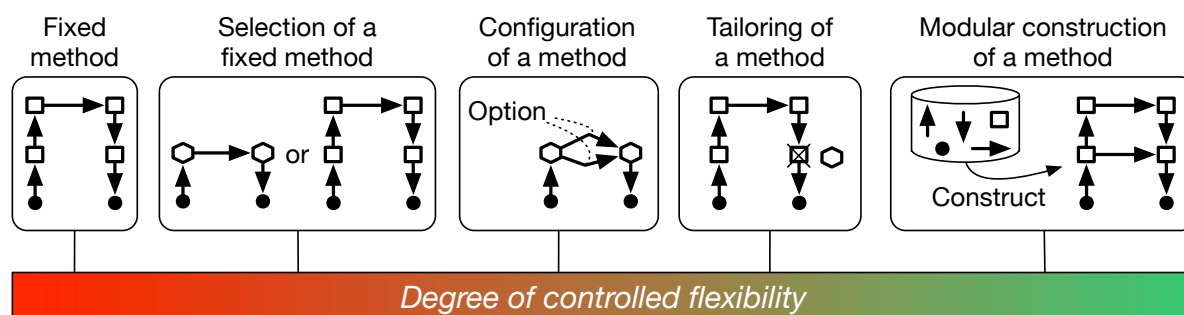


Figure 1.6 Categorization of transformation methods according to their degree of controlled flexibility, based on [HBO94]

Fixed methods are those which do not foresee any situation-specific adaptation as they describe a static set of activities to perform, tools to use, artifacts to generate or roles to include in order to transform a legacy system. In general, the assumed situational context for which the method is applicable is only described implicitly. If the situational context for a set of fixed methods is made explicit, it allows a *selection* of the most suitable one, which can be seen as a more flexible approach to perform situation-specific adaptation. But, since the resulting method will be *fixed*, the adaptability is still limited. A more flexible approach is the definition of a base method that allows *adaptation*. The *configuration* of foreseen variation points can be seen as a special approach to perform an adaptation. In general, arbitrary changes, i.e., *tailorings*, of the base method can be allowed. However, if the base method is comprehensive but only a small part is needed, or if the base method already considers a different situational context than the one observed, performing modifications to derive a valid method can be expensive.

The modular construction of transformation methods by assembling predefined method parts provides the highest degree of controlled flexibility. Besides providing method parts, an approach for modular construction also requires providing construction guidelines on how to perform the assembly itself. Then, in the case that no reusable situation-specific method is available, approaches of this category are advantageous compared to developing a method from scratch since they provide a high degree of controlled flexibility. Unfortunately, approaches that address the modular construction of transformation methods are hardly available. As transformation methods are some kind of reengineering methods, one could argue that software reengineering frameworks can be used to modularly construct transformation methods, an

example being MoDisco [Bru+14]. Although these frameworks are useful in practice to implement tools that are used by transformation methods, they fall short in providing guidance on how to systematically construct the method itself.

During industrial projects in which transformations from Oracle Forms to Oracle ADF were performed repeatedly, we observed such aforementioned situations that no applicable method was available up front or could be derived by adapting existing methods with justifiable effort. In such instances, the lack of support in the development of situation-specific transformation methods hinders a tool-supported modernization and partially promotes redevelopment instead. This is summarized in the following problem statement that forms the basis of this thesis:

**How to enable the flexible but controlled construction of
situation-specific software transformation methods to enable
tool-supported software modernization of information systems in practice?**

1.3 Requirements

First of all, we require that the solution concept provides a high degree of controlled flexibility. However, additional requirements need to be addressed in order to be applicable in a software modernization scenario. These requirements are described subsequently.

Generality The requirement for *generality* claims that the solution concept shall not be limited to a specific environmental change. To fulfill this requirement, the solution concept cannot assume that the legacy system was developed in a specific technology or employs a specific software architecture.

Granularity The requirement for *granularity* claims that the solution concept shall enable to adapt the granularity of the resulting transformation method specification during its development. To fulfill this requirement, the solution concept needs to enable the development of a fine-granular method specification as well as a coarse-granular one.

Versatility The requirement for *versatility* claims that the solution concept shall support the definition of different transformation strategies. The more transformation strategies are provided, the better a method can be adapted to the situation at hand.

Continuity The requirement for *continuity* claims that the use of tools during the enactment of a developed transformation method shall be guided. We assume that considering the use of tools is essential for an acceptance of a method in practice. To fulfill this requirement, the solution concept can either provide guidance in the development of custom tools or provide predefined ones and guide their usage.

Formalization The requirement for *formalization* claims that a developed transformation method shall be specified formally. A formal specification enables a precise description of the method which is essential, e.g., to reuse the method in subsequent modernization projects.

1.4 Solution Concept

In this thesis we propose a Method Engineering Framework for Situation-Specific Software Transformation Methods (MEFiSTo) to address the problem statement identified. The framework is based on Situational Method Engineering (SME) concepts which is an established engineering discipline to systematically develop situation-specific methods [HS+14, p.5]. The general idea is to construct methods by selecting and assembling predefined method fragments that are stored in a repository, called a method base. A method fragment is a reusable, atomic building block of a method [HS+14, p.4]. It can be any constituent of a method, namely an activity, artifact, tool or role.

The proposed framework transfers SME concepts to the domain of software modernization and provides a foundation that enables the modular construction and enactment of situation-specific and model-driven software transformation methods. The method engineering process of the framework is shown in Figure 1.7.

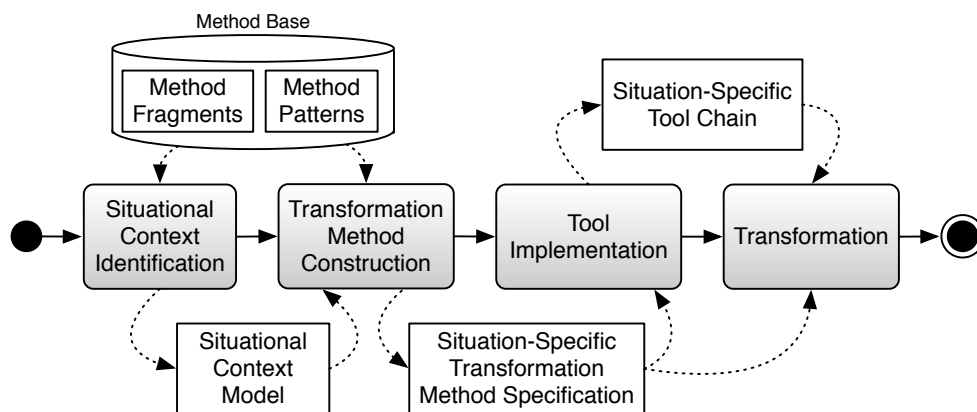


Figure 1.7 Activities and artifacts of the method engineering process for the modular construction and enactment of a situation-specific transformation method

In the beginning, the situational context of the modernization project needs to be identified. This comprises identifying characteristics of the legacy system, the target system as well as the project itself, like the amount of software developers available. The knowledge of the context is an essential prerequisite in order to construct a situation-specific transformation method.

Using the gained knowledge of the situational context, the transformation method gets constructed before related tools are implemented. The construction is essentially supported by

a method base which is a repository that contains predefined method fragments and method patterns. Method fragments are atomic building blocks of transformation methods. In the framework, the fragments are based on principles from the domain of model-driven engineering. This enables performing abstractions during the transformation, which is particularly important in a software modernization scenario.

Method patterns encode proven transformation strategies. In particular, a pattern defines construction guidelines for a method that follows the encoded strategy by defining constraints over the method fragments, i.e., by defining which ones to use in order to realize the strategy. In addition, each pattern, i.e., each strategy, is associated with a set of characteristics that determine its suitability in a given situation.

During the construction of a transformation method, an expert matches the characteristics of the available method patterns against the characteristics of the current situation and selects the most suitable ones. Based on the construction guidelines provided by the selected patterns, the actual transformation method is constructed. The resulting specification guides the subsequent development of tools that are required to automate the transformation. Finally, the actual transformation of the legacy system is performed.

1.5 Overview of Publications

Various papers have been published in peer-reviewed workshops or conferences that are related to the solution concept defined in this thesis. In Figure 1.8 we classify the publications according to the area of *Foundations* for the solution concept, the *Solution Concept* itself as well as the *Application of the Solution Concept*.

The solution concept defined in this thesis is essentially based on foundations from three research areas, namely software reengineering, model-driven engineering and method engineering. We discussed the idea of combining principles from the former two areas in [GG12]. In particular, we sketched the general idea of applying model-driven engineering on the domain of software modernization, expected benefits and challenges. In [GS13], we concretized this idea by discussing its principles against the background of an industrial context that arose from a project in which we were working.

Related to the area of method engineering, we introduced a software development method that focuses on synchronizing globally distributed software development teams using artifacts [FBGS15]. Guiding developers in implementation tasks by using explicit specifications also became an important part within the solution concept developed.

The solution concept, i.e., the MEFiSTo framework, has been introduced in [Gri+16]. We described the general idea and gave an overview of the method engineering process. We

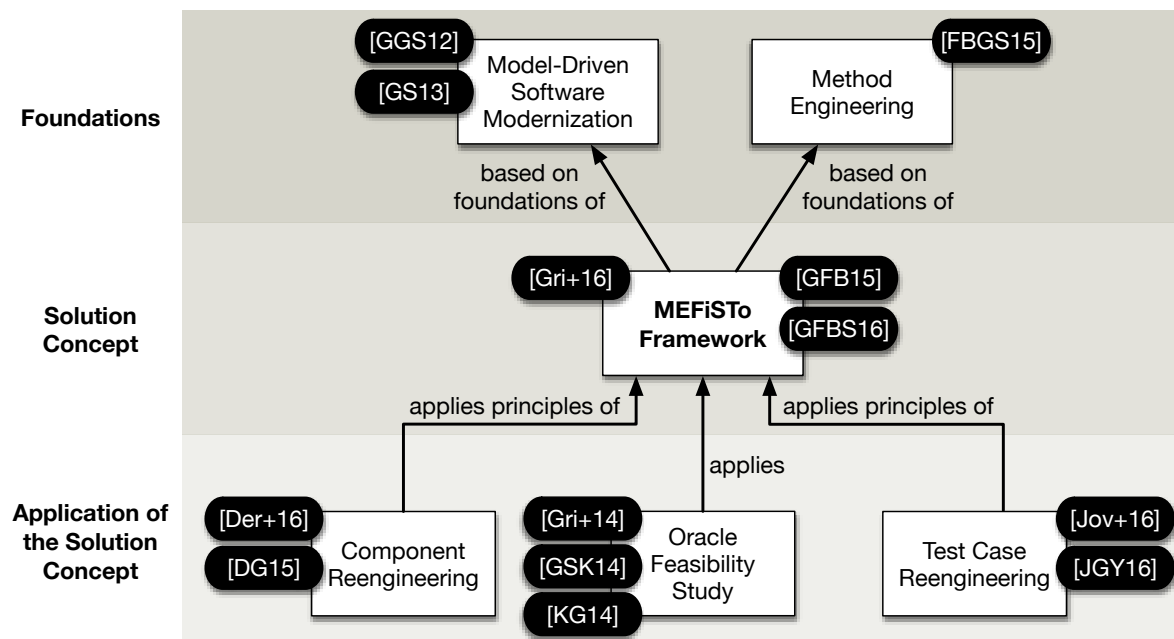


Figure 1.8 Overview of the publications related to this thesis

focused on modeling, i.e., decomposing, a software system by a set of concepts and evaluating the applicability of method patterns. In addition, we discussed the findings of the feasibility studies performed. In [GFB15], we discussed the shortcomings of current method engineering approaches to support the development of transformation methods. In addition, we described how our solution concept addresses these shortcomings. In [GFBS16], we described details of an essential part of the framework, namely the content of its method base. In particular, we described the contained method fragments and method patterns.

The MEFiSTo framework was developed as part of an industrial project that was centered around software of the vendor Oracle. We directly applied the framework in this project and published the results. In [GSK14], we described a process to semi-automatically restructure a legacy system during its transformation on an architectural level. The process is based on combining hierarchical and partitioning clustering whereby intermediate results are validated by system experts. In [Gri+14], we described a method to gather feedback from software developers during a transformation to systematically improve a developed transformation method. In [KGB14], we described the principles of MEFiSTo for the Oracle community.

We transferred principles of the developed solution concept to other fields in which we were working. In [DG15], we motivated that the flexible definition of transformation methods enables increasing the longevity of Model-Integrating Components (MoCos) [Der+14]. A MoCo is a non-redundant, reusable and executable combination of logically related models and code in an integrated form where both parts are stored together in one component. In [Der+16],

we briefly described the role of the MEFiSTo framework to transform existing components to MoCos. In [Jov+16] and [JGY16], we sketched the idea of how to perform the model-driven transformation of test cases in the context of a modernization scenario.

1.6 Structure of this Thesis

An overview of the structure of this thesis is shown in Figure 1.9. As can be seen, the thesis is essentially separated into three parts, namely *Foundations and Related Work*, *Solution Concept* and *Evaluation and Conclusion*.

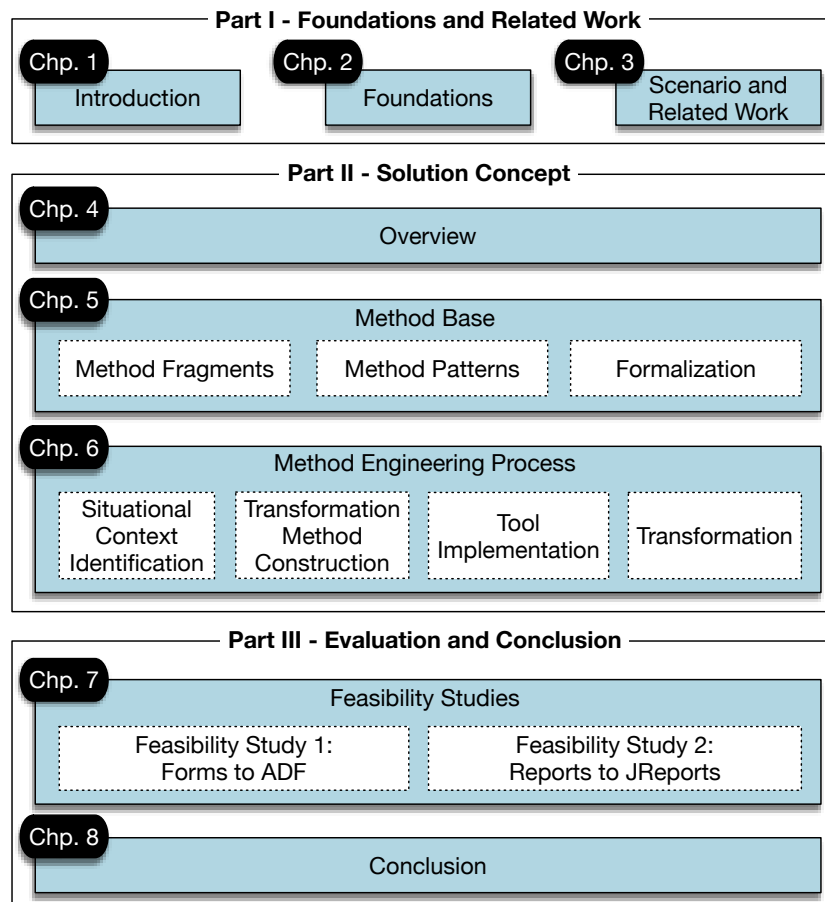


Figure 1.9 Overview of the structure of this thesis

In Chapter 2, we describe foundations from different research areas that are related to the solution concept. In particular, we describe foundations from the fields of model-driven engineering, method engineering and software reengineering,

In Chapter 3, we introduce related work of this thesis. For this purpose, we first describe a modernization scenario that resulted in the problem statement of this thesis. Based on the

scenario, we derive a set of requirements that a solution concept needs to fulfill. We identify and classify existing approaches and evaluate them against the requirements.

In Chapter 4, we give an overview of the solution concept defined in this thesis. We describe the general idea of how to construct transformation methods using method fragments and method patterns. We state a set of evaluation criteria whose fulfillment we aim to discuss by feasibility studies and introduce an example that is used continuously in the thesis.

In Chapter 5, we introduce the first main constituent of the solution concept, namely the method base. We state a set of requirements that the method base shall fulfill. Thereafter, we introduce the content of the method base, i.e., we introduce the contained method fragments and method patterns. For each pattern, we provide a comprehensive example that we use to discuss the characteristics of the pattern. Additionally, we describe how we formalize the content of the method base and developed transformation method specifications.

In Chapter 6, we introduce the second main constituent of the solution concept, namely the method engineering process. We state a set of requirements that the process shall fulfill. Thereafter, we describe the core activities of the process. We discuss the purpose and emphases for each activity. Also, we discuss quality characteristics of arising artifacts and provide detailed examples.

In Chapter 7, we describe the feasibility studies performed to demonstrate the applicability of the developed framework in practice. We performed two feasibility studies in which we transformed real-world legacy systems. The systems differed in the technologies in which they have been realized. We use the findings of the studies to discuss the evaluation criteria that were introduced in Chapter 4.

In Chapter 8, we summarize the main contributions of this thesis. In addition, we describe how the solution concept fulfills the requirements that were introduced in Chapter 3. Finally, we sketch future work by describing possible enhancements of the MEFiSTo framework.

In this chapter, we give an overview of foundations that are relevant for this thesis. We classified these foundations along the research areas that form the basis for the developed solution concept, as illustrated in Figure 2.1. Subsequently, we introduce essential foundations of *Model-Driven Engineering* (MDE) (Section 2.1), *Method Engineering* (Section 2.2), and *Software Reengineering* (Section 2.3).

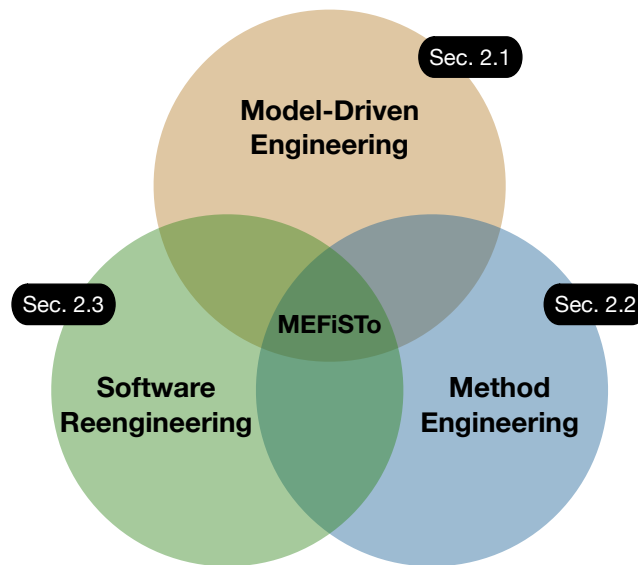


Figure 2.1 Research areas related to this thesis

2.1 Model-Driven Engineering

In this section, we introduce foundations in the area of *Model-Driven Engineering* (MDE). Intuitively, MDE is a paradigm that uses models as the primary artifacts when performing

a software engineering task [BCW12, pp.9-10]. We first describe the general principles of MDE by introducing the Meta-Object Facility (MOF). Thereafter, we introduce two manifestations of MDE, namely the Model-Driven Architecture (MDA) and the Architecture-Driven Modernization (ADM).

2.1.1 Meta-Object Facility

The *Meta-Object Facility (MOF)* standard [OMG15a] defined by the Object Management Group (OMG) can be seen as a framework for metadata that has been designed to support model-driven engineering. Intuitively, it enables establishing a layered architecture that can serve as a common basis to develop or include models and modeling languages. A MOF-based architecture can consist of an infinite amount of so-called *metamodeling layers*, while at least two are required. In practice, it is common to use four layers as shown in Figure 2.2.

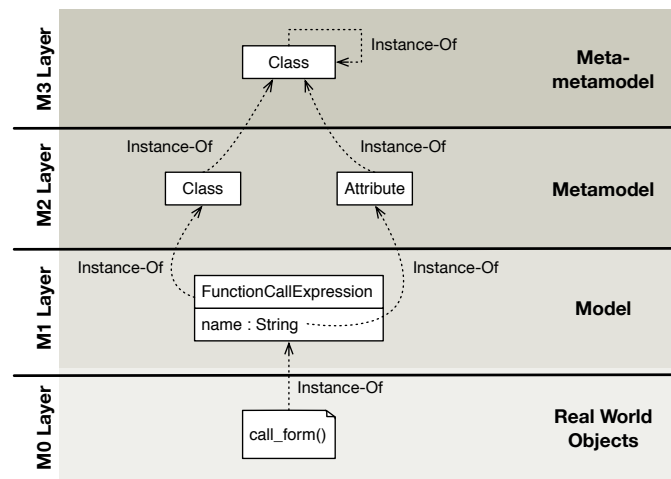


Figure 2.2 Exemplary Meta-Object Facility-based architecture, based on [BCW12, p.14]

On the *M0 Layer* at the bottom of Figure 2.2, objects of the real world reside. These are the objects that we aim to describe by a model. In this example, textual source code is shown. The source code represents the invocation of a function called `call_form`.

On the *M1 Layer*, models of real world objects reside. A model is associated with three characteristics: it should be a representation of something existing in the real world (*Mapping*), while only representing certain aspects (*Reduction*) that are relevant for an intended goal (*Pragmatism*) [Sta73]. In this example, we model the function invocation as a `FunctionCallExpression` which has a name.

To express such a model, we need an associated modeling language. Such a language resides on the *M2 Layer* as a *Metamodel*. A metamodel defines concepts (as metamodel classes) that shall be represented, relationships between these concepts and possibly logical

assertions [BG01]. Existing general-purpose modeling languages for which metamodels are available are, for example, the Unified Modeling Language (UML) [OMG15b] or the Knowledge Discovery Metamodel (KDM) [OMG11b]. However, modeling languages can also be (self-)defined for a specific domain or context. Such a language is called a Domain-Specific Language (DSL). In the example, the `FunctionCallExpression` is an instance of the `Class` concept, while its `name` is an instance of an associated `Attribute`.

We also need a modeling language to express a metamodel. This is the role of a *Meta-model*, which resides on the *M3 Layer*. The MOF standard [OMG15a] defines such a self-describing language which, among other things, defines the concept of a `Class`. The Eclipse Modeling Framework (EMF)¹ provides an implementation of a part of the MOF standard. The corresponding metamodel is called *ECore*².

2.1.2 Model-Driven Architecture

The *Model-Driven Architecture (MDA)* follows the MDE paradigm and was proposed by the OMG [OMG14]. In particular, MDA defines a Model-Driven Development (MDD) process for software systems. An overview of this process can be seen in Figure 2.3.

The general idea of MDA is to distinguish several abstraction levels on which different types of models reside. For each type of model, metamodels are proposed by the OMG. Starting on a high-level of abstraction, the abstraction level is stepwise and systematically lowered until the source code for the software system can be derived. The transition from one model to another is realized by model transformations. Finally, a code generator is used to generate the source code of the system.

A *Computation-Independent Model (CIM)* describes the software system to develop, independent of its technical realization [BCW12, pp.40-41]. This comprises, for example, modeling requirements or business processes. The derived *Platform-Independent Model (PIM)* describes the technical realization of the system, independent of a specific platform, i.e., technology. This comprises, for example, the software architecture of the system or its deployment. Subsequently, the PIM is transformed into a *Platform-Specific Model (PSM)* which represents the system using technology-specific details. By separating platform-independent and platform-specific representations, PSMs for different technologies can be derived from a single PIM. The PSM should describe the system in sufficient technical detail, so that code can be generated. Thereby, the code can either be generated completely, or partially. The latter case requires that software developers complete the generated code.

¹<https://eclipse.org/modeling/emf/> (accessed March 22th, 2016)

²<http://download.eclipse.org/modeling/emf/emf/javadoc/org/eclipse/emf/ecore/package-summary.html> (accessed March 22th, 2016)

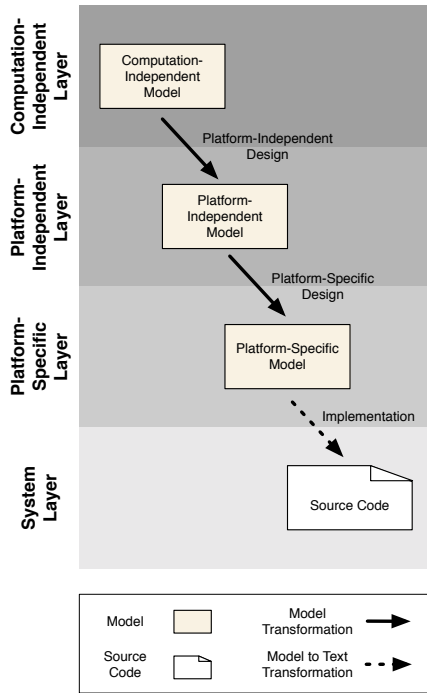


Figure 2.3 Artifacts and abstraction levels of the Model-Driven Architecture [BCW12, p.41]

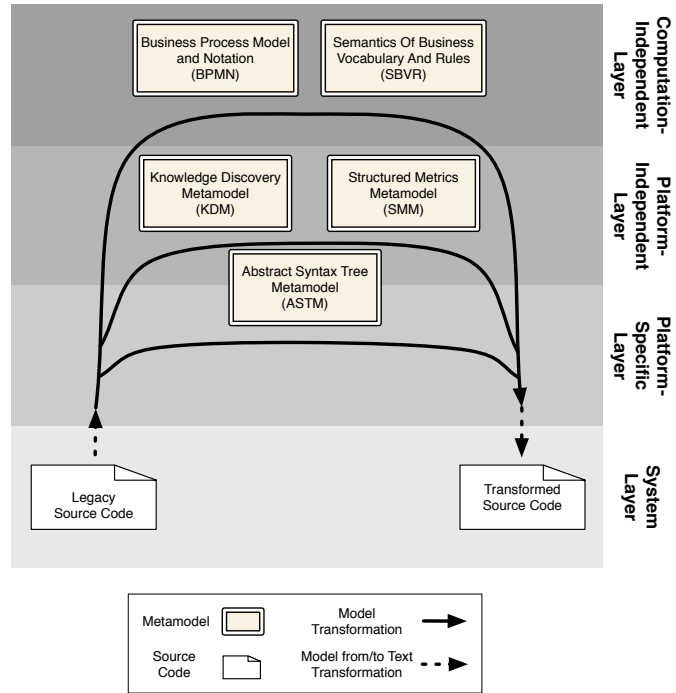


Figure 2.4 Modeling languages (excerpt) and abstraction levels of the Architecture-Driven Modernization, based on [PCDGP11] and [BCW12, pp.45-47]

2.1.3 Architecture-Driven Modernization

The *Architecture-Driven Modernization (ADM)* also follows the MDE paradigm and was proposed by the OMG, more specifically, by the *Architecture-Driven Modernization Task Force (ADMTF)*³ [UN10]. However, it cannot be applied to develop new software systems but to modernize existing ones. An overview of the ADM paradigm can be seen in Figure 2.4.

The general idea of ADM is to distinguish several levels of abstractions which can be aligned with the ones of the MDA [BCW12, pp.45-47]. For each level, the OMG proposes various standards like the Abstract Syntax Tree Metamodel (ASTM) or the Knowledge Discovery Metamodel (KDM) [PCDGP11]. Based on this context, a transformation method (cf. Section 2.3.1) shall be enacted that spans over the proposed abstraction levels to transform the existing *Legacy Source Code* into the desired *Transformed Source Code*.

For this purpose, the ADMTF discusses the possible implications when using the proposed abstraction levels [UN10, pp.18-19]. For example, solely using the platform-specific layer enables a *physical transformation* but prevents substantial changes to the system. Such kind of changes, i.e., changing the architecture of the system or even its requirements, requires using

³<http://adm.omg.org> (accessed March 22th, 2016)

higher levels of abstraction. However, the specification of a precise transformation method, i.e., which artifacts to generate or activities to enact, is missing [BR15, p.140]. In fact, even the relation between proposed standards is not well-defined [DLGB12].

Subsequently, we provide additional details about two metamodels proposed by the ADMTF that are used within this thesis, namely ASTM and KDM.

Abstract Syntax Tree Metamodel

The Abstract Syntax Tree Metamodel (ASTM) [OMG11a] is a metamodel that enables modeling the Abstract Syntax Graph (ASG) of source code (cf. Section 2.3.4). An example of the intended use of the ASTM can be seen in Figure 2.5.

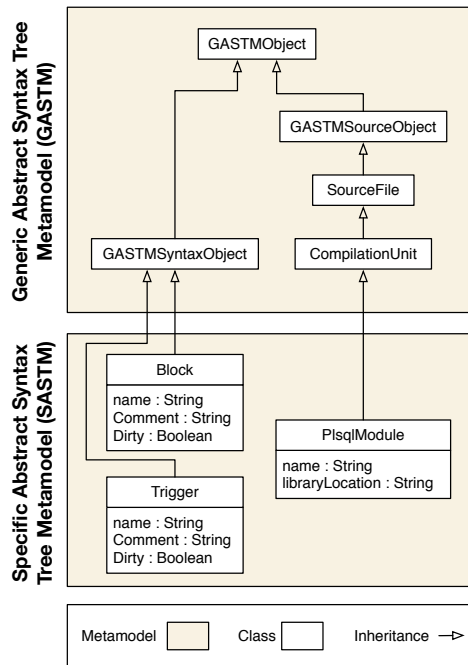


Figure 2.5 Abstract Syntax Tree Metamodel, based on [OMG11a]

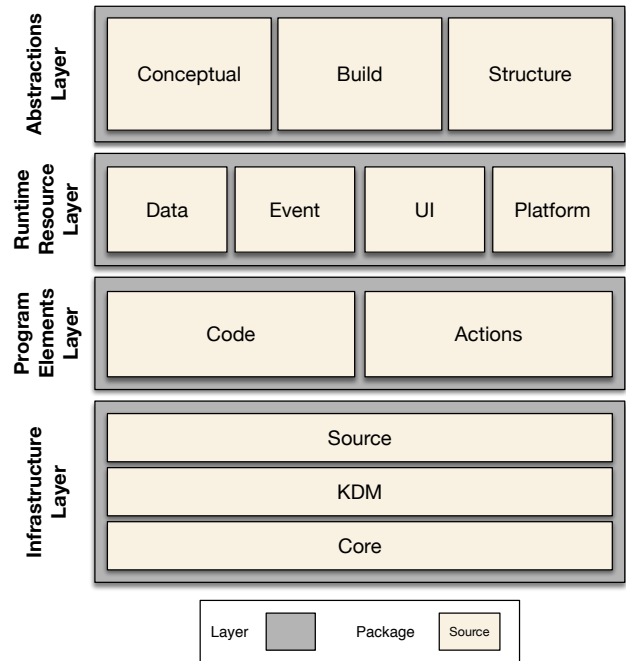


Figure 2.6 Packages and layers of the Knowledge Discovery Metamodel, based on [OMG11b]

The general idea of ASTM is to distinguish different types of metamodels, whereby two of them are relevant for this thesis. On the one hand, the provided specification defines a *Generic Abstract Syntax Tree Metamodel (GASTM)*. This metamodel provides means to model generic language constructs that are common to programming languages, an excerpt can be seen in the upper part of Figure 2.5. For example, a **CompilationUnit** is an entity that contains source code, while a **GASTMSyntaxObject** represents a syntactic element.

On the other hand, it is foreseen to define a *Specialized Abstract Syntax Tree Metamodel (SASTM)*. This metamodel shall extend the GASTM and represent a specific programming

language. However, such a metamodel is not defined within the specification but needs to be self-defined. An excerpt of an SASTM for the programming language of Oracle Forms can be seen in the lower part of Figure 2.5. The SASTM defines the syntactic elements of Blocks and Triggers as well as the `PlsqlModule` which acts as a container for PL/SQL source code.

Knowledge Discovery Metamodel

The Knowledge Discovery Metamodel (KDM) [OMG11b] is a metamodel that enables the integrated modeling of information about a software system on various levels of abstraction. An overview of the packages and layers defined within the KDM can be seen in Figure 2.6.

The lowest layer of abstraction, i.e., the *Infrastructure Layer*, serves two purposes: First, the packages called *Core* and *KDM* define common metamodel classes that are used within other packages. Among other things, these elements provide an extension and annotation mechanism. Second, the *Source* package establishes the link to the software system that shall be modeled by providing means to describe its artifacts, like its directories or source code files.

The packages of the *Program Elements Layer* can be used to model a detailed representation of the source code. However, while ASTM provides means to model the details of single language elements, KDM is focused on modeling the control and data flow (cf. Section 2.3.4). In this sense, ASTM and KDM complement each other. However, the relation between these metamodels is not well-defined [DLGB12].

The packages of the remaining layers can be used to model information that are usually only implicitly visible by the source code. On the one hand, the *Runtime Resource Layer* covers information whose extraction usually requires knowledge of the runtime, e.g., event-based states of the system or the structure of user interfaces. On the other hand, the *Abstractions Layer* covers information whose extraction usually requires knowledge of the domain, e.g., business rules or the architecture of the system.

2.2 Method Engineering

In this section, we introduce foundations in the area of *method engineering* which is the discipline to systematically develop or adapt methods [Bri96]. The purpose of a method is to guide a complex software engineering endeavor, like the development of a software system or its transformation. A method describes this endeavor by specifying the activities to enact, artifacts to generate, tools to use or roles to involve [ES10]. Intuitively, a method is a guideline for the endeavor. As the term of a method is a central one in this thesis, we define it as follows:

Notation 1 (*Method*)

A method is a description of how to systematically perform an endeavor. This comprises a process and its contained activities, artifacts, roles, tools and relationships between these elements on varying levels of granularity.

Subsequently, we first describe a specific manifestation of method engineering, namely Situational Method Engineering (SME). Thereafter, we introduce the Software and Systems Process Engineering Metamodel (SPEM) [OMG08a] which can be used to formally specify methods. In the end, we describe the use of metamodeling in the context of method engineering and discuss differences to MDE.

2.2.1 Situational Method Engineering

Situational Method Engineering (SME) is a kind of method engineering which encompasses all aspects of creating a method for a specific situation [HS+14, p.5]. Intuitively, approaches that follow the SME paradigm consider the situational context in which a method will be applied during the development of the method. Due to this, the method can be adapted to the context and is then called situation-specific.

An SME approach can be realized in various ways, a general classification is described in Section 3.3.1. For this thesis, the class of approaches that enable the *modular construction* of situation-specific methods is particularly important. Those approaches usually define two essential constituents as shown in Figure 1.7 on page 12. On the one hand, a *method base* is provided which constitutes a repository that contains reusable building blocks of methods. On the other hand, a *method engineering process* is defined to systematically construct a method.

The building blocks of methods are called method parts, whereby different types can be distinguished. Common examples are *method fragments*, *method chunks* or *method components* [HS+14, pp.38-45]. A method fragment can be seen as an atomic building block of a method, while chunks as well as components aggregate multiple fragments. In this thesis, we focus on using method fragments as building blocks and define them as follows:

Notation 2 (*Method Fragment*)

A method fragment is a reusable, atomic building block of a method, i.e., a single activity, artifact, role or tool.

Developing a complete method by solely using method fragments is a cumbersome task, as methods can become large in practice. One way to address this problem is to use larger method parts than method fragments. This increases the efficiency of the method development as fewer

elements of the method base need to be considered [HSGPR08]. The solution concept of this thesis follows another way by using *method patterns*.

A pattern in general is associated with a reoccurring problem in a certain context [AIS77]. For the associated problem, it describes the core of a solution. A method pattern transfers this idea to the field of method engineering [FBLE13]. Each pattern is associated with a problem that shall be addressed by enacting a method. The solution to the problem is encoded by the pattern in the form of construction guidelines for the method [RP96]. As method patterns are an essential part of this thesis, we define them as follows:

Notation 3 (Method Pattern)

A method pattern is associated with a problem that shall be addressed by enacting a method. It encodes the solution in the form of construction guidelines for a method, i.e., it specifies which method fragments to use and how to assemble them.

2.2.2 Software and Systems Process Engineering Metamodel

It has been shown that models are well-suited to formally specify methods [RDR03]. A well-established metamodel for this purpose is the Software and Systems Process Engineering Metamodel (SPEM) [OMG08a]. An overview of its packages and an excerpt of the contained metamodel classes can be seen in Figure 2.7.

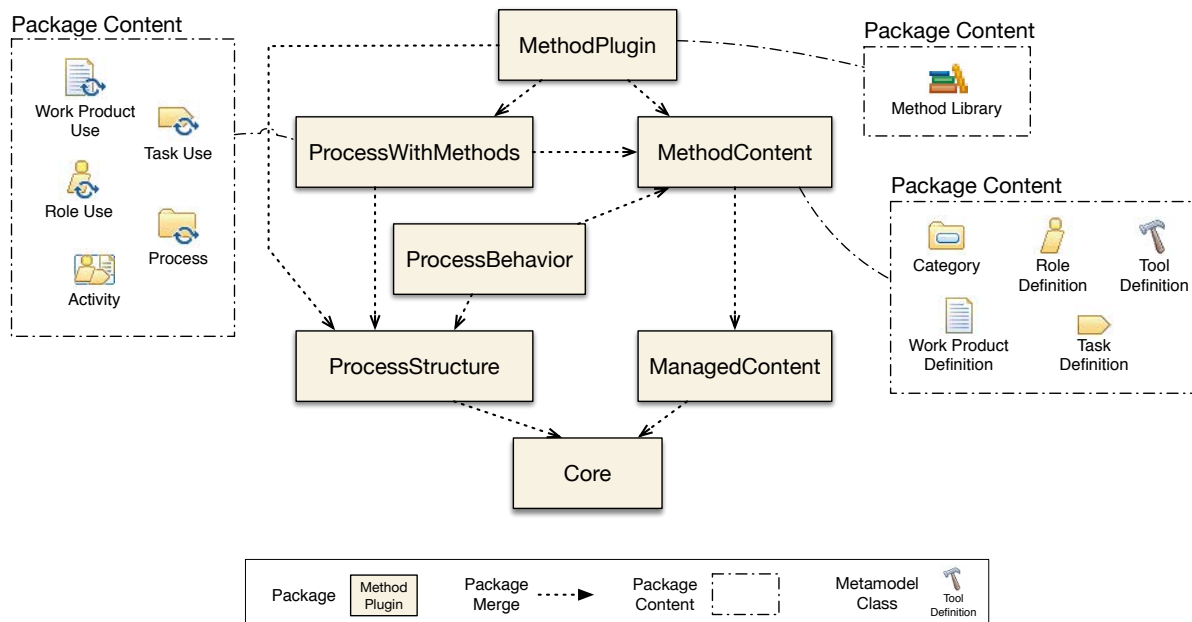


Figure 2.7 Packages and metamodel classes of the Software and Systems Process Engineering Metamodel (SPEM)

A core principle of SPEM is to follow a clear separation between reusable method content and its use as part of a process within a project [OMG08a, pp.12-14]. This can also be seen based on the package structure of the metamodel. The *Core* package defines base metamodel classes used by the other packages. For example, a Work Product Definition, i.e., an artifact, is a specialization of the Work Definition class defined in the core package. The reusable method content is defined by the *Method Content* package. Metamodel classes contained in this package enable to specify essential parts of a method, i.e., activities (Task Definition), artifacts (Work Product Definition), roles (Role Definition) and tools (Tool Definition). In addition, a Category enables classifying method parts. For example, a Discipline is a Category that is used to aggregate similar activities within a method, i.e., tasks with similar concerns are contained in the same discipline.

While activities can be defined as tasks using the method content package, it is not possible to express the control flow between different tasks. This requires using metamodel classes of the *Process With Methods* package, which enables expressing Processes. When specifying a process, elements of the reusable method content are referenced. For example, a process may contain multiple Task Use elements which reference actual tasks defined in the method content package and are related to each other by control flow dependencies.

Besides specifying methods, SPEM also supports managing them. By using capabilities of the *Method Plugin* package, specified method content or processes can be bundled in terms of a Method Library. In addition, the way in which libraries extend or modify each other can be specified which enables a systematic reuse of developed methods.

2.2.3 Metamodeling Layers

When specifying a method by using a model, different metamodeling layers (cf. Section 2.1.1) need to be distinguished. However, when comparing those layers to a pure MDE context, differences can be observed. This is particularly the case, if the method itself describes a model-driven approach. To discuss these differences, an example of a model-driven transformation method is shown in Figure 2.8.

In the figure, a four layered MOF-based architecture is shown. Compared to an MDE context, no differences arise on the two upper layers. On the *M3* layer, a *Meta-metamodel* resides, e.g., as specified in [OMG15a]. The *M2 Layer* contains metamodels, i.e., modeling languages, which can be used to specify methods. An example is SPEM, which defines Artifacts (i.e., work products), Activities (i.e., tasks) and Tools.

On the *M1 Layer*, models of the real world reside. In an MDE context, this usually comprises models of a software system to develop. However, in a method engineering context, the model describes activities which a human or a tool shall perform and the artifacts involved.

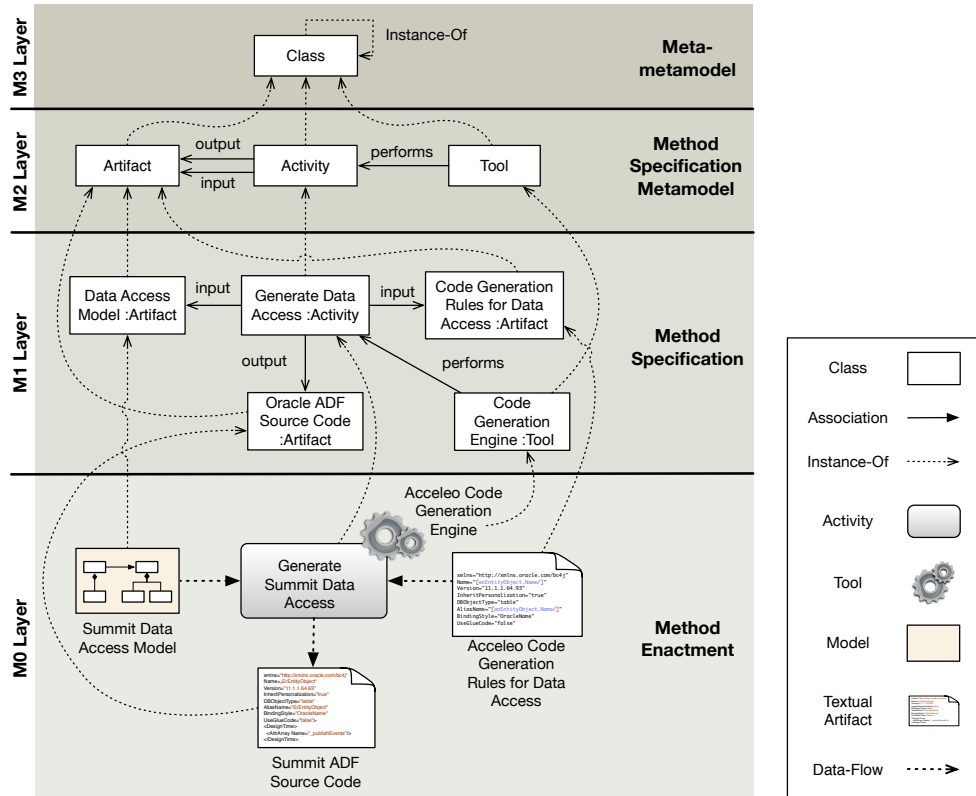


Figure 2.8 Metamodeling layers involved when developing and enacting a method

In the example, a code generation activity is specified that is performed by a code generator. The activity consumes a model and code generation rules, and outputs textual source code.

On the *M0 Layer*, the method is enacted as specified. Only then, specified activities are actually performed and related artifacts are consumed and generated. Note that the activity in the example consumes a model of the software system. In an MDE context that model would usually reside on the *M1 Layer* (cf. Figure 2.2). However, when applying a method engineering-based view on the metamodeling layers, the model is shifted to the *M0 Layer*. One solution to this shifting problem is to distinguish different metamodeling spaces [AK01]. Intuitively, we need to be aware, whether a method engineering-based or an MDE-based view on the metamodeling layers is provided.

2.3 Software Reengineering

In this section, we introduce the area of *software reengineering*. Intuitively, reengineering is a discipline which is concerned with the examination and alteration of a software system to reconstitute it in a new form [CC90]. Subsequently, we first describe two specific manifestations

of reengineering, namely software migration and software modernization and discuss the role of transformation methods in this context. While the relationships between those terms has already been discussed in Chapter 1, we provide a brief recap to define essential notations of this thesis. Thereafter, we introduce various foundations in the area of software reengineering. In particular, we describe the functioning of a compiler, discuss different programming paradigms, and introduce a technique called *concept modeling* that can be used represent the functionality of a software system.

2.3.1 Software Migration, Transformation & Modernization

If a software system does not fulfill all of its requirements but is still valuable for ongoing business, it has become a legacy system. In this case, *software migration* is an established way to deal with such systems by transferring them it into a new environment while retaining its data and functionality [Bis+99].

By this definition, a migration is associated with two essential characteristics: On the one hand, a migration always comprises an *environmental change*. This change can be diverse, e.g., the runtime environment, the software architecture and/or the programming language can be changed [GW05]. On the other hand, the functionality and data of the original system needs to be *preserved*. However, the degree of preservation can vary. For example, the migrated system can preserve the functionality on a technical level in terms of provided programming interfaces. Or, it can preserve the functionality on a higher level of abstraction by enabling to perform the same business tasks, while the technical realization was changed notably. As software migration is a central term of this thesis, we define it as follows:

Notation 4 (Software Migration)

Software migration is a kind of software reengineering concerned with the transition of a legacy system to a new environment while retaining the system's data and functionality [Bis+99].

The primary concern when performing a software migration is to preserve the system's functionality. One simple way to achieve this is to follow a wrapping strategy, e.g., to encapsulate the legacy system behind a layer in the new environment. However, just ensuring that the functionality is preserved is often not sufficient in practice as it might result in a system with decreased non-functional properties. For example, adding an additional layer can increase the complexity of the system and thereby reduce its maintainability or performance [BR15, p.27].

We use the term software modernization to emphasize the additional attempt of adapting the software system to the new environment. As stated in [Fle+07], we do not intend to just create an executable version of the system in the new environment but design the system for that

environment. Ideally, we can additionally preserve the original functionality. However, there might be instances in which an adaptation of the system requires changing existing functionality. In the case of a software modernization scenario, we assume that this is tolerable. In this thesis, we define software modernization as follows:

Notation 5 (*Software Modernization*)

Software modernization is a kind of software migration concerned with the transition of a legacy system to a new environment while adapting the system to the new environment.

Like the development of software systems, modernizing legacy systems by transferring them into a new environment is a complex endeavor. Therefore, it is common practice to establish a modernization project during which a *software modernization method* gets enacted. A modernization method is a method (cf. Section 2.2) that guides the complete endeavor of a software modernization. We define a software modernization method as follows:

Notation 6 (*Software Modernization Method*)

A software modernization method is a method that is used to guide a software modernization endeavor.

An example for a software modernization method is the Reference Migration Process (ReMiP) [SWH10, pp.85-131]. ReMiP can be seen as a reference method as it has been derived by generalizing established ones. Thereby, core disciplines of a software modernization endeavor were identified and described (cf. Figure 1.3).

In this thesis, we focus on the *Transformation* discipline. The method parts belonging to this core discipline form a *software transformation method* which specifies how to perform the transition of a legacy system on a technical level. Transformation methods are reengineering methods as they describe how to perform a (technical) alteration of a software system. Therefore, they are instances of the well-established horseshoe model (cf. Figure 1.4, page 7) which consists of the consecutive phases reverse engineering, restructuring and forward engineering [KWC98]. We define a transformation method as follows:

Notation 7 (*Software Transformation Method*)

A software transformation method is an instance of the horseshoe model and used to guide the technical transition of a legacy system into a new environment during a software modernization endeavor.

2.3.2 Compiler

A compiler is a program that translates a program, written in a source language, to a program, written in a target language [Aho+06, p.1]. A common use case is to translate a program written in a programming language on a higher level of abstraction, e.g., an object-oriented language, to a lower level of abstraction, e.g. to machine code. However, when transferring this definition to the domain of software modernization, one can interpret an automated transformation of a legacy system as some kind of compilation.

Usually, compilers possess a certain structure which can be described in terms of phases, activities and artifacts. This structure is shown in Figure 2.9.

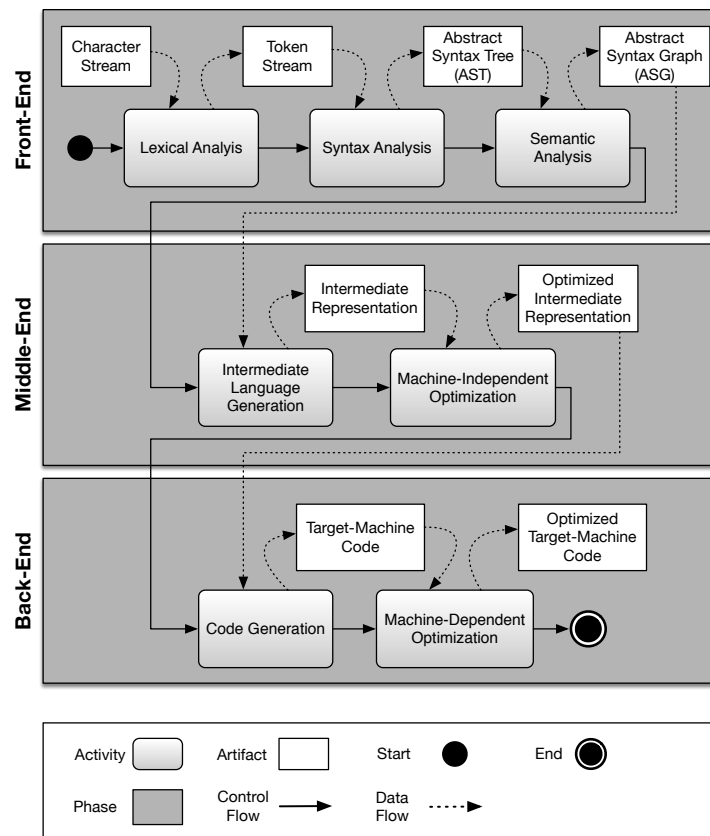


Figure 2.9 Structure of a compiler

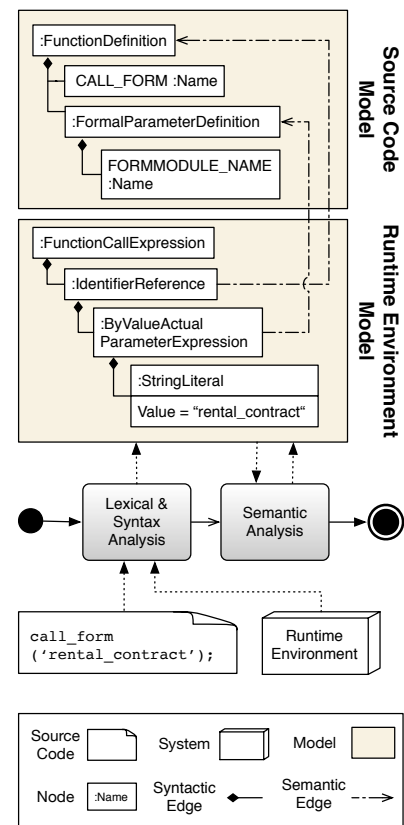


Figure 2.10 Source code and its abstract syntax graph

A compiler can consist of up to three phases, namely the *Front-End*, *Middle-End* and *Back-End* [FCL09, p.395]. Each phase serves a dedicated purpose and contains various activities [Aho+06, pp.4-11]. The front-end is responsible for analyzing the structure of the source program. First, a *Lexical Analysis* decomposes the textual source code that represents the program into so-called *tokens*, e.g., keywords defined by the programming language, numbers

or operators. Based on this *Token Stream*, the *Abstract Syntax Tree (AST)* is constructed during the *Syntax Analysis* activity by a parser. The *Semantic Analysis* turns the AST into an *Abstract Syntax Graph (ASG)*.

The ASG serves as input for the (optional) *middle-end*. This phase is responsible for transforming the source language into a so-called intermediate language (*Intermediate Language Generation*) which is optimized subsequently (*Machine-Independent Optimization*). The use of an intermediate language is motivated by the fact that it enables using different front and back-ends, i.e., it enables supporting different programming languages while reusing the middle-end. Possible optimizations encompass the elimination of redundancy or dead code.

The last phase of a compiler is called the back-end. It is responsible for generating the target program. First, *Code Generation* is performed to transform the intermediate representation into source code of the target environment. Thereafter, environment-specific optimizations are performed on the resulting code.

As Abstract Syntax Trees (ASTs) and Abstract Syntax Graphs (ASGs) represent data structures that are often used by examples in this thesis, we describe them in more detail.

Abstract Syntax Trees and Graphs

An AST results from performing a *Lexical & Syntax Analysis* on source code. The purpose of these activities is twofold [Aho+06, pp.5-8]. On the one hand, the syntactical consistency with the language definition is verified, i.e., it is ensured that the source code conforms to the grammar of the programming language. On the other hand, the information gathered during this activity is persisted in the form of an AST. An abstract syntax tree is a tree-based data structure to represent source code which consists of nodes and edges. The nodes represent syntactic entities, i.e., programming concepts (cf. Section 2.3.4), of the underlying programming language. Intuitively, an AST only preserves the logical information of the source code [KNE92]. In particular, it does not include additional information that is used to increase the readability of the code or assist parsing, like brackets, keywords or indentations.

An example of source code and its corresponding abstract syntax tree is shown in Figure 2.10. The source code consists of a single statement in which a function named `call_form` of the runtime environment is invoked. Thereby, the value `rental_contract` is passed as an argument. The lower AST represents the source code, while the upper AST represents the runtime environment, i.e., the function that gets invoked. As can be seen, the essential logical information represented by the source code and the runtime environment is captured by the ASTs. However, brackets or keywords are lost. Note that the dashed edges do not belong to the AST, as an AST only consists of nodes and edges that form a tree. Edges belonging to the AST are also called *syntactic edges* [KGW98].

The dashed edges result from performing a *Semantic Analysis*. The purpose of this activity is twofold [Aho+06, p.8]. On the one hand, the semantical consistency with the language definition is verified, e.g., it is checked that each variable used has also been defined. On the other hand, the information gathered during this activity is persisted in the form of an ASG. An ASG is an AST which is extended by semantic edges [KGW98], making it a (cyclic) graph. These edges hold additional information related to the syntactic elements used. In the example, the edges associate the invocation of the runtime function with their corresponding definition.

2.3.3 Programming Paradigms

A programming paradigm refers to a way of programming a computer based on a set of principles [VR09]. Examples for paradigms are *object-orientation*, *concurrent programming* or *functional programming*. The reason for the variety of paradigms lies in the fact that some paradigms are better suited for solving particular computational problems than others. Thereby, a programming language can realize one or multiple programming paradigms, e.g., Java is an object-oriented language that enables concurrent programming.

The examples used in this thesis often discuss the challenges that arise when changing the programming paradigms during a software modernization. In particular, we focus on two paradigms, namely *imperative* and *declarative programming*. Intuitively, in imperative programming, a programmer specifies *how* to perform a computational task by specifying single actions, i.e., by describing what to do. In contrast, in declarative programming, a programmer specifies *what* shall happen without describing how. To give an idea for the difference between these two paradigms, we give an example by Figures 2.11 and 2.12. The example demonstrates in which way the same functionality can be realized by using the different paradigms.

The example shows the realization of *attribute validation rules*. Thereby, an attribute refers to a field within a data set whose value shall be validated. In the example, the value of the fields `date_shipped` and `date_ordered` are compared to validate that a good was not shipped before it has been ordered.

In Figure 2.11, the functionality is realized imperatively in Oracle Forms, more specifically, by a code block written in the programming language PL/SQL. The block gets executed by the runtime environment whenever the corresponding data set shall be validated. The validation is realized by describing single actions, i.e., by specifying how to perform the validation. First, the condition of the IF-statement checks whether `date_shipped` is smaller than `date_ordered`, i.e., whether a product is shipped before it has been ordered. If this is the case, a message is shown and an error is raised.

In Figure 2.12, the same functionality is realized declaratively in Oracle ADF. In this environment a language is provided that enables specifying validation rules. Due to that, the

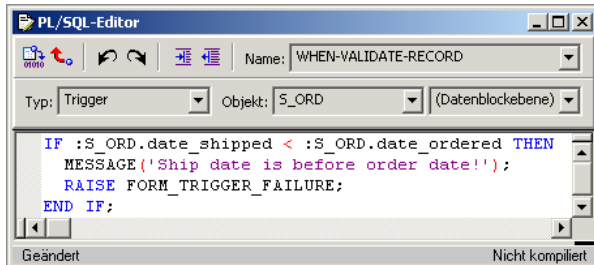


Figure 2.11 Imperative realization of attribute validation rules in Oracle Forms

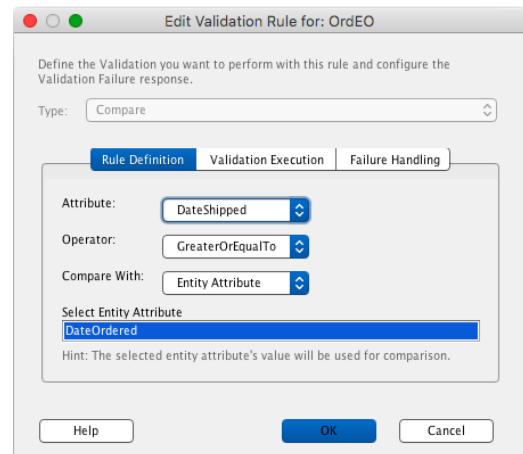


Figure 2.12 Declarative realization of attribute validation rules in Oracle ADF

validation can be realized by describing the condition that shall be ensured and the message that shall be displayed in case of an error, without describing the actions to perform. Here, the underlying platform is responsible for performing the required actions to validate the condition. In the editor shown, it is specified that the attribute `DateShipped` shall be `GreaterOrEqualTo` to the attribute `DateOrdered`. Although not shown, the message that is shown in case the validation fails can be specified in the register called *Failure Handling*.

2.3.4 Concept Modeling

Concept modeling is a technique that was introduced in [KNE92] to represent a software system by a set of concepts. Thereby, each concept belongs to a particular level of abstraction and refers to a specific part of the software system's source code. The general idea of concept modeling is illustrated in the left side of Figure 2.13. A concrete example is shown in the right side of the figure, based on the source code depicted in Figure 2.10 on page 29.

The *M1 Layer* is shown in the upper part of Figure 2.13, on which concepts are modeled (cf. Section 2.1.1). On the *M0 Layer* below, instances of the concepts within the actual software system can be identified. Essentially, we distinguish between two classes of concepts, namely *language concepts* and *abstract concepts*.

On the lowest level of abstraction, language concepts reside. A language concept represents a syntactic entity defined by a corresponding programming language. Therefore, instances of language concepts are nodes within an AST (cf. Section 2.3.2) and can be automatically

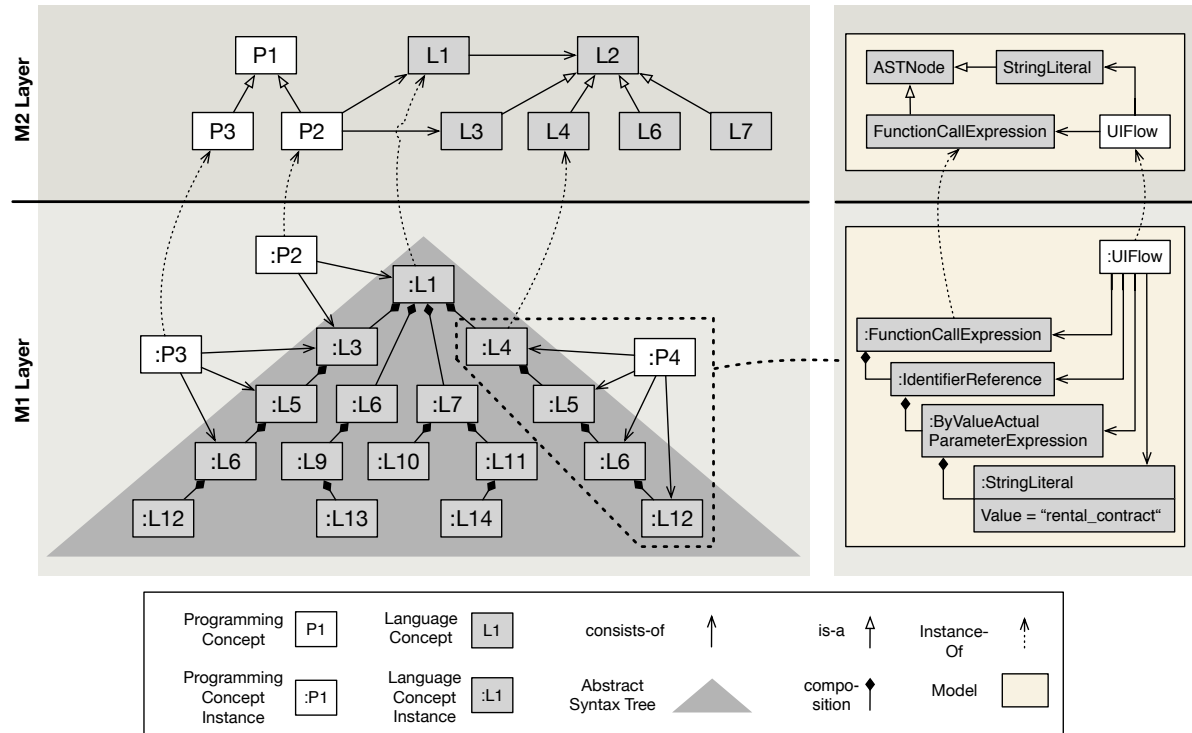


Figure 2.13 Representing a software system as a set of concepts (left), example (right)

identified by a parser. Examples for language concepts are `FunctionCallExpression` or `StringLiteral`. In this thesis, we define them as follows:

Notation 8 (Language Concept)

A language concept is a syntactic entity of a programming language.

Abstract concepts reside on higher levels of abstraction. They represent a general idea of a computation or problem solving principle [KNE92]. Thereby, they are not associated to a specific programming language but represent language-independent principles. In this thesis, we define an abstract concept as follows:

Notation 9 (Abstract Concept)

An abstract concept represents a language-independent idea of a computation or problem solving principle.

Abstract concepts can be further classified into programming concepts and architectural concepts. A programming concept represents general programming strategies, data structures or algorithms [KNE92]. An example for a programming concept is the `UIFlow` which represents the navigation between user interfaces. While the realization of such navigation flows depends

on the programming language used (e.g., by using a `FunctionCallExpression`), it is a language-independent principle. In this thesis, we define a programming concept as follows:

Notation 10 (*Programming Concept*)

A programming concept is an abstract concept and represents general programming strategies, data structures or algorithms.

An architectural concept represents components or interfaces that reside within a software system [KNE92]. In contrast to programming concepts, they do not represent some functionality of the system but focus on describing its overall structure. In this thesis, we define an architectural concept as follows:

Notation 11 (*Architectural Concept*)

An architectural concept is an abstract concept and represents components or interfaces.

Concepts can be related to each other by relations. In this thesis, we focus on two types of relations, namely *is-a* and *consists-of* relations. The *is-a* relation can be used to express a hierarchy between concepts. Intuitively, if concept L3 *is-a* concept L2, then L3 is a sub-concept of L2. For example, a `FunctionCallExpression` is a sub-concept of `ASTNode`.

The *consists-of* relation can be used to express dependencies between concepts. Intuitively, if the identification of a concept requires identifying others first, then that concept consists of the others. Specifying such dependencies is essential to (automatically) identify an instance of a concept within a software system, but not sufficient. In addition, it is required to specify constraints between dependent concepts, e.g., in terms of data or control flow relations. For example, a `UIFlow` *consists-of* a `FunctionCallExpression`, an `IdentifierReference`, a `ByValueActualParameterExpression` and a `StringLiteral`, as illustrated in the right side of Figure 2.13. The `StringLiteral` encodes the target of the navigation flow and needs to be a descendant of the `FunctionCallExpression` within the AST.

CHAPTER 3

Scenario and Related Work

In this chapter, we give an overview of the related work of this thesis. For this purpose, we first describe a real-world modernization scenario in Section 3.1 that resulted in the problem statement addressed by this thesis. Based on this scenario, we derive a set of requirements which a solution concept need to fulfill in Section 3.2. We identify and classify related work in Section 3.3 and evaluate it against the requirements. The findings of this chapter are summarized in Section 3.4.

3.1 Modernization Scenario

The problem statement of this thesis (cf. Section 1.2) originated from a real-world modernization scenario that we observed in an industrial context. In this context, the problem of transforming legacy systems that were developed based on the platform of Oracle Forms was addressed. While Oracle Forms had a large and active installation base, various companies were unsatisfied with the capabilities of the platform. For example, Oracle Forms did not enable to optimize applications for mobile devices. Since, this became an important requirement over time, Forms-based systems were considered to be legacy. One solution to this problem was to transform those systems to the more recent platform called Oracle ADF.

While Oracle itself recognized the desire of customers to transform Forms-based systems to ADF, it did not provide a tool-supported transformation method to guide this endeavor [Ora12]. Since there was no transformation method available, customers that wanted to modernize their systems needed to develop a method for their situation at hand. However, due to missing knowledge of how to develop a situation-specific transformation method, we observed that companies started redeveloping their systems from scratch, instead.

The solution concept provided by this thesis shall address this problem, i.e., guide the development of situation-specific transformation methods in order to transform Forms-based systems to ADF. However, we observed that companies which used Oracle Forms were often facing additional environmental changes. For example, another platform called Oracle Reports was regularly used in combination with Oracle Forms to automatically generate reports. Since it was desired to transform systems based on this platform, too, the solution concept needs to be generic. Nevertheless, we use the transformation from Forms to ADF as a running example of this thesis. For this reason, we introduce both platforms in more detail subsequently, whereby we focus on their software architecture.

3.1.1 Oracle Forms

Oracle Forms is a platform to develop enterprise applications that consist of a set of *dialogs*. A dialog represents a user interface that enables a user to interact with an underlying data source. In this way, it supports the user in performing an underlying business tasks. An example for such a dialog can be seen in the left side of Figure 1.2 on page 5. Those dialogs are defined declaratively by a proprietary Fourth-Generation Programming Language (4GL) as well as by using imperative PL/SQL source code. The software architecture of an Oracle Forms-based system can be seen in the left side of Figure 3.1.

The architecture of Forms-based systems can be described by *tiers* and *layers* [Fow02, p.19] as well as *components*. Tiers describes a physical separation, i.e., a tier represents a distinct device on which components can be executed. In contrast, layers describe a logical separation, i.e., a layer represents a distinct concern that is addressed by contained components. Taken together, these components realize the functionality of the legacy system.

Usually three tiers are used by Oracle Forms-based systems, namely a *Client Tier*, a *Middle Tier* and a *Database Tier*. On the client tier, i.e., the end user's device, the *Presentation Layer* is located. The corresponding component is provided by the platform and responsible for rendering the user interface of the system and processing user interaction. It behaves like a thin client as processing is mostly limited to forwarding the interaction events to the middle tier.

The actual source files that form the Forms-based system are executed on the middle tier, i.e., on an application server. While various components are located on this tier, they cannot be classified into distinct concerns, i.e., layers. For example, the *PL/SQL engine* component executes imperative source code blocks. However, the source code can be used for different purposes, e.g., to validate data, to react on system events or to adapt the user interface. Therefore, the middle tier can be seen as *monolithic*, i.e., it only consists of a single layer.

¹Based on http://de.slideshare.net/oracle_imc_team/oracle-forms-modernization-strategies (accessed March 22th, 2016)

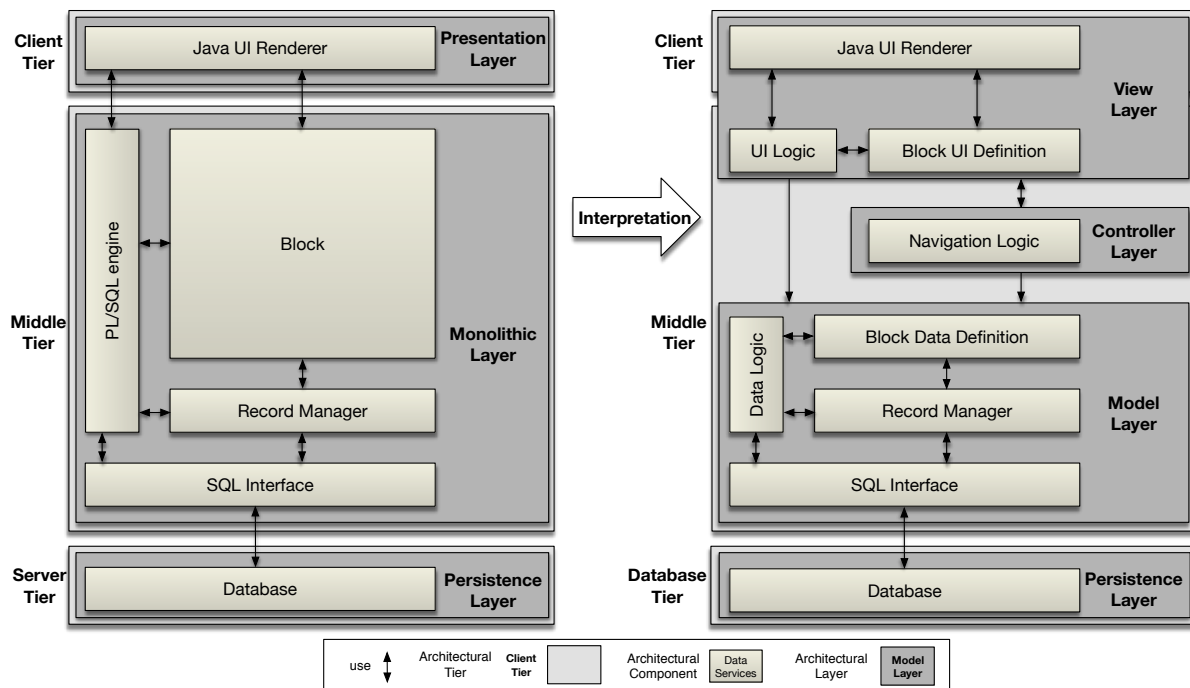


Figure 3.1 Architecture of Oracle Forms-based systems (left), interpreted as Model-View-Controller (MVC) architecture (right)¹

On the database tier, i.e., the backend server, the *Persistence Layer* is located. On this layer a *Database* is executed that enables storing and retrieving the persistent data of the application. Besides data sets, executable programs can also be defined here in terms of *stored procedures*².

Related to the modernization scenario considered, we assume that we do not change the content of the database tier, i.e., the database. In addition, we neglect the thin component of the client tier as it just renders the user interface. Instead, our focus lies on transforming the content of the middle tier, i.e., the actual legacy system. One of the challenges of the transformation lies in the fact that systems in the target environment Oracle ADF possess a Model-View-Controller (MVC) architecture. The result of interpreting the architecture of Forms-based systems as an MVC architecture can be seen in the right side of Figure 3.1.

In an MVC architecture, components are separated into three layers [Bus+96, pp.125-143]. The *Model* layer contains those components that provide core data and associated functionality. Components of the *View* layer display information to a user while components of the *Controller* layer process user interaction. Aligning the monolithic architecture of Oracle Forms with the MVC architecture requires breaking up existing components. For example, the *PL/SQL engine* executes source code for different concerns. These concerns need to be separated, i.e., *UI Logic*, *Navigation Logic* and *Data Logic* needs to be distinguished.

²https://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm (accessed March 22th, 2016)

3.1.2 Oracle ADF

Oracle ADF is a platform to develop enterprise applications based on Java EE standards. It provides a set of infrastructure services to ease the development and is not restricted to a specific technology. Rather, a choice between different technologies exists on various layers. The software architecture of an Oracle ADF-based system can be seen in Figure 3.2.

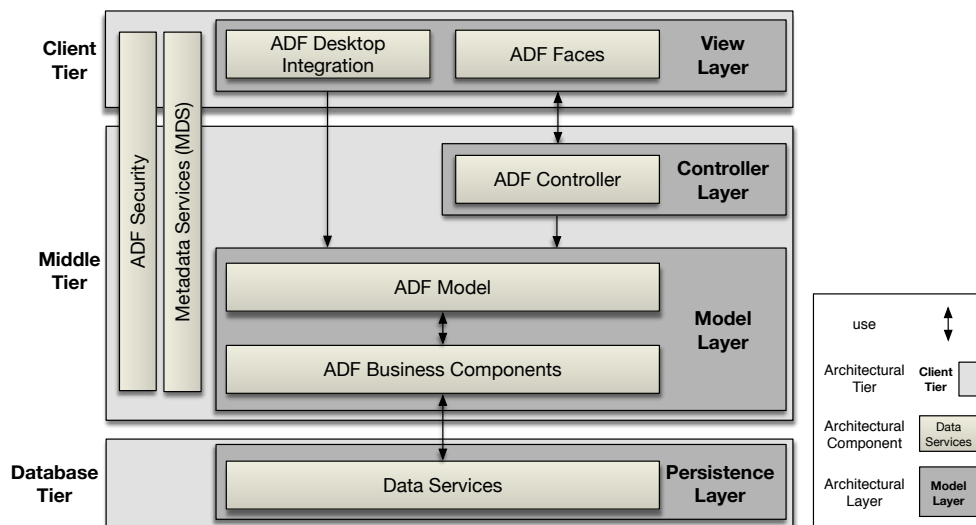


Figure 3.2 Model-View-Controller (MVC) architecture of Oracle ADF-based systems³

As can be seen, ADF-based systems usually use the same three tiers as Forms-based systems. However, in terms of layers, Oracle ADF follows an MVC architecture by design. The components shown are just examples and can be exchanged as desired. For example, the *ADF Business Components* represent one way to connect to external data sources and to provide access to these data sources within the model layer. An alternative would be to use a self-developed, Java-based database connection instead. Note that some cross-cutting components exist, i.e., components that are present on all layers. They provide cross-cutting functionality, i.e., security or multi-language support.

This concludes the description of the modernization scenario, the platforms of Forms and ADF as well as their differences on the architectural level. Subsequently, we use the modernization scenario to derive a set of requirements for the solution concept of this thesis.

³Based on <http://docs.oracle.com/middleware/1221/adf/concepts/GUID-422ED063-2643-4F8D-B4BB-A8FA4C8CF536.htm> (accessed March 22th, 2016)

3.2 Requirements

In this section, we discuss challenges related to the definition of a transformation method that arise due to the modernization scenario described in the previous section. Based on these challenges, we derive a set of requirements that any method engineering approach needs to fulfill in order to support the definition of transformation methods in the context of the modernization scenario. We classify the requirements into four categories, namely *Controlled Flexibility*, *Adaptability*, *Tooling* as well as *Formalization*.

Controlled Flexibility

The fact that no transformation method was available in the first place was the main problem of the modernization scenario described. In addition, no method engineering approach could be used to develop a method for the situation at hand. The reason for this is that existing approaches provided a low degree of *flexibility* [HBO94]. Intuitively, this essential characteristic of a method engineering approach determines the degree of freedom given during the development of a method to adapt the method to a prevailing situation. To address this problem, Requirement 1 claims that the solution concept shall provide a high degree of flexibility.

Requirement 1 (*Flexibility*)

<i>The method engineering approach shall provide a high degree of flexibility.</i>
--

Nevertheless, some method engineering approaches provided a high degree of flexibility. As an edge case, consider the development of a transformation method from scratch which provides the highest degree of flexibility possible. However, such approaches were not applicable in the modernization scenario, as they provided a low degree of *control* [HBO94]. Intuitively, this essential characteristic of a method engineering approach determines the degree of guidance given during the development of the method. Without such guidance, it is not possible to ensure the result of the development, e.g., the correctness or quality of the method. To address this problem, Requirement 2 claims that the solution concept shall provide a high degree of control.

Requirement 2 (*Control*)

<i>The method engineering approach shall provide a high degree of control.</i>
--

Adaptability

Method engineering approaches can support specific environmental changes. For example, related to the modernization scenario, an obvious solution would be to define a method

engineering approach that supports the transformation of Oracle Forms-based legacy systems to Oracle ADF. However, in this case, the same method engineering approach would not be applicable for other environmental changes, i.e., the transformation of Oracle Reports-based systems. As the modernization scenario explicitly foresees the applicability of the solution concept to different environmental changes, Requirement 3 claims that the solution concept shall be generic, i.e., it shall not be limited to a specific environmental change.

Requirement 3 (*Generality*)

The development of transformation methods shall not be limited to a specific environmental change.

Methods are specified to guide an endeavor performed by people. Among other things, transformation methods are specified to guide the development of tools and reimplementation activities. The granularity of a method, e.g., the scope of an activity needs to be chosen in a way that sufficient guidance is provided. For example, related to the transformation from Forms to ADF, activities could be aligned with the layers of the MVC architecture. In this case, an example for an activity could be to *Reimplement the View Layer*. However, it would also be possible to choose a smaller scope, e.g. by specifying to *Reimplement the UI Logic*. As we aim for an expert to decide which level of granularity fits best to a given situation, Requirement 4 claims that the solution concept shall not be limited to a specific level of granularity.

Requirement 4 (*Granularity*)

The development of transformation methods shall not be limited to a specific level of granularity.

Different established software transformation strategies exist to transform a legacy system into a new environment, i.e., wrapping, reimplementation or conversion [SWH10, pp.10-13]. We assume that situation-specific transformation methods often require the joint use of different strategies. For example, related to the transformation from Forms to ADF, it could be desired to *convert the Model Layer* while the *Controller Layer* gets *reimplemented*. To enable the joint use of strategies, Requirement 5 claims that the solution concept shall not be limited to a specific transformation strategy.

Requirement 5 (*Versatility*)

The development of transformation methods shall not be limited to a specific transformation strategy.

Tooling

Transformation methods are used to systematically transfer legacy systems into new environments. Automating at least parts of the transformation is essential in order to reduce the overall effort and to avoid errors that could result due to manual interactions [SWH10, p.131]. Related to the modernization scenario, we assume that automating the transformation from Forms to ADF, i.e., providing tool-support, is essential for an acceptance of a method in practice. In other words, we assume that companies will otherwise favor a redevelopment instead. To enable tool-support during the transformation, Requirement 6 claims that the use of tools during the enactment of a transformation method shall be guided, i.e., considered during its development.

Requirement 6 (*Continuity*)

The use of tools during the enactment of a transformation method shall be guided.

Formalization

A method can be specified formally, for example by using the Software and Systems Process Engineering Metamodel (SPEM) (cf. Section 2.3.1). Related to the modernization scenario, we assume that a formal specification of transformation methods is an essential prerequisite to reuse (parts of) a developed method within a subsequent modernization project. The formal specification enables reengineering a developed method, i.e., to extract relevant parts. Due to that, Requirement 7 claims that transformation methods that get developed with the solution concept shall be specified formally.

Requirement 7 (*Formalization*)

Developed transformation methods shall be specified formally.

3.3 Related Work

In this section, we give an overview of the related work of this thesis, i.e., we introduce existing approaches that could be applied in the modernization scenario described in Section 3.1 to develop a transformation method. Thereby, the related work falls into two categories: *Situational Method Engineering (SME) approaches* and *reengineering frameworks*. We introduce both categories and corresponding approaches subsequently.

3.3.1 Situational Method Engineering Approaches

Situational method engineering approaches can be used to systematically develop or adapt methods for a situation at hand (cf. Section 2.2.1). Therefore, such an approach could have been

applied in the modernization scenario introduced in Section 3.1 to derive a situation-specific transformation method. However, as discussed in the previous section (cf. Requirements 1 and 2), the practical applicability of an SME approach is mainly determined by the degree of *controlled flexibility* it provides. While it is difficult to precisely quantify the degree of control or flexibility that a specific approach provides, a coarse granular classification is described in [HBO94]. In Figure 3.3, we applied the classification on state-of-the-art situational method engineering approaches that support the development of transformation methods [GFB15]. We describe the classes subsequently.

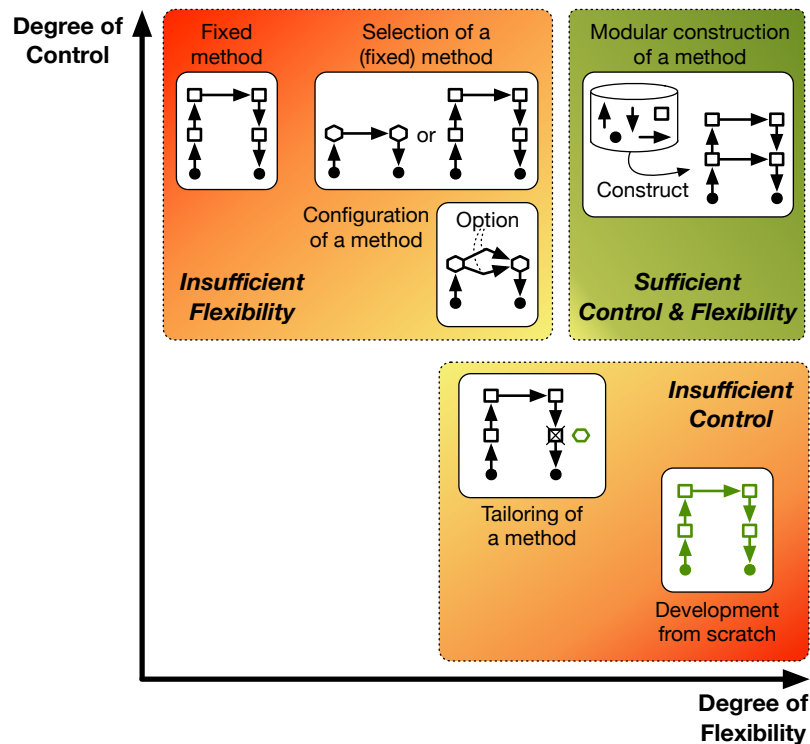


Figure 3.3 Categorization of situational method engineering approaches to develop transformation methods according to their degree of controlled flexibility, based on [HBO94]

Fixed Method The provisioning of a *fixed* method can be seen as an edge case as situation-specific adaptation is not foreseen. Instead, a fixed set of activities to enact, artifacts to create, tools to use and roles to apply is described in order to transform a legacy system. An approach belonging to this category provides a low degree of flexibility as the method needs to be applied as is. At the same time, one can argue that the use of a fixed method provides a high degree of control. Since no adaptation is foreseen, the result is always predetermined.

Examples for existing approaches of this category are *Sensoria* [MH11], *Guizmo* [SSG14] or *WA-to-RIA* [RE+12]. *Sensoria* is a transformation method to transform legacy systems into

a service-oriented environment. In particular, the method guides the restructuring of existing source code so that layers and services are distinguished afterwards.

Selection of a Method A method engineering approach can provide a set of (fixed) methods to *select* the most suitable one for the situation at hand. This can be seen as a way to perform situation-specific adaptation. An approach belonging to this category provides a higher degree of flexibility than a fixed method as it enables selecting the method as a whole. Nevertheless, the flexibility is only based on the selection itself and therefore quite limited. Especially, since an adaptation of the resulting method is usually not foreseen. However, this results in a high degree of control.

An example for an existing approach of this category is the *SOA migration framework (SOA-MF)* [RL15]. The defined method engineering process enables selecting a transformation method to transform a legacy system into a service-oriented environment. The selection of a situation-specific method is based on identifying the knowledge that is available inside a company or that shall be generated.

Configuration of a Method A method engineering approach can enable to *configure* a provided method for the situation at hand. In this case, the provided method needs to have some variation points. An approach belonging to this category provides a higher degree of flexibility than a fixed method as it enables exchanging parts of the method. Nevertheless, as the flexibility is only related to foreseen variation points, it is quite limited. However, as the variation points are predefined, such an approach is associated with a high degree of control.

An example for an existing approach of this category is the *ARTIST framework* [Men+14]. The framework enables developing transformation methods to transform legacy systems into a cloud environment. The defined method engineering process foresees the use of provided tools to assess the situation, e.g., by automatically analyzing metrics of the legacy system. Based on the results, the method gets configured.

Tailoring of a Method A method engineering approach can enable to *tailor* a provided method for the situation at hand. In this case, the provided method needs to be specified formally to enable a systematic manipulation. An approach belonging to this category provides a high degree of flexibility as, in principle, the provided method can be changed arbitrarily. However, usually such approaches provide an insufficient degree of control, as the changes itself are not guided by a corresponding method engineering process.

Examples for existing approaches of this category are the *Reference Migration Process (ReMiP)* [SWH10, pp.85-131], *REMICS* [Bar+10], *SOAMIG* [Fuh+12] and *ARTIST* [Men+14]. *ReMiP* is a comprehensive and generic modernization method that contains a transformation method (cf. Figure 1.3, page 6). Technically, the method is specified by customizing the

Rational Unified Process (RUP), which in turn is specified in (a predecessor of) SPEM [Kru03, p.51]. Therefore, existing tools to tailor RUP or SPEM-based methods can be used to tailor the *ReMiP* as well. However, no guidance is given on how to perform the tailoring.

Modular Construction of a Method A method engineering approach can enable the modular *construction* of a method for the situation at hand. In this case, a set of predefined building blocks for methods needs to be provided as well as a method engineering process to guide the construction. An approach belonging to this category provides more flexibility than the use, selection or configuration of a method. This is due to the fact that the building blocks usually enable a wide range of possible combinations to develop a method. On the other hand, such an approach provides more control than the new development or tailoring of a method. This is due to the fact that a method engineering process to construct a method essentially needs to be provided. As a result, the situation-specific adaptation is usually foreseen.

An example of an approach that goes into this direction is the *ServiciFi* method [Kha+11] which can be used to transfer legacy systems into a service-oriented environment. The method itself is constructed by assembling building blocks of SOA-transformation methods. However, the method engineering process to construct the method is not generalized, i.e., it is only applied once to develop the *ServiciFi* method.

To the best of our knowledge, no approach that enables the modular construction of transformation methods had been defined, yet. Therefore, current approaches lack a sufficient degree of controlled flexibility. We aim to address this problem by the solution concept provided in this thesis.

3.3.2 Reengineering Frameworks

Reengineering frameworks can be used to realize reengineering tools, i.e., tools that can be executed to (semi-)automatically reengineer a software system (cf. Section 2.3). As transformation methods are some kind of reengineering methods, a reengineering framework could have been applied in the modernization scenario as well. Such a framework could have been used to develop a reengineering tool that transforms the legacy system into the new environment. Thereby, reengineering frameworks can be distinguished based on the reengineering phases they cover as well as the degree of controlled flexibility they provide.

Examples for frameworks that only address specific software reengineering phases are *GUPRO* [Ebe+02] or *Bauhaus* [RVP06]. *GUPRO* is an integrated and generic framework to realize reverse engineering tools. The corresponding method consists of the consecutive activities of *Extract-Abstract-View*: Information about the legacy system is extracted from the

source code and stored in a central repository. Thereafter, the information can be processed by abstracting and visualizing it.

Examples for frameworks that cover all reengineering phases but provide a low degree of control are *CORUM* [KWC98] or *MoDisco* [Bru+14]. As part of the *CORUM* framework, the horseshoe model was introduced with the intention to integrate existing tools. This is achieved by formalizing the horseshoe as a schema that includes different abstraction levels and a corresponding process on how to instantiate it. However, the schema is only introduced as a whole. An adaptation of the schema is not guided, i.e., controlled.

The *MoDisco* framework is a reengineering framework based on model-driven engineering (cf. Section 2.1). On the one hand, it defines a generic infrastructure that provides some capabilities like storing, querying or transforming models. On the other hand, it provides a set of technology-dependent tools like parsers, metamodels or a code generator. Based on these capabilities of the framework, it is foreseen to develop custom tools for the situation at hand. However, the development of a situation-specific tool is also not guided, i.e., controlled.

An example for a framework that covers all reengineering phases and aims to provide a high degree of controlled flexibility is the *SENSEI* framework [Jel15]. The general idea of the framework is to provide capabilities to develop tools by integrating a set of predefined reengineering services. To some extent, this is comparable with method engineering approaches that enable the modular construction of transformation methods. However, the identification of a set of generic reengineering services is ongoing research [FRE14]. Therefore, the framework does not provide a predefined set of services, i.e., the building blocks for tools are missing. Instead, services need to be created when applying the framework. Since this endeavor is not guided, the approach lacks control in its current state.

In conclusion, reengineering frameworks are useful to realize tools as they focus on associated technical challenges and provide a high degree of flexibility. However, most fall short in providing guidance on how to systematically construct the tools, i.e., how to develop the underlying method. Therefore, we focus evaluating method engineering approaches against the requirements described in Section 3.2.

3.3.3 Evaluation

In this section, we use the requirements defined in Section 3.2 to evaluate the method engineering approaches introduced in Section 3.3.1. Thereby, we choose one approach for each class introduced. In addition, we briefly describe how the solution concept of this thesis addresses each requirement. A more detailed description can be found in Section 8.2. A summary of the evaluation can be seen in Table 3.1.

			<i>Fixed</i>	<i>Configuration</i>	<i>Selection</i>	<i>Tailoring</i>	<i>Construction</i>
			Sensoria [MH11]	ARTIST [Men+14]	SOA-MF [RL15]	ReMiP [SWH10]	MEFiSTo
<i>Controlled Flexibility</i>	RQ1	The method engineering approach shall provide a high degree of flexibility	✗	○	○	✓	✓
	RQ2	The method engineering approach shall provide a high degree of control	✓	✓	✓	✗	✓
<i>Adaptability</i>	RQ3	The development of transformation methods shall not be limited to a specific environmental change	✗	✗	✗	✓	✓
	RQ4	The development of transformation methods shall not be limited to a specific level of granularity	✗	✓	○	✓	✓
	RQ5	The development of transformation methods shall not be limited to a specific transformation strategy	✗	✓	○	✓	✓
<i>Tooling</i>	RQ6	The use of tools during the enactment of the transformation method shall be guided	✓	✓	○	○	✓
<i>Formalization</i>	RQ7	Developed transformation methods shall be specified formally	✗	✓	○	✓	✓

Table 3.1 Evaluation of selected method engineering approaches against requirements

Requirement 1 & 2 We already discussed in detail in Section 3.3.1 that none of the approaches provides a sufficient degree of *Flexibility* and *Control* at the same time. Fixed methods like *Sensoria* lack flexibility but provide control, while the opposite is true for tailoring-based approaches (cf. Figure 3.3).

The solution concept of this thesis, i.e., the Method Engineering Framework for Situation-Specific Software Transformation Methods (MEFiSTo) (cf. Section 4.1), fulfills both requirements as it is a construction-based approach. Transformation methods are developed in a modular way by assembling predefined building blocks. The endeavor is guided by a corresponding method engineering process.

Requirement 3 *Generality* can be achieved if the reusable method or method building blocks provided by a method engineering approach are not attuned to a specific environmental change. This is only the case for the *ReMiP* approach as it defines a generic transformation method. All

other approaches focus on a specific environmental change, like the transformation to a SOA (*Sensoria*, *SOA-MF*) or to a cloud (*ARTIST*) environment.

In MEFiSTo, this requirement is addressed by the method base, i.e., the repository that contains the reusable building blocks for transformation methods. The content of this repository was designed to be independent of a specific environmental change or technology (cf. Sections 5.3 and 5.4).

Requirement 4 Varying the *Granularity* of the developed method can be achieved by various means. One way is to explicitly foresee an adaptation of the granularity during the development of a method as part of the method engineering process. Another way is to formalize the method in a language that enables refinement. For example, the granularity of SPEM-based method specifications can be adapted retroactively. *ARTIST* and *ReMiP* follow the latter approach. In contrast, *Sensoria* does not foresee an adaptation of the granularity and, as the method is not specified formally, it is also not supported by the specification language. In *SOA-MF*, whether an adaptation of the granularity is possible depends on which method gets selected.

In MEFiSTo, this requirement is addressed in two ways. On the one hand, the method engineering process explicitly foresees to specify a decomposition of the legacy system to transform by so called concepts (cf. Section 6.3.1), whereby the granularity can be varied. The decomposition has a direct influence on the granularity of the resulting transformation method. On the other hand, methods are formalized by using SPEM which supports refinement.

Requirement 5 *Versatility* can be achieved by enabling to combine multiple transformation strategies within the resulting method. *ARTIST* and *ReMiP* both combine different strategies like conversion and reimplementation. In contrast, *Sensoria* only foresees a conversion of the legacy system. In *SOA-MF*, the strategy again depends on which method gets selected.

In MEFiSTo, this requirement is addressed by the method base, i.e., the repository that contains the reusable building blocks for transformation method. These building blocks can be used to specify transformation methods that follow a conversion and/or reimplementation-based strategy (cf. Section 5.3).

Requirement 6 *Continuity* can be achieved by various means. One way is to explicitly provide a set of tools along with the method engineering approach. These tools should be aligned with the resulting method, so that they are directly applicable. Another way is to guide the development of required tools as part of the approach. *ARTIST* follows the former approach, i.e., it provides a set of tools⁴. *Sensoria* and *ReMiP* follow the latter approach by specifying the capabilities of required tools. However, while *Sensoria* describes the required tools in great detail, e.g., by specifying technologies, metamodels or model transformations, *ReMiP* does

⁴<https://github.com/artist-project/> (accessed March 22th, 2016)

not provide many details. In *SOA-MF*, whether the use of a tool is guided, or not, depends on which method gets selected.

In MEFiSTo, this requirement is addressed in two ways. On the one hand, it is explicitly foreseen that a developed transformation method specifies how to develop required tools (cf. Section 5.4). On the other hand, the development shall be based on an associated, generic tool infrastructure (cf. Section 6.5).

Requirement 7 *Formalization* can be achieved by specifying the resulting method formally. For example, a metamodel like SPEM can be used for this purpose (cf. Section 2.2.2). *ARTIST* and *ReMiP* both use SPEM to formally specify the proposed methods. In contrast, *Sensoria* does not provide a formal description. In *SOA-MF*, whether the resulting method is described formally, or not, depends on which method gets selected.

In MEFiSTo, each developed transformation method is specified formally (cf. Section 5.7). First, the MEFiSTo Intermediate Modeling Language (MIML) is used during the development of the method. When the development of a method is completed, the MIML-based specification gets automatically transformed into a SPEM-based specification.

3.4 Summary

In this thesis, we aim to provide a solution concept for a problem statement that originated from a real-world modernization scenario. In this chapter, we introduced that scenario and derived corresponding requirements. We introduced related work, i.e. we introduced existing approaches that could have been applied in the modernization scenario and evaluated them against the requirements. We concluded that existing approaches have various shortcomings and discussed how they are addressed by the solution concept of this thesis.

First, we introduced the modernization scenario of this thesis in Section 3.1. We described the situation that various companies face in practice, i.e., the desire to transform Oracle Forms-based systems to Oracle ADF. We described both environments on an architectural level and described their differences.

In Section 3.2, we discussed a set of challenges that arise due to the considered modernization scenario. Based on these challenges, we derived a set of requirements that a solution concept needs to fulfill in order to be applicable in the modernization scenario.

In Section 3.3, we introduced related work of this thesis. In particular, we introduced and classified existing situational method engineering approaches and reengineering frameworks that could have been applied in the modernization scenario. We evaluated them against the identified requirements and concluded that various shortcomings exist. In addition, we discussed how the solution concept of this thesis addresses the requirements.

PART II

SOLUTION CONCEPT

*“We cannot solve our problems with the same thinking
we used when we created them.”*

– ALBERT EINSTEIN

CHAPTER 4

Overview

In the previous chapter, we identified that state-of-the-art approaches fail to provide sufficient controlled flexibility in the development of transformation methods. In this chapter, we give an overview of the solution concept which addresses this issue. First, we explain the general idea of the solution concept and give an overview of its main constituents in Section 4.1. In Chapters 5 and 6, these constituents are revisited and discussed in more detail. In Section 4.2, we state a set of evaluation criteria related to the solution concept whose fulfillment we aim to discuss by the feasibility studies described in Chapter 7. In Section 4.3, we introduce a legacy system which acts as a running example throughout the thesis. Finally, the findings of this chapter are summarized in Section 4.4.

4.1 The MEFiSTo Framework

To enable the modular construction of situation-specific software transformation methods, we propose a method engineering framework called MEFiSTo. Compared to state-of-the-art approaches to develop methods, the framework provides a higher degree of controlled flexibility. It consists of two main constituents: a method base that provides building blocks to assemble transformation methods and a corresponding method engineering process that guides the development and enactment of situation-specific methods. Subsequently, we first describe the purpose and content of the method base to explain the general idea of how transformation methods are constructed using the framework. Then, we give an overview of the corresponding method engineering process of the framework. In the end of this section, we discuss the design rationale of the main constituents of MEFiSTo.

4.1.1 Method Base

The MEFiSTo framework is a Situational Method Engineering (SME) framework. SME is an established engineering discipline to develop situation-specific methods (cf. Section 2.2.1) by considering the situational context in which the method will be applied. Related to the domain of software modernization, this context consists of the characteristics of the legacy system, the intended target design and the characteristics of the modernization project.

The distinguishing characteristic between SME approaches is the degree of controlled flexibility they provide. Intuitively, flexibility refers to the degree of freedom to adapt a method to the situation at hand, while control refers to the degree of guidance given for this endeavor. In Section 3.3.1, different classes of SME approaches and their degree of controlled flexibility have been introduced. Related to this classification, the MEFiSTo framework belongs to the class of approaches that enable the *modular construction* of transformation methods. These kinds of approaches provide a high degree of controlled flexibility as methods are developed by assembling reusable building blocks of methods. Such building blocks are stored in a repository called a *method base* [Bri96]. In the case of MEFiSTo, we use two different types of building blocks, namely method fragments and method patterns. These constituents of the method base can be seen in the upper part of Figure 4.1.

In this thesis, a method fragment is defined as an atomic constituent of a method, i.e., a single activity, artifact, role or tool (cf. Section 2.2.1). The fragments proposed in this thesis are constituents of transformation methods. Solely using the proposed method fragments would provide a high degree of flexibility, as transformation methods could be freely assembled from them. However, we aim for controlled flexibility, i.e., we want to make sure that the assembled method possesses desired properties. For example, we want to ensure that a transformation method can actually be used to transform the legacy system on which it is applied. Then, a necessary prerequisite is that a consistent path starting from the source code of the legacy system to the resulting transformed source code needs to be specified. Such methodological knowledge is encoded by the method patterns that are also contained in the method base. Intuitively, the patterns represent transformation strategies by describing constraints over the method fragments, e.g., by defining which one to use and how to order them.

The guidance provided by the method base in the development of transformation methods is restricted to the structure of the method specification to develop. To guide the development itself, we additionally provide a corresponding method engineering process. In MEFiSTo, the development of a method can be seen as being pattern-based. The idea is to first (I) select a method pattern that fits the situational-context observed. The pattern (II) then determines the method fragments to be customized. This is exemplified in the lower part of Figure 4.1.

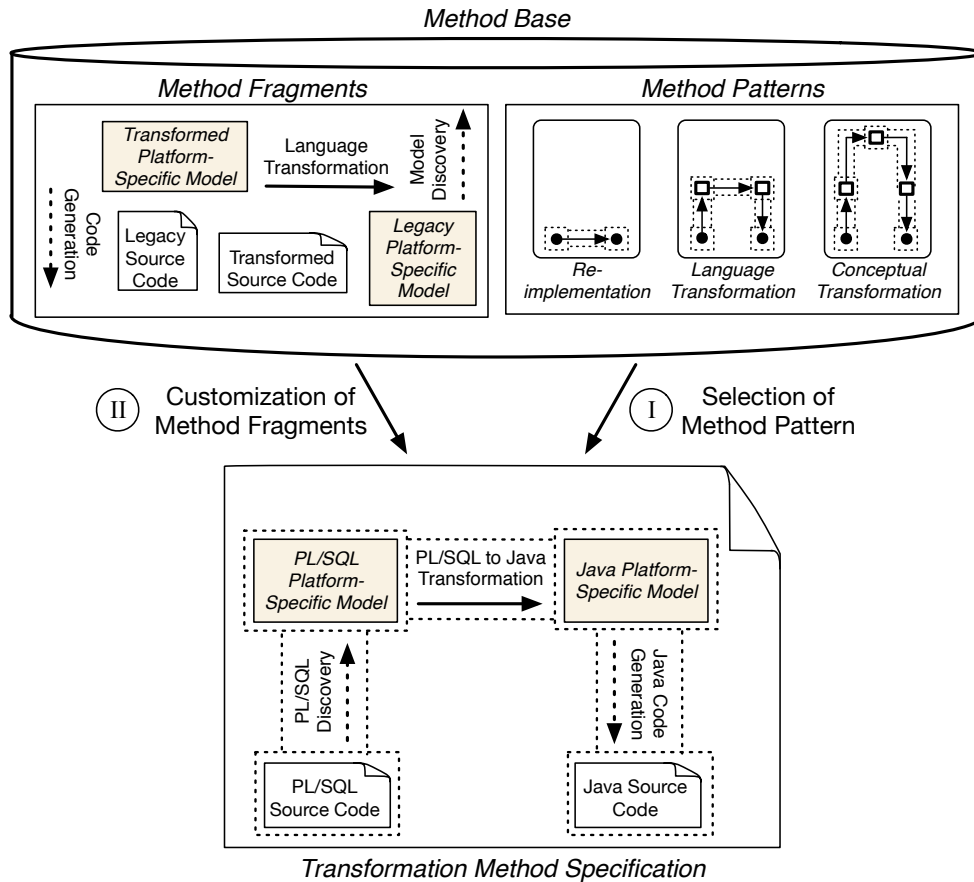


Figure 4.1 Pattern-based development of transformation methods

To get an intuitive idea for the pattern-based development process, consider a scenario in which the source and target environment use different programming languages, e.g., a transformation from PL/SQL source code to Java source code needs to be performed. In addition, a high degree of automation is required as the legacy system consists of millions of lines of code. First (I), we select a method pattern based on the identified situational context. Here, we choose the *Language Transformation* pattern. As it encodes a conversion-based transformation by performing automated transformations between the involved artifacts, we assume it to be a good fit for the observed context. Second (II), we customize the method fragments as determined by the pattern. In this example, the pattern prescribes to start with the *Legacy Source Code* artifact, which gets customized to a *PL/SQL Source Code* artifact. In this way, we end up with a situation-specific transformation method specification.

A detailed description of the proposed method base and its content is given in Chapter 5. In the next section, we give a more detailed overview of the proposed activities that make up the method engineering process.

4.1.2 Method Engineering Process

The method engineering process that is part of the MEFiSto framework describes the activities and related steps necessary in order to develop and enact a situation-specific transformation method. An overview of the core activities of the process and its relation to the proposed method base is shown in Figure 4.2.

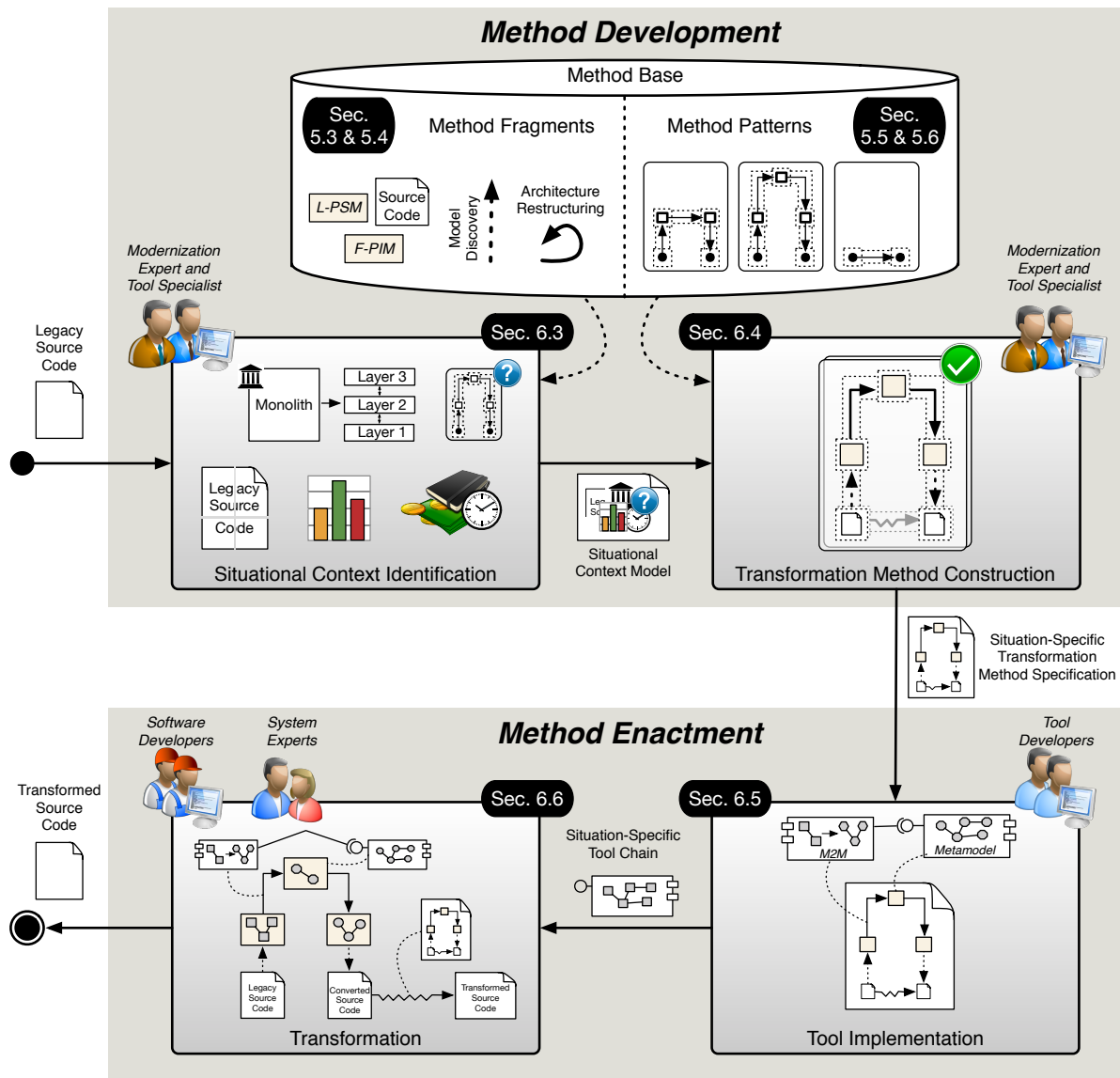


Figure 4.2 Overview of the MEFiSto framework

The *Legacy Source Code* of the system to be transformed is an essential input artifact of the method engineering process. By enacting the process, it will become *Transformed Source Code*, which constitutes the transformed system in the new environment. The process itself

can be separated into two disciplines, namely *Method Development* and *Method Enactment*. By performing activities of the former discipline, a situation-specific transformation method gets developed. The developed method is then performed by activities of the latter discipline to actually transform the legacy system. As can be seen in the figure, the transition between activities of both disciplines is associated with the flow of a *Situation-Specific Transformation Method Specification*, which is a description of the developed method. Subsequently, we briefly describe the purpose of the four core activities.

Developing a situation-specific transformation method essentially requires knowledge of the situational context, as it is a prerequisite to perform informed decisions during the development. For example, it is required to have knowledge about the characteristics of the legacy system and the target design. Systematically discovering this context is the purpose of the *Situational Context Identification* activity, it is described in detail in Section 6.3.

The situational context identified enables performing informed decisions in the construction of a transformation method. As MEFiSTo follows a pattern-based development (see 4.1.1), this essentially encompasses selecting method patterns and customizing method fragments. The purpose of the *Transformation Method Construction* is to guide that endeavor, it is described in detail in Section 6.4

Ideally, transformation methods employ a high degree of conversion, in order to reduce the overall effort and avoid errors that could result due to manual interactions [SWH10, p.131]. Therefore, for every specified activity that shall either be performed automatically or semi-automatically, a corresponding tool as part of an integrated tool chain needs to be implemented. This is performed as part of the *Tool Implementation* activity. In Section 6.5, we discuss the capabilities a generic tool infrastructure needs to provide in order to support the development of project-specific tools.

When the transformation method has been developed and required tools have been implemented, the actual transformation of the legacy system needs to occur. Corresponding parts of the method are performed within the *Transformation* activity. In Section 6.6, we discuss challenges when enacting the activities incrementally.

4.1.3 Phases & Roles

The method engineering process of MEFiSTo consists of four core activities. The rationale for this segmentation is twofold. On the one hand, the activities proposed are common for SME approaches. They are discussed on a generic level in [Bri96], we adapted them to the domain of software transformation. On the other hand, and more importantly, we aim for a clear *separation of concerns* in terms of expertise required. For example, developing a transformation method requires knowledge of software modernization, while the enactment of the developed

method requires tool development skills. Clearly distinguishing parts of the solution concept based on their required expertise will enable the targeted inclusion of external experts. The separation is addressed by associating the core activities of the MEFiSTo method engineering process with different roles and phases, as shown in Figure 4.3.

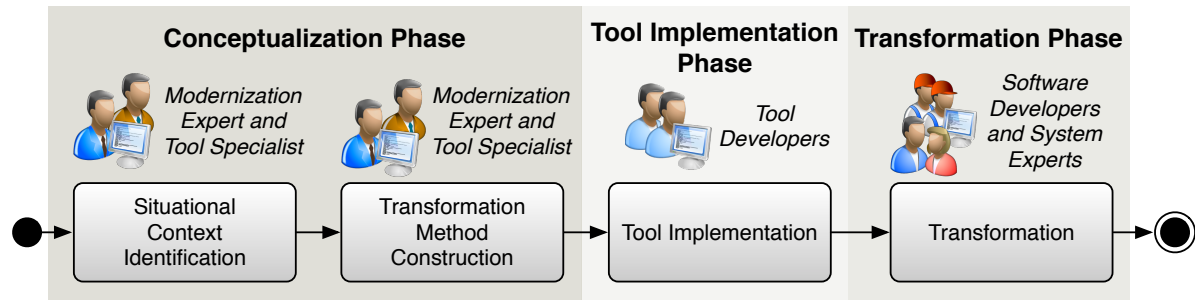


Figure 4.3 Core activities, associated roles and phases of MEFiSTo

The first phase is the *Conceptualization Phase* in which a transformation method gets developed. Both associated activities are performed by a person in the role of a *Modernization Expert*. The expert needs to have knowledge of the source and the target environment as well as of software modernization and method engineering. Knowledge of the involved environments is required to perform a systematic identification of the situational context, e.g., the exploration of the characteristics of the legacy system to transform. Knowledge of modernization enables relating the identified context to the characteristics of transformation strategies, i.e., to method patterns stored in the method base. Method engineering skills are required to perform the construction of the transformation method, e.g., the customization of method fragments (cf. 4.1.1).

In addition, both associated activities require the inclusion of a person in the role of a *Tool Specialist*. The expert needs to have knowledge of reengineering tools as well as method engineering. Knowledge of reengineering tools is essential to assess the effort required to adapt or develop tools. In particular, this comprises knowledge of model-driven reengineering tools as the MEFiSTo framework focuses on constructing model-driven tool chains (cf. Section 5.3). Method engineering skills are required to specify the use of tools or their adaptation as part of the method construction activity.

The second phase is the *Tool Implementation Phase*. It is performed to develop the tool chain that is required to automate (parts of) the transformation. The associated *Tool Implementation* activity is performed by one or multiple persons in the role of a *Tool Developer*. Tool developers need to have knowledge of developing reengineering tools and model-driven engineering. This knowledge is required to implement a tool chain as prescribed by the developed transformation method specification in advance of the actual transformation.

The third phase is the *Transformation Phase*. It is performed to actually transform the legacy system as it encompasses enacting the developed transformation method, using the tool chain developed. This activity requires general software development skills, which is provided by one or multiple persons in the role of a *Software Developer*. In addition, it can be required to include *System Experts* with specific knowledge. For example, software architects might need to perform decisions about the resulting architecture.

4.2 Evaluation Criteria

In this section, we state a set of evaluation criteria as questions that address core characteristics of the MEFiSTo framework. We answer those questions in Section 7.4 by discussing the results of the feasibility studies performed.

The evaluation criteria have been derived from the requirements stated in Section 3.2. However, they have been condensed into three questions that address the focus areas of the MEFiSTo framework, namely the *method base*, the *method development process* and the *method enactment process*. In addition, a fourth questions addresses the suitability of the framework for different environmental changes.

We formulate the corresponding evaluation criteria as follows:

EQ1: Does the content of the method base of MEFiSTo enable to develop situation-specific software transformation methods for the modernization of legacy systems?

EQ2: Does the method engineering process of MEFiSTo guide the development of situation-specific transformation method specifications?

EQ3: Can a transformation method that had been developed by applying the MEFiSTo framework be enacted to transform a real-world legacy system?

EQ4: Does the MEFiSTo framework support different environmental changes?

The first evaluation criteria aims to evaluate the content of the method base, i.e., whether the content provides sufficient *flexibility* to develop situation-specific transformation methods. A high degree of flexibility is essential in order to adapt a method to the situation at hand (cf. Requirement 1). Different capabilities can increase the flexibility provided, like the capability to vary the granularity of developed methods (cf. Requirement 4) or the inclusion of different software transformation methods (cf. Requirement 5). Besides providing flexibility, the method base shall support the modernization of legacy systems (cf. Section 2.3.1) to be suitable for the scenario described in Section 3.1.

The second evaluation criteria aims to evaluate the method development process, i.e., whether the development of a transformation method specification is sufficiently guided. Such guidance is also essential for a method engineering approach in order to be applicable in practice (cf. Requirement 2). Different capabilities can increase the control provided, like the provisioning of a continuous process or the formalization of artifacts that arise during the construction (cf. Requirement 7).

The third evaluation criteria aims to evaluate the method enactment process, i.e., whether a developed transformation method can be performed to transform real-world legacy systems. In other words, we aim to evaluate whether the framework can be applied in practice. Among other things, an essential prerequisite for the enactment of a transformation method is that the use or development of tools is addressed (cf. Requirement 6).

The fourth evaluation criteria aims to evaluate to which extent the application of the framework is limited to a specific environmental change. We specifically address this characteristic, as we intend to use the framework in different scenarios (cf. Requirement 3).

4.3 Running Example

In the next two chapters, we describe details on the constituents of the MEFiSTo framework. In addition, we show how to apply the framework to develop a situation-specific transformation method. To illustrate the descriptions in subsequent chapters, we introduce a legacy system in this section. The system will act as a running example of this thesis, i.e., we develop a transformation method for it. Here, we only give a black box description of a selected set of the legacy system's functionality and its architecture that we aim to transform. Its technical details as well as related concerns will be discussed when the example is revisited later on, e.g., how to realize the functionalities in the target environment.

In this thesis, the running example is based on a real-world modernization scenario between platforms of the vendor Oracle, namely the transformation from Oracle Forms to Oracle ADF (cf. Section 3.1). Although Oracle Forms has a large installation base, many companies consider a transformation to the more recent platform Oracle ADF for various reasons. However, transformation methods are currently developed in an ad-hoc manner. For this situation, the MEFiSTo framework is a natural fit, as it supports the systematic development of situation-specific transformation methods. Subsequently, we introduce a legacy system developed in Oracle Forms which constitutes the running example.

4.3.1 Summit

We use the *Summit*¹ application provided by Oracle as a running example. Within the Oracle community, Summit is a well-known application that has originally been developed in Oracle Forms to demonstrate the capabilities of the platform. In general, every Oracle Forms application consists of a set of *dialogs* that support use cases of the application. A dialog is an interactive user interface that enables a user to interact with an underlying data source, which usually is a database. To achieve a desired business goal, a user might use one dialog or navigate between different ones.

The main use case of the Summit application is to view and edit orders of sporting goods that have been placed by customers. The entry dialog of the application enables selecting a customer, based on their geographical location or the associated sales representative. When a customer is selected, the navigation to another dialog can be performed to view and edit an associated order. This dialog is shown in Figure 4.4, as seen by an end user of the application.

The figure shows a screenshot of the 'Orders and Items' dialog in the Summit application. The dialog is divided into three main sections: the **Toolbar Section** at the top, the **Order Section** in the middle, and the **Item Section** at the bottom. The **Toolbar Section** includes checkboxes for 'Immediate' and 'Auto Query', and buttons for 'Stock', 'Image Off', 'Help', and 'Exit'. The **Order Section** displays order details for Order Id 103, including dates, customer information, sales representative, and shipping options. It also shows a product image of a 'New Air Pump' and a warning message from the 'Forms' dialog: 'Ship date is before order date!'. The **Item Section** contains a table listing the items in the order, including 'New Air Pump' and 'Slaker Water Bottle', with their respective prices, quantities, and item totals. The total order value is 377.00.

Item Id	Product Id	Description	Price	Qty	Shipped	Item Total
1	30433	New Air Pump	20	15	15	300.00
2	32779	Slaker Water Bottle	7	11	11	77.00
Order Total						377.00

Figure 4.4 User interface (dialog) of the Summit application to view and edit order information

The dialog can be separated into three distinct sections. The *Toolbar Section* is located at the upper end of the dialog. It provides a set of checkboxes and buttons to configure the query behavior (e.g., turn auto-query on or off), to customize the visual representation (e.g., turn the product image on or off) and to navigate to other dialogs (e.g., to show the stock information). At the center of the dialog, the *Order Section* is located, which provides various information of the selected order. This encompasses the ID of the order, the ordering and shipping dates as well

¹<http://www.oracle.com/technetwork/developer-tools/forms/downloads/forms-downloads-11g-2735004.html> (accessed March 22th, 2016)

as the name of the customer and the associated sales representative. Some fields are editable, like the order and shipping date fields, changes result in an update of the corresponding dataset in the underlying database. At the lower end of the dialog, the *Item Section* is located, which shows the items of the selected order. Each item has an ID, a reference to a product from a product catalogue of sporting goods and associated pricing as well as shipping information. The button in the lower left corner can be used to add additional items to the current order.

Subsequently, we describe a selected set of functionalities realized by the Summit application. The purpose of the description is to get a high-level understanding of the application. The technical details are discussed when the running example is revisited in subsequent chapters.

Table-Based Data Access

Like most Oracle Forms applications, the Summit application stores its data in database tables. The *table-based data access* functionality refers to data structures defined within a Forms application that represent tables and enable to perform Create, Read, Update and Delete (CRUD) operations on them. We aim to transform this functionality to the target environment, i.e., we intend to provide the same data structure in the resulting Oracle ADF application. Thereby, we focus on two concepts: transforming the *internal representation of the tables* used (C1) and related *attribute validation* rules (C2).

- (C1) The usual way to access datasets stored in tables is to implement an *internal representation of the tables used* within the application. Then, changes on the internal representation are propagated to the database by the Forms platform automatically, reducing the amount of database connections necessary. Therefore, an internal representation consists of placeholders for a table and related attributes, as well as detailed properties like data types for the attributes used. In Figure 4.4, the data that is accessed when using the order information dialog can be seen. Each input or output field (boxes with a white background) contains data retrieved from a database table. The tables store datasets related to orders, customers, employees, items and products.
- (C2) When defining an internal representation for a table, some properties are essential like the attributes and their data types. However, additional properties can be implemented. An example are *attribute validation* rules that prevent the insertion of invalid data. In the right side of Figure 4.4 a pop-up can be seen that results due to an attribute validation rule. In this example, the shipping date has been changed to be earlier than the ordering date. As this is invalid, a validation rule has been defined within the Summit application that checks for this kind of error to prevent persisting flawed values.

View-Based Data Access

Each dialog of an application usually uses unique datasets, i.e., a dataset often does not correspond directly to a database table. Instead, dialog-specific database views are used whereby attributes can correspond to fields of various underlying database tables. The *view-based data access* functionality refers to internal data structures that represent such views. Besides transforming the *internal representation of views* (C3), we focus on transforming *calculated view attributes* (C4) and *view relations* (C5).

- (C3) Like accessing a table, accessing a dialog-specific view requires implementing an *internal representation of the view* within the application itself. While this representation can be based on an existing view located in the database, views usually only exist within the application. Therefore, an internal representation consists of placeholders for the view and related attributes, as well as detailed properties like data types for the attributes used. The dialog shown in Figure 4.4 uses two views: one for the order information and another one for the related items. The order information view, shown in the order section, aggregates data from three underlying tables: orders (to show shipping and ordering dates), customers (to show the customers name) and employees (to show the sales representatives name).
- (C4) Each attribute of a view can either be based on a field in a database table, or not. Fields that are not based on tables are often *calculated attributes*. The values are derived by evaluating defined expressions. The view used in the item section in Figure 4.4 consists of table-based as well as calculated attributes. The price of an item as well as how often it has been ordered (quantity) was fetched from a table. In contrast, the column at the most right position which shows the total price for each item is calculated dynamically by multiplying the price of the item and the quantity of the order.
- (C5) When a dialog uses multiple views, some of them might be related to each other due to *view relations*. Technically, such a relation is usually based on a foreign-key relationship between the tables used, but this is not a prerequisite. A common type of relation is the master-detail relationship in which the selected dataset in a master-view determines the resulting dataset in a detail-view. Such a relationship can be seen in the dialog shown in Figure 4.4 between the views of the order and the item section. Thereby, the order view constitutes the master. Its selected dataset determines the items shown.

The Form Modules of the Summit application and its contained dialogs are shown in Figure 4.5. For each Form Module, the contained dialogs as well as possible navigation flows between them are visualized. When starting the Summit application, the dialog to select a customer (a) is shown. Based on the selection, the sales representative associated can be changed (b) or the associated order can be edited (c) (cf. Figure 4.4). When editing an order, stocking information for an item can be shown (d) and new items can be added (e).

4.4 Summary

Current method engineering approaches do not provide sufficient controlled flexibility in the development of situation-specific software transformation methods. In this chapter, we introduced the solution concept of this thesis which addresses this problem.

The solution concept consists of a method engineering framework, called MEFiSTo. The framework aims to achieve a high degree of controlled flexibility by enabling a pattern-based development of transformation methods. We described its main constituents in Section 4.1, namely a method base and a corresponding method engineering process.

To evaluate characteristics of the MEFiSTo framework, we stated a set of evaluation criteria as questions in Section 4.2. In particular, the questions address characteristics of the core constituents of MEFiSTo. We revisit and answer these questions in Section 7.4 in the context of the feasibility studies performed.

To exemplify details of the MEFiSTo framework and demonstrate its application, we introduced a running example in the form of a legacy system in Section 4.3. We gave an overview of the functionality and architecture of the Oracle Forms-based application. It will be revisited in subsequent chapters whereby technical details are discussed.

In the next two chapters, we will go into detail on the two main constituents of MEFiSTo. In chapter 5, details on the proposed method base are given, while details of the corresponding method engineering process are described in chapter 6.

MEFiSTo Method Base

In the previous chapter, an overview of the MEFiSTo framework to define situation-specific transformation methods has been given. In this chapter, we introduce the content of the method base as part of the framework. First, we discuss requirements that specifically address the method base in Section 5.1. In Section 5.2, we refine the structure of the method base and describe its content. Essentially, the method base consists of two constituents, namely method fragments and method patterns. We propose a set of method fragments in Sections 5.3 and 5.4, while a set of method patterns is proposed subsequently in Sections 5.5 and 5.6. In Section 5.7, we formalize the introduced content of the method base. The findings of this chapter are summarized in Section 5.8.

5.1 Requirements

Before defining the content of the method base, we discuss related requirements. Thereby, we have to distinguish *functional* and *non-functional requirements*. When developing a method, the content of the method base will determine the possible manifestations of a method. For example, if the method base does not include a method fragment that represents an automated activity, it will not be possible to specify a tool-supported transformation method. In this sense, *functional requirements* are those that describe which methods shall be developable using the content of the method base.

The problem statement of this thesis emerged from the real-world problem of transforming Oracle Forms applications to Oracle ADF (cf. Section 3.1). Due to the differences between these environments, a transformation method that is applicable in general is not available. As a result, transformation methods are currently developed in an ad-hoc manner. We intend to address this problem by the solution concept of this thesis, that is, the MEFiSTo framework.

Therefore, Requirement 1 claims that the content of the method base, i.e., the method fragments and patterns, shall support the development of situation-specific transformation methods for the modernization from Oracle Forms to Oracle ADF.

Method Base Requirement 1 (*Support for Forms/ADF*)

The composition of content elements, stored in the method base, shall enable the development of situation-specific transformation methods for the modernization from Oracle Forms to Oracle ADF.

While functional requirements describe which methods shall be developable using the content of the method base, we also discuss the form that a developed method should take. In this sense, *non-functional requirements* are those that relate to the characteristics of the content of the method base.

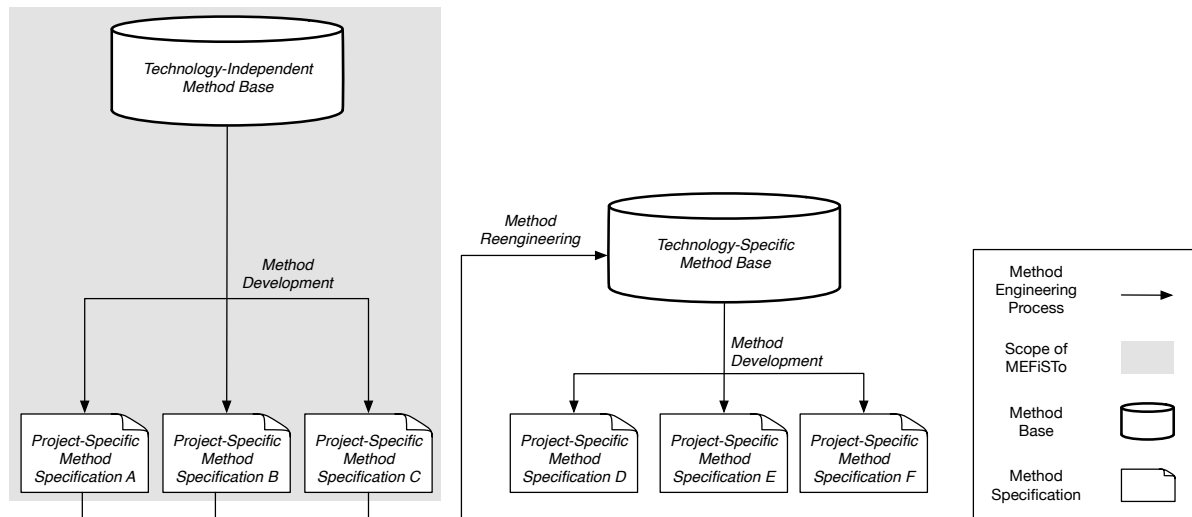


Figure 5.1 Use of a technology-independent method base in MEFiSTo (left), reengineering of developed methods to derive a technology-specific method base (right)

Requirement 1 states that the content of the method base shall support the development of transformation methods for a specific modernization scenario. More precisely, the specific technologies Oracle Forms and Oracle ADF shall be considered. An obvious solution to this requirement would be to store technology-specific method fragments and patterns in the method base. However, while the problem of transforming systems based on Oracle Forms was the main driver to define MEFiSTo, there exist legacy systems based on related technologies for which transformation methods are also developed in an ad-hoc manner. Most notably, Forms-based systems often invoke the creation of printable reports, developed in the technology Oracle Reports. Using a technology-specific method base would exclude the development of transformation methods for different technologies such as Oracle Reports.

An alternative solution is to store technology-independent content inside the method base. Then, a corresponding method engineering process needs to consider technology-specific customization of the content, as proposed in [HV97]. This will enable the development of transformation methods for other modernization scenarios but require manual effort for the customization. However, as an additional advantage we expect that developed methods can be later on reengineered to systematically derive the content of a technology-specific method base (see Figure 5.1). Therefore, Requirement 2 claims that the content of the method base shall not be associated to a specific technology.

Method Base Requirement 2 (*Technology-independence*)

The method fragments and patterns shall not be associated to a specific technology.

We intend to use the MEFiSTo framework to modernize software systems. While a software migration aims to preserve the functionality of a system while transferring it to a new environment, software modernization additionally emphasizes to adapt the system to the new environment. We assume that Model-Driven Engineering (MDE) (cf. Section 2.1) is a key principle to enable automation in software modernization scenarios. Using models enables representing a software system on higher levels of abstraction, i.e., it enables abstracting from its current technological realization. We assume that this is essential to adapt a system to its new environment. Therefore, Requirement 3 claims that the content of the method base shall be based on principles from the domain of model-driven engineering.

Method Base Requirement 3 (*MDE principles*)

The method fragments and patterns shall be based on model-driven engineering principles.

The Architecture-Driven Modernization Task Force (ADMTF) is an initiative of the OMG that applies concepts from the domain of model-driven engineering to the domain of software modernization (cf. Section 2.1.3). Its main contribution consists of various standards in the form of metamodels, like the Abstract Syntax Tree Metamodel (ASTM) or the Knowledge Discovery Metamodel (KDM). In the context of a transformation method, artifacts in the form of models can conform to these metamodels. The motivation of the ADMTF to develop these standards is to enable the reusability of tools and foster their integration. To potentially take advantage of existing tools in MEFiSTo, too, Requirement 4 claims that the content of the method base shall consider compatibility to standards from the ADM context.

Method Base Requirement 4 (*Compatibility to standards*)

The method fragments and patterns shall be compatible to standards from the ADM context.

Software transformation methods can be described formally, or not. Providing a formal description of a method requires additional specification effort, but is considered as a key success factor for modernization projects [Cab+15]. For example, it enables providing support in the enactment of a method, e.g., by using a process engine that assigns tasks to involved persons [DF94]. More importantly, we assume that a formal description will be beneficial for reusing a developed method, e.g., it allows reengineering reusable parts. It is common practice to use metamodels for formalization [RDR03]. Therefore, Requirement 5 claims to formalize the content of the method base using the Software and Systems Process Engineering Metamodel (SPEM) which is a standard [OMG08a] defined by the OMG.

Method Base Requirement 5 (*Formalization in SPEM*)

The method fragments and patterns shall be described formally by using SPEM.

5.2 Overview of the Structure

In this section, we give an overview of the structure of the method base that is part of MEFiSTo. The purpose of the method base is to provide reusable building blocks for transformation method specifications. In Section 4.1.1 we motivated that the method base contains two types of such blocks: method fragments and method patterns. In Figure 5.2, these constituents are further classified.

5.2.1 Method Fragments

Reusable method fragments form the basis of the method base. They can be seen as atomic building blocks for methods (cf. Section 2.2.1). At the highest level, we classify the method fragments based on the phase they primarily belong to, namely the tool implementation or transformation phase (cf. Section 4.1.3). While the fragments related to the tool implementation phase can be used to specify tool development activities, the fragments related to the transformation phase can be used to describe the transformation itself. For each phase, we classify the fragments based on their type.

In general, the intersection of the fragments contained in the tool implementation and transformation phase is empty, i.e., no fragment is contained in both. However, this does not mean that fragments are exclusively used in their associated phase. Instead, some fragments act as an interface between both phases. For example, model transformations are *developed* in the tool implementation phases (i.e., they are an output) but *used* in the transformation phase (i.e., they are an input). The same is true for metamodels or tools.

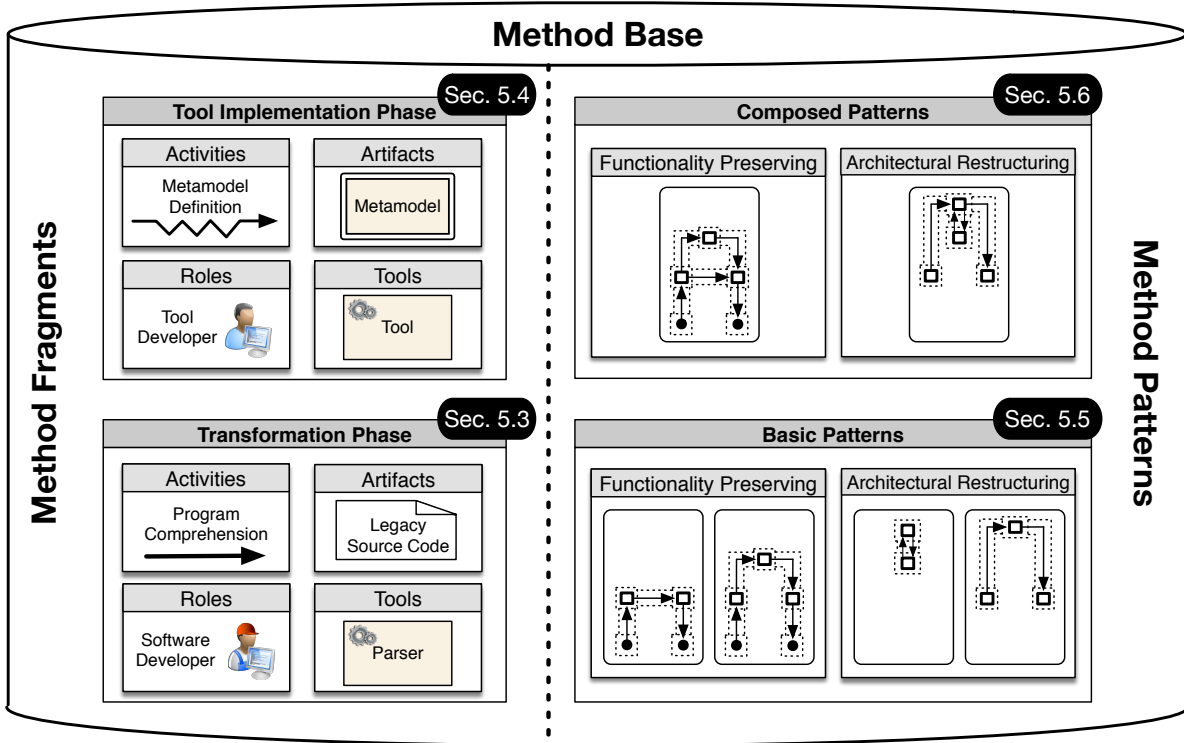


Figure 5.2 Overview of the structure of the method base in MEFiSTo

We exploit this relation between the method fragments of both phases to systematically engineer the method fragments for the tool implementation phase. In particular, we use the fact that the fragments of the tool implementation phase shall be able to express the development of tools or artifacts that are used as part of the transformation phase. Therefore, we first describe the fragments of the transformation phase in Section 5.3 and derive the fragments of the tool implementation phase subsequently in Section 5.4.

To engineer the method fragments for the transformation phase, we consider different techniques that have been developed over time. In [Ral04], these techniques are categorized into two main categories, namely the *reengineering* of existing methods and the *ad-hoc construction* of new fragments. Techniques of the first category describe different ways to disassemble existing methods into smaller fragments. To reengineer the method fragments for the MEFiSTo framework, we would require specifications of model-driven transformation methods that are not limited to specific environmental changes. As existing methods or method engineering frameworks that fulfill these requirements were not available, we applied an ad-hoc technique to construct the fragments from scratch.

For an ad-hoc technique, it is essential to clearly identify the requirements of the domain for which the development of methods shall be supported. In the case of MEFiSTo, these require-

ments have been discussed in Section 5.1. In this thesis, we meet the functional requirements if the method fragments proposed can be used to develop situation-specific transformation methods for the modernization scenario considered, that is, for the transformation from Oracle Forms to Oracle ADF. We evaluate the fulfillment of this requirement by the feasibility study described in Section 7.2. In addition, we use the feasibility studies to demonstrate the fulfillment of non-functional requirements, like the compatibility of the fragments to ADM. To demonstrate the technology-independence of the method base, we performed a second feasibility study that is described in Section 7.3. In this study, we developed and enacted a transformation method for another modernization scenario, namely the transformation from Oracle Reports to Jasper Reports.

5.2.2 Method Patterns

A method pattern is associated with a problem that shall be addressed by enacting a method. For this purpose, it encodes methodological knowledge in the form of construction guidelines for a method. In the case of MEFiSTo, a method pattern specifies which method fragments to customize and how to assemble them (cf. Section 2.2.1).

The patterns proposed in this thesis have been observed in practice, when developing transformation methods for the transformation from Oracle Forms to Oracle ADF. Therefore, each proposed pattern is associated with the problem of transforming legacy systems into new environments. In this context, one particular problem is to retain the functionality of the original system [Bis+99]. The *functionality preserving* patterns address this issue by encoding methodological solutions. In particular, each pattern follows a specific transformation strategy, e.g., a pattern can follow a conversion or reimplementation-strategy. Besides retaining the functionality of a system, another problem is to adapt its structure to the new environment. This issue is addressed by the *architectural restructuring* patterns, which encode methodological solutions to change the architecture of a legacy system during its transformation.

After having identified the patterns, we noticed that we can distinguish at least two types of them: *atomic* and *non-atomic* ones. The atomicity of a pattern arises from the fact that a pattern becomes invalid when any mandatory fragment is removed, that is, it does not fulfill its methodological purpose. In this thesis, we call an *atomic* method pattern a *basic pattern*. In contrast, non-atomic patterns arise by combining one or more basic patterns. In this thesis, we call them *composed patterns*. As each basic pattern follows a specific transformation strategy, composed patterns can be beneficial as their solution consists of combining different strategies. We describe basic patterns in Section 5.5, followed by composed patterns in Section 5.6.

In MEFiSTo, the development of transformation methods can be seen as being pattern-based (cf. Section 4.1). In order to select one or multiple of the proposed patterns for a situation

observed, it is essential to understand the characteristics of each pattern. Therefore, we describe each pattern according to the schema shown in Table 5.1 that we derived from the schema defined in [Gam+95, pp.6-7].

Intent	Which problem does the pattern address?
Strategy	Which methodological solution does the pattern provide?
Structure	The structure of the pattern, depicted as a path in the horseshoe model (cf. Figure 5.4)
Applicability	In which situations is the pattern suitable? What are the most important influence factors on its efficiency or effectiveness?
Preparation	Which artifacts or tools have to be developed in advance of the transformation when applying the pattern?
Example	An example of the pattern's application on the running example (cf. Section 4.3)
Known Uses	How do existing techniques relate to the pattern? What are examples of existing methods that (partially) conform to it?
Related Patterns	Relations to other patterns proposed

Table 5.1 Schema to characterize method patterns

The schema provides a condensed, tabular description that summarizes the most important characteristics of each pattern briefly. To understand these characteristics in detail, i.e., the rationale for each characteristic, we additionally provide an interrelated, extensive description. For this description we use the following structure:

1) Example For each pattern an *example* is introduced, i.e., a method that conforms to the pattern. The method specifies how to transform parts of the running example that has been described in Section 4.3. The application shall demonstrate how the pattern addresses the problem of transforming selected parts of an application into a new environment. It is used as a reference when discussing specific characteristics.

2) Description We give a *generic description* of each pattern, i.e., which problem it addresses and which methodological solution it provides. Thereby we refer to the example to substantiate the generic characteristics described. We discuss its relation to already existing techniques and mention transformation methods in literature that conform to it.

3) Suitability Finally, we discuss the *suitability* of each pattern, i.e., in which situations it is appropriate. To do so, we identify influence factors of a modernization project that help a modernization expert to assess the efficiency and effectiveness of a method pattern.

We discuss the suitability of each method pattern separately, as an understanding of how to evaluate the suitability is essential to develop situation-specific methods. To be able to do so, we want to point out when we consider a method to be situation-specific. In this thesis, we define the situation-specificity of a method as follows:

Notation 12 (Situation-Specificity)

A transformation method is situation-specific, if it is efficient and effective against the background of the associated situation. Efficiency relates to properties of the enacted process, while effectiveness relates to properties of the transformed functionality.

As a method pattern provides construction guidelines for a method, we also use Notation 12 to assess the situation-specificity of a pattern. On the one hand, the efficiency of a pattern is related to characteristics of the *resulting* process. While this can refer to any characteristic of the process, we exemplify and discuss the patterns by focusing on the *effort* required. In this sense, the most efficient method pattern is the one which results in a process that can be enacted with the least effort required. On the other hand, the effectiveness of a pattern relates to characteristics of the *resulting* transformed functionality. While this can refer to any characteristic of the system, we exemplify and discuss the patterns by focusing on the *maintainability*. In this sense, the most effective method pattern is the one which results in a transformed system that is the easiest to maintain.



Figure 5.3 Simplified conceptual model of the main concepts that are involved when assessing the suitability of a method pattern

To determine the situation-specificity of a method, it is common practice in situational method engineering approaches to use a model of the situational context, consisting of a set of influence factors [HS+14, pp.8-10]. In this thesis we adopt this view, resulting in the

conceptual model shown in Figure 5.3. It describes the main concepts that are involved when assessing the suitability of a method pattern. We consider *Influence Factors* as characteristics of a *Modernization Project*. Each factor has an impact, in the sense that it influences the efficiency or effectiveness of the method that results when applying the pattern. Then, the suitability of a pattern needs to be assessed based on the impact of its *Influence Factors*.

5.3 Transformation Phase Fragments

In this section, we introduce method fragments for the transformation phase that are used to specify the actual transformation. To describe these fragments, we use the fact that tool-supported transformation methods are instances of the established horseshoe model [KWC98] (cf. Section 2.3.1). Therefore, the fragments proposed are shown as part of a connected horseshoe model in Figure 5.4, which is a common way to describe reengineering methods.

The model shows an artifact-centric view, that is, it shows artifacts on various levels on abstractions as well as activities that consume and produce them. Subsequently, we describe all fragments shown in Figure 5.4 separately. Thereby, we structure the section according to the different type of method fragments, i.e., we first describe activities and artifacts, followed by roles and tools. We describe the fragments on the M1 layer (cf. Section 2.2.3), although in an informal way. The formalization is introduced in Section 5.7, when all fragments and patterns have been described.

Artifacts

A transformation method that is based on the horseshoe model essentially requires specifying artifacts to be created as part of the transformation. The method fragments that constitute artifacts can be distinguished based on their corresponding abstraction layer, namely the *System-*, *Platform-Specific*, or *Platform-Independent Layer*.

On the *System Layer*, textual artifacts are located that represent source code. This can either be the *Legacy Source Code* of the existing system, or the resulting *Transformed Source Code*. Besides textual artifacts, external systems like *Platforms* or *Databases* are also located there. If the legacy system depends on them, e.g., by using their interfaces, it can be necessary to capture them as part of a model, too.

On the *Platform-Specific Layer*, Platform-Specific Models (PSMs) are located that represent the legacy system (*L-PSM*) and the transformed system (*T-PSM*) respectively. These models describe the source code of the legacy system and its environment by modeling the corresponding Abstract Syntax Trees (ASTs) or Abstract Syntax Graphs (ASGs) (cf. Section 2.3.2). They

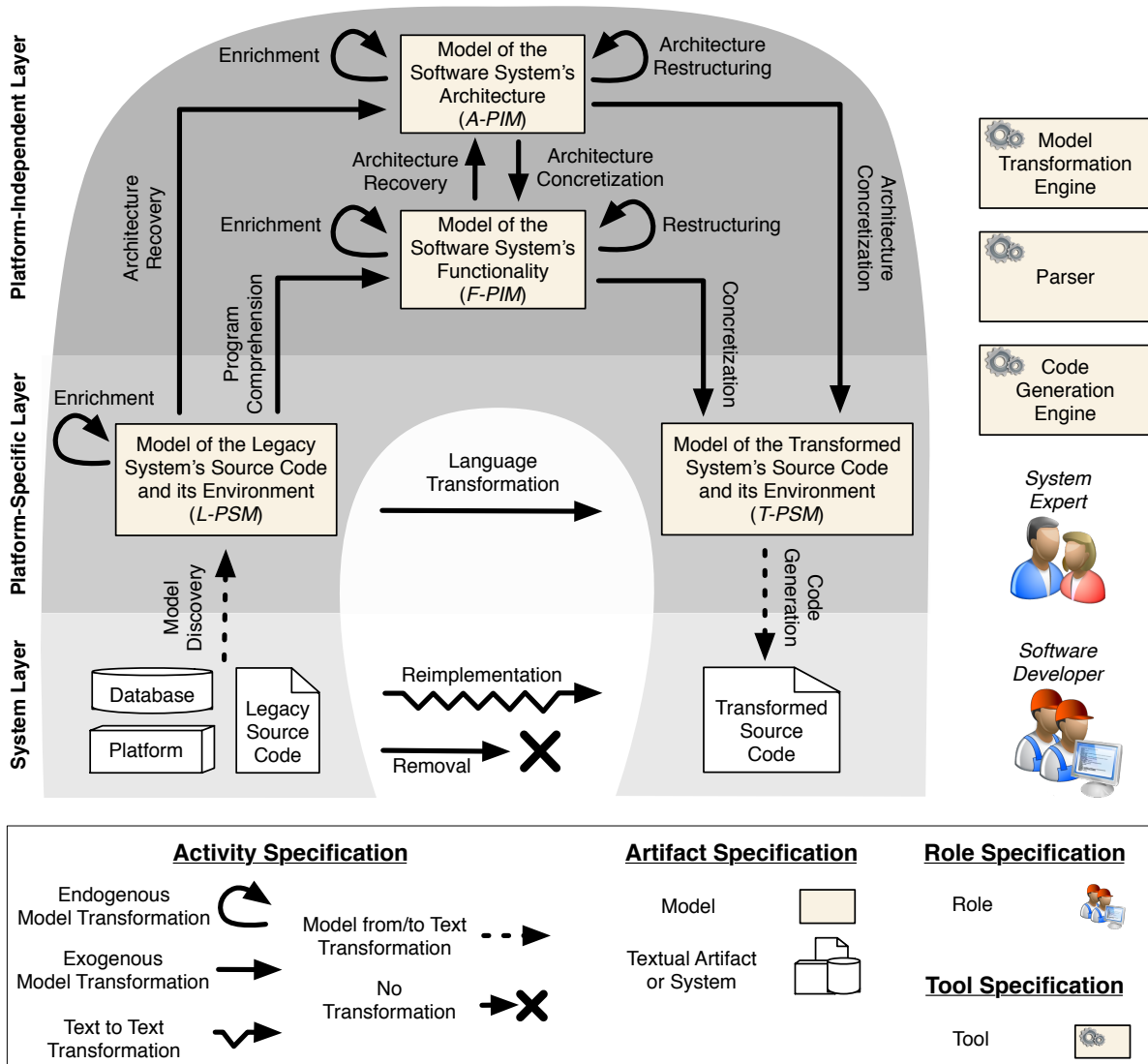


Figure 5.4 Method fragments of the transformation phase, visualized as a horseshoe model

are considered to be platform-specific since technology-specific concepts are used to model the system, e.g., by using a metamodel of a specific programming language like PL/SQL. From the ADM context, the Abstract Syntax Tree Metamodel (ASTM) [OMG11a] can be extended to derive such a platform-specific metamodel (cf. Section 2.1.3). The resulting metamodel is then called Specialized Abstract Syntax Tree Metamodel (SASTM).

On the *Platform-Independent Layer*, Platform-Independent Models (PIMs) are located that act as an intermediate representation in the transformation. We distinguish two kinds of them, based on the abstraction level of the contained information.

The model on the lower level of abstraction (*F-PIM*) represents the functionality of the system to transform by technology-independent concepts. For example, by using a metamodel of general programming language concepts, like loops, conditions or function calls, a platform-independent ASG can be modeled. However, the model is not limited to solely describing source code but any information that represents the functionality of the system. Such functionality is often only implicitly described by the source code, like states of the system, internal data structures or structures of user interfaces.

On the highest level of abstraction, we consider a model that represents architectural structures of a software system (*A-PIM*), e.g., a model representing components or layers. Architectural structures usually aggregate entities that are represented by a model on a lower level of abstraction, e.g., a component can consist of classes in an object-oriented system.

From the ADM context, the Knowledge Discovery Metamodel (KDM) [OMG11b], which is separated into various layers and packages (cf. Section 2.1.3), can be used for both platform-independent models.

Activities

Activities in the horseshoe model produce and consume related artifacts. In Figure 5.4 we distinguish five different types of activities, e.g., *Endogenous Model Transformations* or *Text to Text Transformations*. We want to point out that these types describe the artifact-related transformations that need to occur when enacting the activity. For example, enacting a *Language Transformation* activity results in the execution of a model transformation that transforms the L-PSM into the T-PSM. Nevertheless, we assume that the same activity might consist of sub-activities that form a process itself. For example, the *Language Transformation* activity may consist of two sub-activities: In the first activity, a developer needs to provide some input which is then used as a parameter for a model transformation that is executed in a second activity.

The method fragments that constitute activities can be distinguished based on the part of the reengineering process they belong to. In the following we use the classification described in [CC90] that distinguishes between the phases of reverse engineering (going upward), restructuring (going horizontally), and forward engineering (going downward).

Reverse engineering is the process of analyzing a subject system to create representations of the system in another form or at a higher level of abstraction. In this context, the *Model Discovery* [Bru+14] activity represents a bridge between two *technological spaces* [AKB02] as it specifies the transformation of entities of the *grammarware* space to entities in the *modelware* space. Techniques like *syntactic* and *semantic analysis* (cf. Section 2.3.2) are applied on source code to derive the AST or ASG. The *Program Comprehension* [Rug95]

activity specifies to perform an abstraction from platform-specific concepts by interpreting the corresponding Legacy Platform-Specific Model (L-PSM). Usually, this activity is performed to make information explicit that are described implicitly by the source code, requiring expert knowledge about the legacy system or the platform and its behavior. The *Architecture Recovery* activity requires this knowledge too, but specifies the extraction of architectural information, e.g., by applying clustering techniques.

Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior. On the platform-independent layer, we foresee to perform restructurings of the F-PIM as well as of the A-PIM. The *Language Transformation* activity specifies the performance of a direct transformation between both Platform-Specific Models (PSMs), which requires defining mappings between the programming languages used. The *Reimplementation* activity is performed manually by software developers. Thereby, developers explore the functionality of the legacy system and reimplement it in the new environment. *Enrichment* is an activity that can be performed on various models, i.e., the L-PSM, F-PIM or A-PIM can be enriched. It specifies the addition of external information to an existing model, e.g., by annotating parts of a model. The activity can either be performed manually by a person with expert knowledge of the information that shall be incorporated, or automatically. The *Removal* activity specifies to not perform any transformation of the code to which it is related. Compared to the other activities, the *Removal* activity is the only one that does not produce an output.

Forward engineering is the process of moving from high-level abstractions to the physical implementations of a system. In this context, *Architecture Concretization* specifies the propagation of changes on the architectural level to the platform-independent or platform-specific level. By performing a *Concretization*, the platform-independent representation of some functionality is transformed to a platform-specific one. *Code generation* is the opposite of *Model Discovery*, it specifies the generation of source code of the transformed system based on the T-PSM.

Note that the Figure does not state which activities are performed automatically and which are not. In fact, we only assume that the *Reimplementation* activity is always performed manually. All other activities can be completely automated, but might also include some manual interaction. For activities belonging to the *reverse* or *forward-engineering* process, we assume that the model transformation itself is performed automatically by a corresponding engine. However, a human might be involved in certain steps (i.e., sub-activities) of the activity. For example, a person in the role of a *System Expert* might evaluate intermediate results, before the model transformation is finally executed. For activities belonging to the restructuring process, we assume that they can have any degree of automation. As stated before, it might even be the case that such an activity is performed completely manually.

Roles

Two roles are associated with the transformation phase, as discussed in Section 4.1.3. For both roles, we provide corresponding method fragments. First, a *Software Developer* can be involved who is responsible for performing reimplementation activities. The developer can have expertise in developing software for the new environment, or not.

Second, a *System Expert* can be involved that has specific knowledge of certain aspects. One example for a system expert is a developer of the legacy system. His knowledge can be used during the transformation by letting him perform enrichment activities. For example, a system developer can be responsible for annotating all database tables in a database model for which write access is required by the legacy system. Another example for a system expert is a *software architect* for the new system who can be involved in an architectural restructuring. For example, such a restructuring can require performing decisions between multiple alternatives when it comes to clustering parts of the software, or it can require deciding on appropriate names for resulting clusters.

Tools

Based on the activities introduced, we can derive three tools that will be required by any transformation method that has been developed using the MEFiSTo framework and automates part of the transformation.

First, a *Parser* is required to transform textual source into a model. Second, as created models get transformed into each other by specified model transformations, a *Model Transformation Engine* is required to execute them. Finally, code that realizes the transformed system needs to be generated out of a model. A *Code Generation Engine* is required to execute the code generation rules that are used for this purpose.

5.4 Tool Implementation Phase Fragments

In this section, we introduce method fragments for the tool implementation phase that are used to specify the development of required tools. They can be derived from the method fragments that specify the actual transformation, which we introduced in the previous section. For example, the fragments of the transformation phase enable to specify that a *Language Transformation* activity shall be performed which transforms one platform-specific model (L-PSM) into another one (T-PSM). Performing this activity requires the availability of an artifact, in this case, it requires corresponding *Model Transformation Rules*. Developing such rules is a necessary preparation.

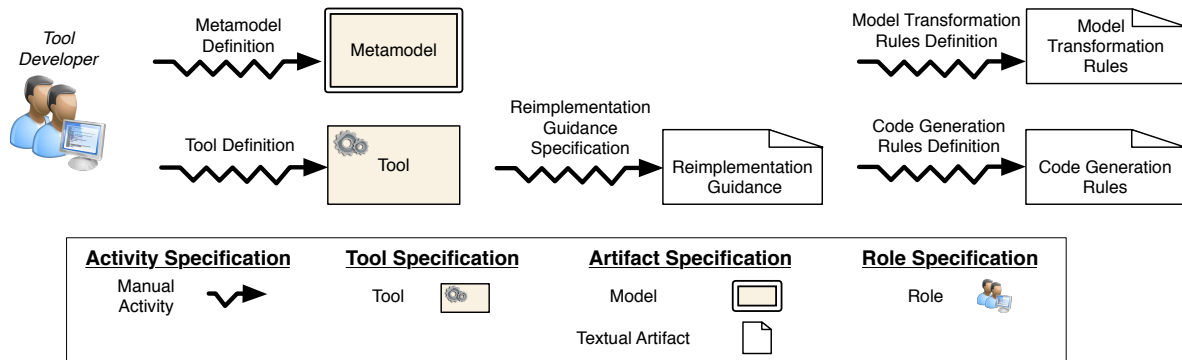


Figure 5.5 Method fragments of the tool implementation phase

Based on this relation between the fragments of both phases, we derived the fragments for the tool implementation phase from the ones of the transformation phase. The resulting activities, artifacts and tools are shown in Figure 5.5.

Artifacts & Activities

We foresee six method fragments that constitute activities and corresponding artifacts. As automated transformations are based on the use of models, corresponding metamodels are required. The *Metamodel Definition* activity represents the definition of a required metamodel, while the artifact itself is specified by the *Metamodel* fragment. We use the term *definition* for the activity as we foresee different ways on how to realize the metamodel. First, a metamodel can be *developed* from scratch. However, in the context of a transformation method, metamodels are required for the programming languages that are used in the source or target environment. These metamodels are in general well-defined and stable. Therefore, secondly, existing metamodels can be *reused* if available. However, it might still be beneficial to adapt them to specific needs. Therefore, thirdly, profiles [Des00] can be used to *adapt* an existing metamodel.

Various activities require the use of tools. Performing the *Model Discovery* activity requires using parsers, *Program Comprehension* can require reverse engineering tools and an *Architectural Restructuring* might require a clustering tool. The corresponding *Tool Definition* activity represents the development of such a required tool. Like metamodels, tools can be developed from scratch or an existing one can be used. Note that the output of this activity is not an artifact but a tool, in contrast to the other activities.

Activities that constitute transformations between models are realized by automatically executed model transformations. The activity called *Model Transformation Rules Definition* represents the definition of required model transformation rules. The *Model Transformation Rules* fragment represents the output of the activity, that is, the resulting artifact. Similar

fragments are required for the *Code Generation* activity. The activity called *Code Generation Rules Definition* represents the definition of corresponding rules, while the *Code Generation Rules* fragment represents the resulting artifact.

As the *Reimplementation* activity is performed manually by software developers, it needs to be guided. The activity called *Reimplementation Guidance Specification* represents the definition of such guidance. For example, a tool developer can provide a step-by-step instruction on how to perform the reimplementation. The *Reimplementation Guidance* fragment represents the corresponding artifact.

Roles

One role is associated with the tool implementation phase, namely a *Tool Developer* (cf. Section 4.1.3), for which we provide a corresponding method fragment. The tool developer is responsible for the activities introduced, e.g., the definition of metamodels, reimplementation guidance or any tool required. Therefore, we assume that a person in this role has comprehensive knowledge of model-driven engineering and developing reengineering tools.

Tools

In terms of tools, we have to distinguish between two types of tools: those, which are required by any transformation method and those, which have to be specifically developed for a method.

On the one hand, some tools are required by each developed method that specifies to automate part of the transformation. For example, a model transformation engine is required to execute model transformation rules. Method fragments that describe these kinds of tools have already been introduced as part of the transformation phase fragments (cf. Section 5.3).

On the other hand, some tools have to be specifically developed for a defined method. Examples encompass semantic analyzers, clustering or reverse engineering tools. We provide a generic fragment to specify these kinds of tools.

5.5 Basic Transformation Patterns

In this section, we introduce a set of basic patterns, shown in Figure 5.6. Each pattern is associated with a path in the horseshoe model that we introduced in the previous Section (cf. Figure 5.4). The path visualizes the solution provided by a pattern, as it informally indicates which method fragments essentially or optionally shall be customized when applying it. For example, an application of the *Reimplementation Pattern* F_3 requires customizing the *Legacy*

Source Code artifact, the *Reimplementation* activity as well as the *Transformed Source Code* artifact. A formal description of the patterns is given at the end of this chapter in Section 5.7.

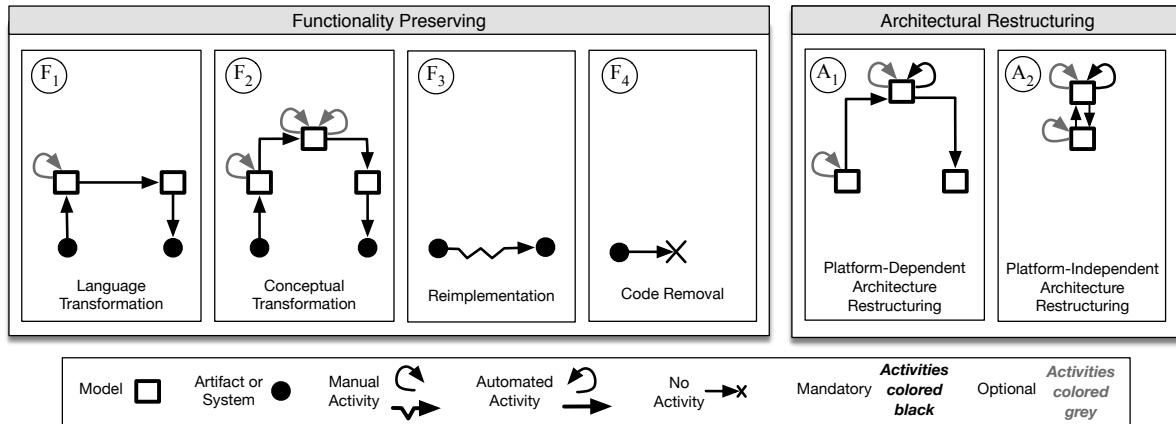


Figure 5.6 Basic transformation method patterns

Subsequently, we begin introducing each pattern with an example, i.e., we describe the enactment of a method that conforms to the pattern. Thereby, the method is used to transform parts of the Summit legacy system which has been introduced as a running example in Section 4.3. We omit a description of the pattern A_2 , as it differs only marginally from A_1 . Its characterization can be found in the appendix (cf. Section A).

So far, we described the functionality and architecture of Summit on a high level of abstraction. In particular, we omitted technical details and a mapping of the functionality into the target environment. In terms of the mapping, we use the one described in [RM11] that has been defined as part of a case study. For each mapping used, we provide a description of the technical background. We assume that a detailed technical example for each pattern is essential to create an understanding of its general characteristics. The first pattern we introduce is the *Language Transformation Pattern*. Its methodological solution to preserve functionality consists of converting it, using an automated, model-driven tool chain.

5.5.1 Language Transformation (F_1)

Intent	Perform an automated transformation of the legacy system's functionality into a new environment, following a conversion-based transformation strategy
---------------	---

Continued on next page

Strategy	Definition of a direct mapping between the programming languages of the environments involved. This is realized by representing the legacy system as a model of its ASG ¹ on a platform-specific layer. A model transformation that transforms this model into an ASG ¹ of the target environment is a realization of the mapping between the programming languages involved
Structure	
Applicability	Use when the functionality to transform is realized comparably in the legacy and target environment and the legacy system has a sufficient size. The difference in the realization determines the complexity of the mapping between the programming languages involved, influencing the efficiency and effectiveness of the pattern
Preparation	Applying this pattern essentially requires realizing a parser, model-to-model transformations and code generation rules. Also, it can be necessary to realize a semantic analyzer or model views
Example	Figure 5.7 shows the enactment of a method which conforms to the pattern. Thereby, the internal representation of database tables of the running example is transformed (cf. Section 4.3).
Known Uses	The strategy realized by the pattern is comparable to the strategy followed by a compiler. However, compiler design usually focuses on automating the transformation. In contrast, the pattern also enables developing semi-automatic methods, which can be necessary to end up with a situation-specific method. The method described in [Fuh+12, pp.174-178] conforms to the pattern
Related Patterns	/

Table 5.2 Characterization of the *Language Transformation Pattern*

Example

This section is separated into three parts. First, we introduce the (i) technical background required to understand the example. Then, we describe the (ii) enactment of a method that conforms to the *Language Transformation Pattern*. As this is the first pattern we introduce, we describe how the method has been (iii) developed by using the MEFiSTo framework.

Technical Background First, we need to understand the concept, i.e., the functionality, which we aim to transform. In this example, we aim to transform a part of the *table-based data access* functionality realized in the Summit application (cf. Section 4.3). More specifically, we demonstrate the transformation of the *internal representation of the tables* used (C1). Such an internal data structure is beneficial for a database-intensive application, as the underlying platform can take care of populating it and propagating changes made back to the database.

Second, we need an understanding of how the concept is realized in the legacy system. In Oracle Forms-based applications like Summit, provided language constructs can be used to specify the structure of an internal representation, namely Blocks and Items [Ora00a].

We can distinguish two types of Blocks: those, which are associated to a database table (called *data* Blocks), and those, which are not (called *control* Blocks). Data Blocks can be seen as a placeholder for one or multiple datasets, i.e., rows of a database table. Blocks are an example for declarative language constructs provided by Oracle Forms. Such language constructs have associated properties to specify their details. For example, a Block has the property `DatabaseBlock` which can be set to `True` or `False`, indicating whether the Block is a data Block, or not. If a data Block is defined, the property named `QueryDataSourceName` is used to indicate the database table whose datasets the Block represents. Related to the Summit application, an excerpt of the definition of the Block called `S_ORD` can be seen in the artifact called *Summit Block Source Code*, shown in the lower left of Figure 5.7. It shows that `S_ORD` is a data Block which refers to the identically named table.

In addition, a Block serves as a container for Items. Just like Blocks, there are *control* and *data* Items, indicated by the Boolean property `DatabaseItem`. A data Item can be seen as a placeholder for a field in a dataset, i.e., an entry in a column of a database table. Related to the Summit application, an excerpt of the definition of Items that are part of the `S_ORD` Block can also be seen in the figure. It shows the definition of various Items that store the ID of an order (ID), the ID and the name of a related sales representative (`SALES_REP_ID`, `SALES_REP_NAME`) as well as the ordering and shipping dates (`DATE_ORDERED`, `DATE_SHIPPED`).

Blocks and Items can be used to define the structure of the internal representation of datasets. In order to transform this functionality, we also need to have knowledge of how the

¹Depending on the situation, using an AST can be sufficient

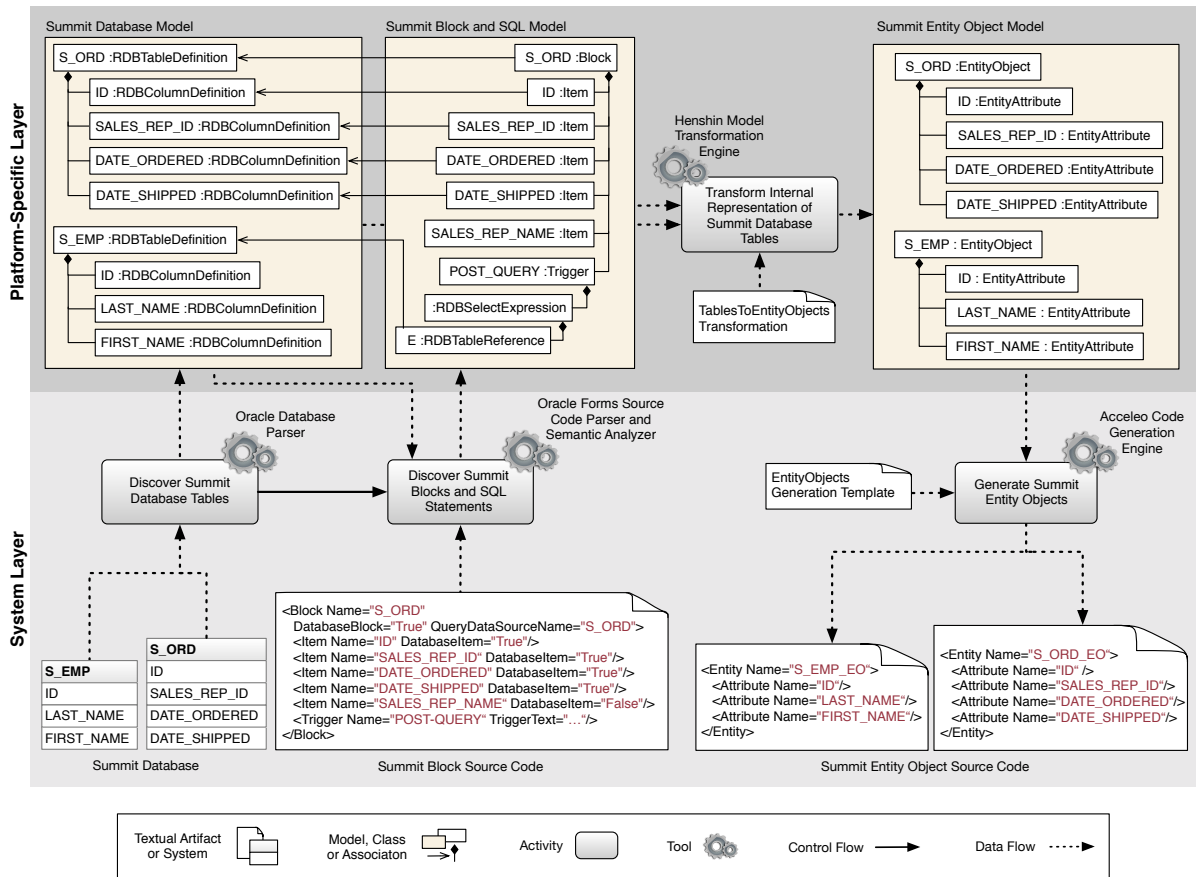


Figure 5.7 Enacting a transformation method to transform the internal representation of database tables of the running example by using the *Language Transformation Pattern*

connection to the underlying database is realized. In general, there are two ways to realize this connection: declaratively and imperatively. If a data Block defines a relation to a table using the `QueryDataSourceName` property, each Item whose name corresponds to a column in that table is populated automatically. In this way, the connection is realized declaratively, which is the case for most Items in the Summit application.

The `SALES_REP_NAME` Item shown in the lower left of Figure 5.7 reveals that it is a control Item, as the `DatabaseItem` property is set to `False`. In fact, this Item is populated imperatively by executing code implemented as part of a Trigger. A Trigger is a declarative language construct that contains program code, written in the programming language PL/SQL. Each Trigger is executed automatically by the Forms runtime when an associated, predefined point in time is reached or an event occurs. Related to the Summit application, the `SALES_REP_NAME` Item is populated by the code contained in the `TriggerText` property of the Post-Query Trigger, shown in Listing 5.1. The Trigger belongs to the same Block as the Item and is executed after a dataset has been fetched by the runtime, but before the result is

displayed to the user. Here, an SQL statement is executed that, among other things, retrieves the name of the sales representative and stores the result in the corresponding Item.

```

1 SELECT E.last_name
2 INTO :S_ORD.sales_rep_name
3 FROM S_EMP E
4 WHERE :S_ORD.sales_rep_id = E.id;
```

Listing 5.1 Source code of the POST-QUERY Trigger which is a part of the S_ORD Block

The reason for why different ways are used to realize the connection between the internal representation of datasets and the database can be seen by having a look at the database schema, shown in lower left side of Figure 5.7. While the S_ORD Block and most of its Items correspond to the S_ORD table and its attributes, the value of the SALES_REP_NAME attribute is retrieved from another table called S_EMP. In other words, the imperative program code joins both tables, using the SALES_REP_ID attribute as a foreign key.

Besides knowledge of the concept and its realization in the legacy system, we thirdly need an understanding of how the concept shall be realized in the target environment Oracle ADF. Here, we will use the mapping described in [RM11, p.12]. In this case study, Entity Objects are used for the internal representation of the tables used by the application. Like a Block, an Entity Object can be used to represent a dataset stored in a table [Ora13]. It consists of a set of Entity Attributes that can correspond to columns of a table.

Based on the description, one could get the impression that each Block needs to be transformed into an Entity Object, while Items need to be transformed to Entity Attributes. However, this is not the case as a Block does not necessarily represent a database table but rather a database view. An example is the S_ORD Block, which aggregates the data of multiple tables. Instead, we need to identify the tables that are actually used by the Summit application and create corresponding Entity Objects for them.

Method Enactment The desired transformation is achieved by enacting the transformation method as shown in Figure 5.7. Note that most of the artifacts are presented shortened. The method consists of customized method fragments that have been introduced in Section 5.3 and is an instance of the *Language Transformation Pattern*.

In the beginning, two activities need to be performed that are customizations of the *Model Discovery* fragment, namely the activities called *Discover Summit Database Tables* and *Discover Summit Blocks and SQL Statements*. The activities gather required information related to the functionality to transform and represent them homogeneously as an L-PSM. The *Discover Summit Database Tables* activity is performed automatically by using an *Oracle Database Parser*. During its performance, the schema of the database used by the application gets parsed.

This encompasses parsing the tables and their attributes as well as related details, like the data types. The output of the activity consists of a model representing the database schema. Related to the Summit application, the discovery of the tables S_EMP and S_ORD is shown in the lower left of Figure 5.7. Their parsing results in the *Summit Database Model* which conforms to the Generic Abstract Syntax Tree Metamodel (GASTM) defined in [OMG11a, pp.115-123].

The *Discover Summit Blocks and SQL Statements* activity is also performed automatically, but using an *Oracle Forms Source Code Parser and Semantic Analyzer*. Parsing Oracle Forms source code requires parsing the Forms-specific, declarative language constructs, like Blocks, but also the parts written in the imperative programming language PL/SQL. The output consists of a model representing the source code. Related to the Summit application, the discovery of the S_ORD Block is shown. The resulting model represents the Block and its Items as well as a decomposition of the SQL statement that is stored in the related Trigger. The model conforms to a Specialized Abstract Syntax Tree Metamodel (SASTM) [OMG11a] that has been defined for the programming language of Oracle Forms (cf. Section 2.1.3). Note that associations exist between the model of the source code and the model of the database. For example, the relation between the Block called S_ORD and the equally named table is represented by an association between the classes S_ORD:Block and S_ORD:RDBTableDefinition. This is due to the fact that the discovery step not only considers parsing, that is, syntactic analysis, but also semantic analysis [Aho+06, pp.8-9]. This results in the addition of semantic edges [KGW98], i.e., edges that do not belong to the tree structure of the AST, making the L-PSM an ASG (cf. Section 2.3.2). Here, the association between the data Block and the table to which it refers to is an example for such an edge. This analysis is also the reason why the discovery of the source code is performed after the discovery of the database tables. A model representing the tables is a prerequisite to resolve the targets of such edges.

After the L-PSM has been created, it gets transformed into a T-PSM by performing the activity called *Transform Internal Representation of Summit Database Tables*. This activity is a customization of the *Language Transformation* fragment which prescribes to perform a direct transformation between the programming languages used. Here, this is realized by executing a set of model transformation rules by a corresponding engine that transform the L-PSM into the T-PSM. An excerpt of these rules for the example can be seen in Figure 5.8. Executing the rule TableToEntityObject instantiates an Entity Object class for each table that is referenced within a Form Module by a data Block. The ColumnToEntityAttribute rule instantiates the corresponding EntityAttributes. Note that these rules need to be executed repeatedly, as long as one of them is applicable. The resulting model is shown in the upper right side of Figure 5.7. It conforms to an SASTM [OMG11a] that has been defined for the programming language of Oracle ADF.

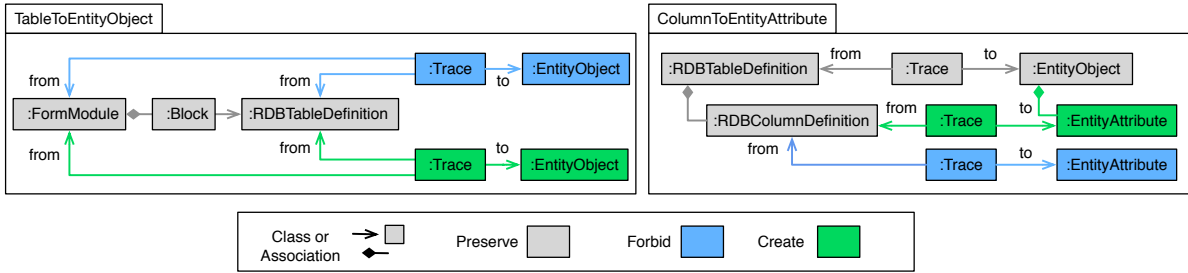


Figure 5.8 Excerpt of the model transformation rules to transform the L-PSM into the T-PSM

As a last step, the code is generated based on the T-PSM by performing the activity called *Generate Summit Entity Objects*. This activity is a customization of the *Code Generation* fragment which describes the execution of a set of code generation rules by a corresponding engine to transform the T-PSM into textual source code. Here, a code generation technique is applied that is based on the use of an abstract definition of the code to generate [Her03, pp.87-96], which in this case is the role of the T-PSM. The T-PSM can be considered as a parameter for the code generation template in which variable parts are exchanged during execution. The resulting source code can be seen in the lower right side of Figure 5.7. In Oracle ADF, *Entity Objects* are defined by XML files, as shown.

This concludes the description of the enactment of the method. Subsequently, we describe how it has been developed.

Method Development The development of the transformation method occurs on the Method Specification Layer (M1) (cf. Section 2.2.3) in advance of its enactment. An overview of the development is shown in Figure 5.9. To clarify the relation between the activities and artifacts on the M0 and M1 Layer, the figure shows both layers and relationships between them.

In the upper right of Figure 5.9, the performance of the activities called *Situational Context Identification* and *Transformation Method Construction* is shown. Both activities are core activities of the method engineering process of the MEFiSTo framework (cf. Section 4.1.2). Subsequently, we describe the activities in the context of the example. At this point, we do not go into detail but aim to motivate the fact that both activities are critical for the development of situation-specific transformation method. Details of these activities are described in Chapter 6.

One purpose of the activity called *Situational Context Identification* is to gather knowledge that is required to develop the method. This knowledge at least encompasses the technical background that we described in the beginning of this section, i.e., knowledge of the functionality to transform, its realization in the legacy system as well as the desired realization in the target environment. Without this knowledge, informed decisions on how to perform the transformation would not be possible. In the example, without having knowledge of the

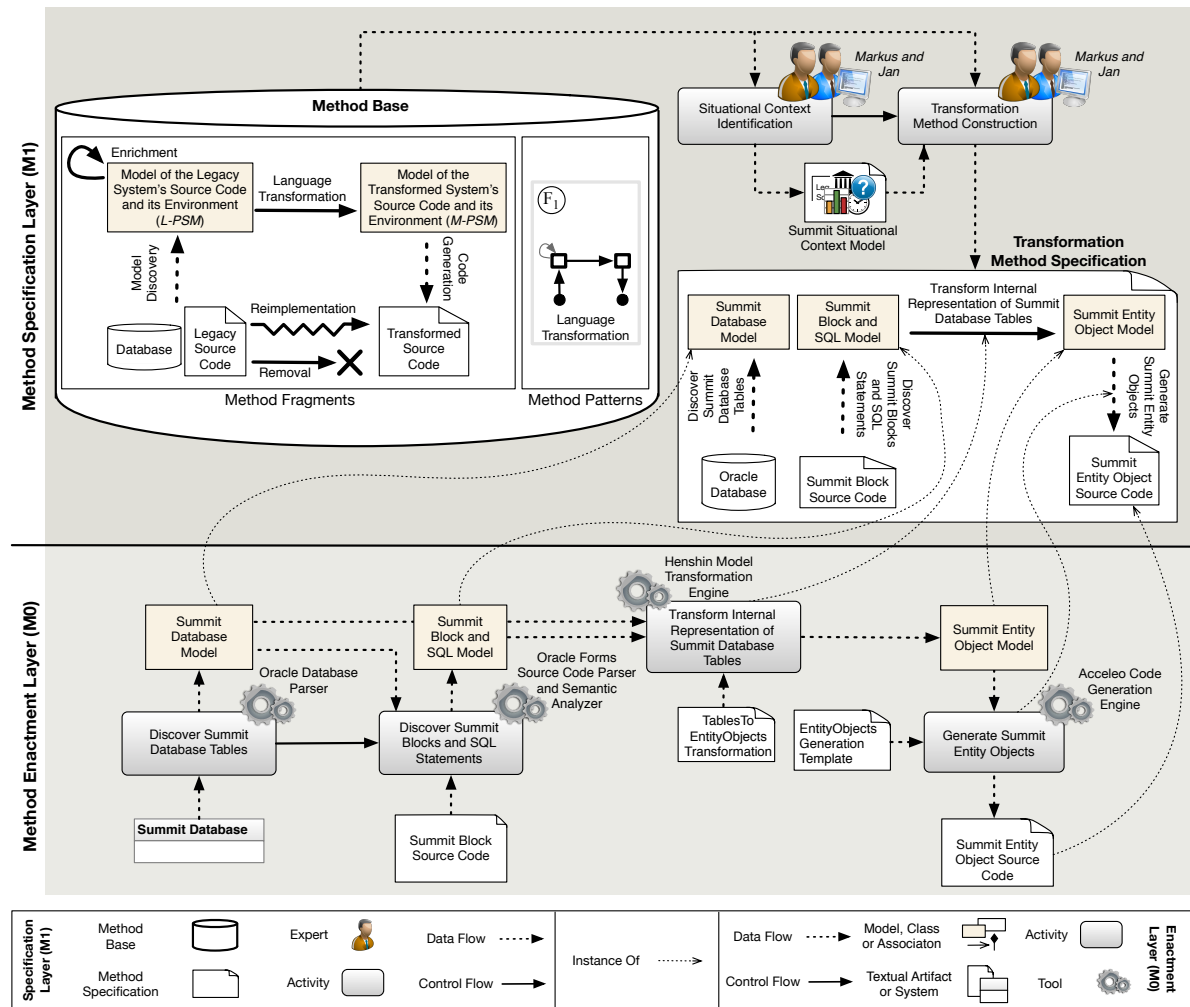


Figure 5.9 Developing a transformation method to transform the internal representation of database tables of the running example by using the *Language Transformation Pattern*

semantics of Blocks and their relation to database tables, it would not have been known that the discovery of the database schema is essential to transform the functionality.

Another purpose of the activity is to perform an assessment of the method patterns stored in the method base, based on the gathered knowledge. This encompasses assessing whether a method pattern is applicable and estimating the effort required to apply it, e.g., the effort required to develop tools like parsers or model transformations. Note that this requires knowledge of the content of the method base. Here, we do not discuss whether the *Language Transformation Pattern* is the best fitting one for the situation observed, but assume that this is the case. The gathered information as well as the result of the assessment is preserved in the form of a *Situational Context Model*.

The activity called *Transformation Method Construction* is performed subsequently. Its purpose is to create a *Transformation Method Specification*. The specification prescribes how to perform the transformation by defining the necessary activities to perform or artifacts to generate. It describes the method explicitly and formally, wherefore it can be used to provide guidance during the enactment of the method.

In MEFiSTo, the construction of a method can be seen as being pattern-based. As a first step, a method pattern gets selected from the method base. In particular, selecting a pattern consists of *choosing* and *configuring* it. Here, we choose the *Language Transformation Pattern* which we configure as needed. On the one hand, configuring a pattern encompasses a decision on variable fragments. As can be seen in Figure 5.6 on page 80, the *Enrichment* fragment is optional for the *Language Transformation Pattern*. In the example, an enrichment activity could have been included to annotate the database tables used by the application. In this case, parsing the Oracle Forms source code would not have been necessary. On the other hand, configuring a pattern encompasses a decision on the amount and type of fragments to instantiate. In the example, two method fragments have been instantiated for the *Model Discovery* fragment, namely *Discover Database Tables* and *Discover Blocks and SQL Statements*. Also, different fragments have been used as input for the discovery activities, namely the *Database* and the *Legacy Source Code* fragment.

As a second step, the instantiated method fragments get customized for the functionality to transform. This encompasses specifying each instantiated fragment in such detail that it provides guidance during the enactment of the method. For example, related to the *Discover Summit Blocks and SQL Statements* fragment it is essential to state which language constructs need to be discovered and how, i.e., Blocks, Items and SQL-statements have to be discovered by a syntactic and semantic analysis. This customization leads to the *Transformation Method Specification* as shown in the figure, which is also the output of the activity.

Based on the steps required to develop the transformation method, it becomes clear that some expert knowledge is required. For example, gathering knowledge of how a functionality is realized in the legacy system requires reengineering skills. These are provided by a person in the role of a modernization expert (in this case, by a person called *Markus*). Another example encompasses the assessment of the effort for tool development, which requires expertise in the field of model-driven engineering. This knowledge is provided by a person in the role of a tool specialist, in this case, by a person called *Jan*.

This concludes the description of an example for the *Language Transformation Pattern*. As this was the first pattern we introduced, we gave a holistic overview of its application. For the next patterns, we leave out the development of the method. Subsequently, we describe generic characteristics of the pattern.

Description

The *Language Transformation Pattern* can be applied to transform the functionality of a legacy system into a new environment, following a conversion-based transformation strategy. The basic idea of how to transform the functionality is to define a mapping between the programming languages of both environments. More precisely, applying the pattern requires representing the functionality to transform on the platform-specific layer as a model. The advantage over a transformation on the system layer is that the functionality and its environment can be described in a homogeneous structure that enables representing relations between them explicitly. Initially, the functionality is represented by a platform-specific model of the legacy environment (L-PSM) that gets created by performing a model discovery. A model transformation transforms the L-PSM into a platform-specific model of the target environment (T-PSM). The model transformation is a realization of the mapping between the programming languages involved. Based on the content of the T-PSM, source code is generated in the target environment.

The transformation strategy described by this pattern is related to the mechanics of a compiler. A compiler is a program that translates a program, written in a source language, to a program, written in a target language [Aho+06, p.1]. In comparison, a transformation method that follows the *Language Transformation Pattern* does exactly that, too. Also, the structure of the resulting method is comparable to the structure of a compiler (cf. Section 2.3.2). The L-PSM is an output of the front-end, while the T-PSM can be seen as an intermediate representation of the middle-end. Then, the transformed source code is an output of the back-end. In this sense, the pattern can be used to describe the mechanics of a compiler.

On the other side, we can also use the pattern to describe methods that do not necessarily follow the principles of a compiler. In particular, one of the core principles of compiler design is to provide a high degree of automation. This becomes clear from a historical perspective, as the process of compiling was originally called *automatic programming* [FCL09, p.2]. However, the transformation method as shown in Figure 5.7 does not primarily focus on automation.

This can be seen when having a look at the proposed mapping between the L-PSM and T-PSM, which requires program comprehension [Rug95]. The mapping defined by the model transformation rules shown in Figure 5.8 aims to identify the use of database tables by references from a Block. This interpretation of the source code is required, in order to perform a mapping to *native* language constructs [TV00] in the target environment, namely to Entity Objects. But, it comes with the drawback that the mapping is not applicable in general. For example, it would not be appropriate for legacy systems that change the data sources of Blocks dynamically at runtime, which is a capability of Oracle Forms. In this sense, applying the transformation method might involve manual effort to develop model transformation rules specifically for the legacy system to transform.

We do not claim that the analysis required to determine which tables are used cannot be realized generically, but aim to point out that this interpretation would not be required to transform the functionality. Instead, a compiler for the programming language Oracle Forms can just translate the language construct of a `Block` into the new environment, possibly emulating it as there is no direct equivalent. However, this is not the mapping that we intended, i.e., the mapping on native language constructs described in [RM11]. In this instance we can see an example for the recurring trade-off between performing a mapping on native language constructs to increase the maintainability or emulating legacy language constructs, potentially increasing the degree of automation [TV00]. While in compiler design the latter characteristic would be favored, the pattern does not make any restrictions. In [Fle+07], it is even stated that the aim of any modernization project shall not be to just create an executable version in the target environment, but to adapt the application to the new environment. Effectively, this can be enabled by the pattern.

An application of this pattern is described in [Fuh+12, pp.174-178]. In this work, a model-driven transformation from Cobol to Java has been performed as a case study. The transformation method described conforms to the *Language Transformation Pattern*. First, a platform-specific model of the Cobol source code is discovered that gets directly transformed to a platform-specific model of the programming language Java. Thereby, different types of model transformation rules are used, namely *customer-specific* and *generic* rules. Based on the Java model, source code is generated in the target environment.

This concludes the generic description of the pattern, whereby we discussed various characteristics. However, so far we did not discuss in which situations an application of the pattern is appropriate. This discussion is part of the next section.

Suitability

We first discuss influence factors on the effectiveness, followed by influence factors on the efficiency. We want to point out that the influence factors are a subjective reflection of our experiences made in industrial projects. As a result, we do, for example, not ensure their completeness. The same applies to the influence factors discussed for subsequent patterns.

In terms on influences on the effectiveness, we need to consider at which point in time, related to a modernization project, a method pattern is chosen. Following established modernization methods, the target design is defined before the transformation method gets developed (cf. Figure 1.3, page 6). Therefore, we consider the defined, i.e., intended way of realizing of a functionality in the target environment as an optimum in terms of effectiveness. In addition, we assume that the *Language Transformation Pattern* can always be used to transform a selected functionality into the desired realization. This might only be possible under certain conditions,

e.g., the model transformation language used needs to provide a sufficient expressiveness, but we assume that these conditions are met. In this case, the method pattern can always be effective, so that it would not be necessary to consider influence factors on the effectiveness.

However, in practice we observed instances in which the application of the pattern led to a deviation from an initially desired realization, even though this negatively influenced the effectiveness. The main reason for a deviation that we observed is the interdependency between the effectiveness and efficiency, i.e., the fact that a decrease in the effectiveness can increase the efficiency. Such an effect can also be seen in the example, shown in Figure 5.7. Assume that we do not create `Entity Objects` for all database tables that are actually used by the application, but for all tables present in the schema. This would decrease the effectiveness, as unused `Entity Objects` get created, which has a negative influence on the performance and the maintainability. However, it does not affect the overall result of the transformation as the functionality of the resulting application does not change. On the other side, it would increase the efficiency, as no syntactic and semantic analysis of the Forms source code would be necessary anymore. Assume that such a tool is not available beforehand and needs to be developed from scratch, then this is an important saving related to time and effort required. Removing the unnecessary `Entity Objects` can be part of a subsequent refactoring project. Due to the interdependencies between efficiency and effectiveness, we consider that the initially desired target realization can change, depending on a method pattern chosen. For that reason, the desired realization needs to be considered as an influence factor on the effectiveness.

In terms on influences on the efficiency, we observed the complexity of the *Language Transformation* activity that transforms the L-PSM into the T-PSM to essentially influence the efficiency of the pattern. The example reveals that the model transformation needs to address at least two concerns. First, the T-PSM needs to be interpreted to identify the functionality to transform. This step requires some form of program comprehension, whose complexity may vary. In the example, the database tables used could be identified reliably by a static analysis of the ASG. As discussed earlier, there exist legacy systems that employ implementations which are harder to interpret. Second, after the functionality has been identified, it needs to be mapped to language constructs of the target environment. Having a look at the structure of database tables and `Entity Objects`, it can be seen that there nearly exists a one-to-one mapping between the structures involved. This does not need to be the case, so that in these cases a mapping gets harder. In summary, the way in which some functionality is realized in the legacy system as well as how it shall be realized in the target environment needs to be considered as an influence factor to assess the complexity of the model transformation.

In general, each method fragment whose use is prescribed by a method pattern needs to be evaluated to identify potential influence factors on its efficiency. For example, the availability of

syntactic and semantic analyzers, code generators and related metamodels are critical influence factors for all method patterns that follow a conversion-based transformation strategy. For these types of patterns, the effort for preparing required tools is not negligible. If this effort outweighs the benefits of automating the transformation, the pattern will not be efficient. This is only the case, if the legacy system or the functionality to transform has a sufficient size [Fle+07]. Therefore, the size of a legacy system constitutes an influence factor on the efficiency.

In conclusion, we assume that the pattern is particularly suitable if the functionality to transform is realized comparably in both environments while the source code to transform is sufficiently large. If the realization of the functionality in both environments differs significantly, using a higher level of abstraction can reduce the complexity of the transformation. The *Conceptual Transformation Pattern* which we introduce subsequently, follows this strategy.

5.5.2 Conceptual Transformation (F_2)

Intent	Perform an automated transformation of the legacy system's functionality into a new environment, following a conversion-based transformation strategy
Strategy	Use an intermediate representation of the functionality to transform on a platform-independent layer. The representation is reverse engineered from an ASG ² of the legacy system on a platform-specific layer. After a potential restructuring, it is transformed into an ASG ² of the target environment, before code gets generated
Structure	

Continued on next page

Applicability	Use when the functionality to transform is realized significantly different in the legacy and target environment and the legacy system has a sufficient size. The use of an intermediate representation can reduce the complexity of the transformation by separating the concerns of reverse engineering, restructuring and mapping the functionality. The complexity of these concerns essentially influences the efficiency and effectiveness of the pattern
Preparation	Applying this pattern essentially requires realizing a parser, model-to-model transformations and code generation rules. In addition, it can be necessary to realize a semantic analyzer, dedicated reverse engineering algorithms or model views
Example	Figure 5.10 shows the enactment of a method which conforms to the pattern. Thereby, the attribute validation rules of the running example are transformed (cf. Section 4.3)
Known Uses	The strategy realized by the pattern is comparable to the strategy followed by a multi-language compiler. However, compiler design usually focuses on automating the transformation. In contrast, the pattern also enables developing semi-automatic methods, which can be necessary to end up with a situation-specific method. The methods described in [Fle+07] and [SSG14] conform to the pattern
Related Patterns	/

Table 5.3 Characterization of the *Conceptual Transformation Pattern*

Example

This section is separated into two parts. First, we describe the (i) background knowledge covering technical details of the example. Then, we describe the (ii) enactment of a method that conforms to the *Conceptual Transformation Pattern*. From this point on, we omit how the method has been developed. An initial idea of this endeavor has been described for the *Language Transformation Pattern* in Section 5.5.1. The development will be revisited in detail in Chapter 6, when the method engineering process of MEFiSTo is described.

²Depending on the situation, using an AST can be sufficient

Technical Background First, we need to have an understanding of the functionality to transform. In this example, we aim to transform another part of the *table-based data access* functionality (cf. Section 4.3). In this instance, we show how the pattern can be used to transform the *attribute validation rules* of the Summit application (C2). The validation rules are used to ensure that no invalid entries are stored persistently in the database.

Second, we need an understanding of how the concept is realized in the legacy system. Oracle Forms provides a set of Triggers that can be used for this purpose, in particular, Triggers whose name begins with WHEN-VALIDATE [Ora00c, p.423]. These Triggers are invoked whenever the internal representation changes, for example due to a user input or a programmatic manipulation. An invocation leads to an execution of the associated, imperative source code that validates the changes made and raises an exception, if the validation fails.

In the example, the WHEN-VALIDATE-RECORD Trigger, which is a part of the S_ORD Block, is such a Trigger. It can be seen in the lower left of Figure 5.10, as well as its contained source code. The purpose of this Trigger is to ensure that the shipping date of ordered goods is later than the ordering date of those goods. Technically, this verification is realized by an If-Statement that compares the value of the DATE_SHIPPED Item to the value of the DATE_ORDERED Item. If the shipping date is earlier than the ordering date, a message is shown to the user (cf. Figure 4.4, page 59) and an exception is raised. The exception would cancel an ongoing commit to the database.

Besides knowledge of the concept and its realization in the legacy system, we thirdly need to have an understanding of how to realize the functionality in the target environment. In Oracle ADF, there exist two ways to realize such validation rules [Ora13]. On the one hand, the rules can be realized imperatively by using Java code. Such code would get invoked by an event, which is raised when an internal representation, i.e., an Entity Object, needs to be validated. On the other hand, Oracle ADF enables defining validation rules declaratively. For this purpose, a set of language constructs is provided that can be used to express the rules as part of the definition of Entity Objects.

While the imperative realization comes close to how the functionality is currently realized in the legacy system, the declarative realization is favored by the mapping described in [RM11, pp.15-16]. This is due to the fact that Oracle suggests to use the declarative realization if possible, as it is associated with various benefits³. For example, declarative rules are executed by a dedicated framework which takes care of stacking exceptions. Imperative realizations shall only be used in complex scenarios that cannot be expressed declaratively. Therefore, the challenge is to transform an imperative realization into a declarative one (cf. Section 2.3.3).

³<http://docs.oracle.com/middleware/1213/adf/develop/adf-bc-validation-rules.htm> (accessed March 22th, 2016)

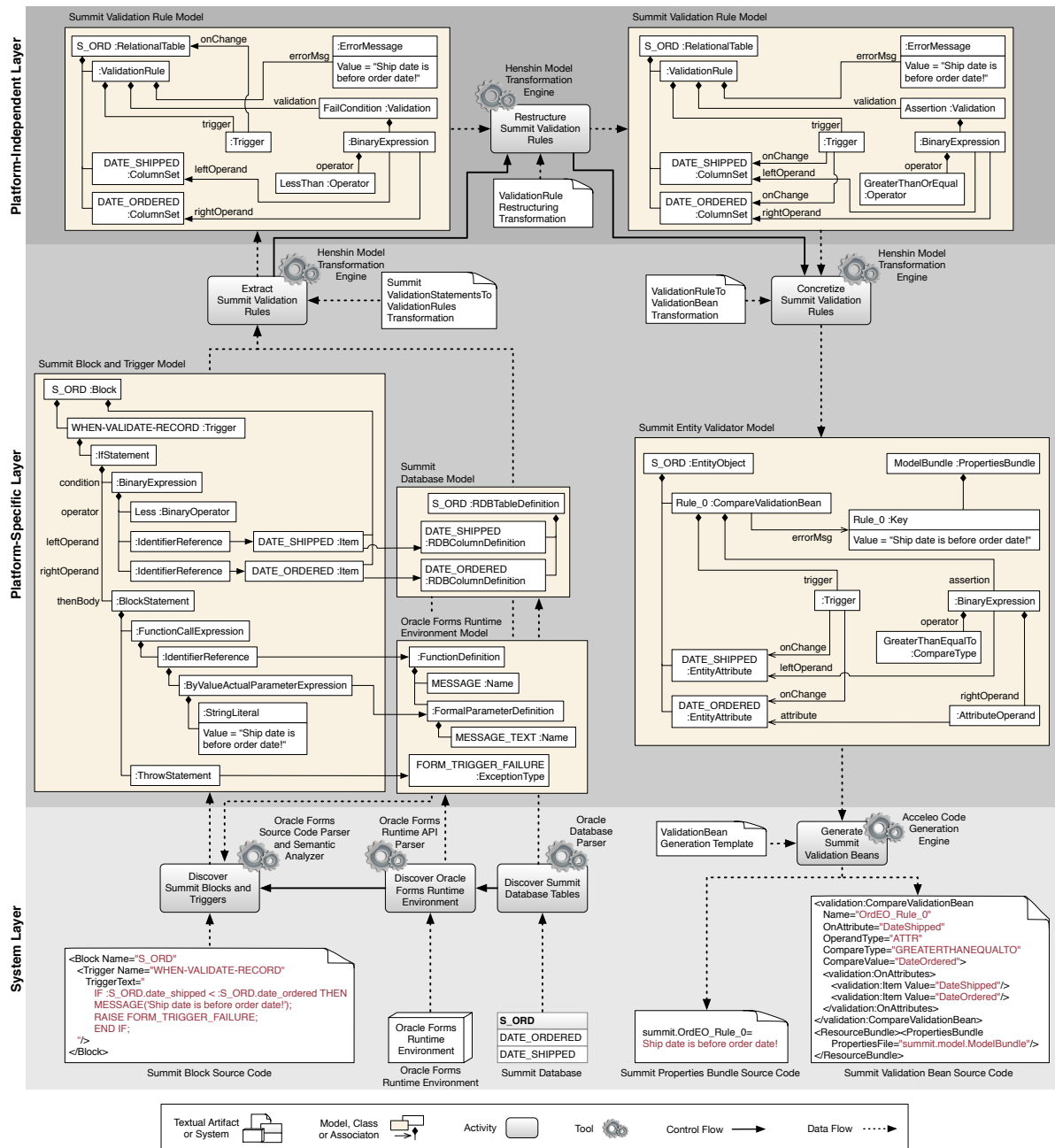


Figure 5.10 Enacting a transformation method to transform the attribute validation rules of the running example by using the *Conceptual Transformation Pattern*

Method Enactment The desired transformation is achieved by enacting the transformation method as shown in Figure 5.10. Note that most of the artifacts are presented shortened. The method consists of customized method fragments that have been introduced in Section 5.3 and is an instance of the *Conceptual Transformation Pattern*.

In the method shown, the *Model Discovery* fragment has been customized three times to gather information associated with the functionality and represent it by an L-PSM. The activity called *Discover Summit Database Tables* is the same as the one described for the *Language Transformation Pattern* in Section 5.5.1. It is performed automatically by an *Oracle Database Parser* to parse the schema of the database used by the Summit application and create a corresponding model. Related to the Summit application, the discovery of the table S_ORD is shown in the lower center of Figure 5.10. Their parsing results in the *Summit Database Model* which conforms to the GASTM defined in [OMG11a, pp.115-123].

The activity called *Discover Oracle Forms Runtime Environment* is performed automatically using an *Oracle Forms Runtime API Parser*. Its purpose is to represent the runtime environment of Oracle Forms by modeling its Application Programming Interface (API). In this example, this comprises defined functions as well as exceptions. A textual description of the API could be a conceivable input for this activity. However, the method specifies to use a management interface of the runtime environment itself. As the interface is Java-based, the API can be extracted using the reflection capabilities of Java. The resulting *Oracle Forms Runtime Environment Model* conforms to the GASTM defined [OMG11a].

The activity called *Discover Summit Blocks and Triggers* is also performed automatically, but using an *Oracle Forms Source Code Parser and Semantic Analyzer*. Performing the activity requires parsing the source code of the Summit application, in particular, the Blocks and Triggers as well as their related properties. Related to the Summit application, the discovery of the S_ORD Block and the contained WHEN-VALIDATE-RECORD Trigger is shown in the lower left of Figure 5.10. The resulting *Summit Block and Trigger Model* represents these language constructs and, more importantly, the PL/SQL source code that is defined within the Trigger Text property. The model conforms to an SASTM [OMG11a] that has been defined for the programming language of Oracle Forms. Note that there exist edges between the *Summit Block and Trigger Model* and the *Oracle Forms Runtime Environment Model*, e.g., modeling the call of a function provided by the runtime environment. These edges are semantic edges that result due to a semantic analysis, making the model an ASG (cf. Section 2.3.2).

After the L-PSM has been created, it gets transformed into a Functional Platform-Independent Model (F-PIM) by performing the *Extract Summit Validation Rules* activity. This activity is a customization of the *Program Comprehension* fragment and prescribes to perform reverse engineering. In particular, it prescribes to abstract from platform-specific language constructs by interpreting the L-PSM and representing the result on a platform-independent layer. Here, this is realized by executing a set of model transformation rules by a corresponding engine. At this point, we do not go into detail on the rules or how to develop them. The challenge of this transformation is discussed at the end of this section.

Related to the Summit application, the reverse engineering of the rule that validates the shipping and ordering dates is shown. The ASG that is part of the *Summit Block and Trigger Model* represents an imperative realization of that rule in the programming language PL/SQL. It can be seen that the rule is only described implicitly, i.e., specific knowledge is required to understand that this part of the source code is, in fact, a validation rule. Take for example the `String Literal` that is a parameter of the function call to the platform-specific function called `MESSAGE`. Just having knowledge of the semantics of the language constructs itself that are associated with the function call would not enable to understand the meaning of the literal. In this case, the key is on the one hand to know the semantics of the `WHEN-VALIDATE-RECORD` Trigger and on the other hand the structure of a validation rule. Then, interpreting the literal in this context reveals that it represents the error message for the case that the validation fails.

The result of such an interpretation can be seen in the *Summit Validation Rule Model*, shown in the upper left of Figure 5.10. The model conforms to the KDM [OMG11b] that has partly been extended to represent validation rules. The validation rule of the example is represented by the equally named class `Validation Rule`. It can be seen that the concept of a validation rule is associated with three characteristics, represented as classes: a `Trigger` that determines when to evaluate the rule, a `Validation` that represents the condition to validate as well as an `Error Message` that shall be displayed in case that the validation fails.

After the F-PIM has been created, its structure is changed by performing the activity called *Restructure Summit Validation Rules*. The activity is a customization of the *Restructuring* fragment and prescribes to perform changes on the F-PIM by an endogenous model transformation, i.e., the metamodel does not change. In terms of the *Conceptual Transformation Pattern*, this is an optional activity. Related to the Summit application, the result of the restructuring can be seen in the upper right of Figure 5.10. In the example, this intermediate activity is performed to bring the structure of the validation rule that has been reverse engineered from the legacy system closer to the desired structure in the target environment. In particular, two differences are addressed by the restructuring.

First, the `Trigger` that determines when to evaluate the rule, is changed. In the legacy system, the rule is part of the `WHEN-VALIDATE-RECORD` Trigger so that it is always evaluated whenever any of the `Items` of the `S_ORD` Block changes. This means that it is also evaluated even when neither the `DATE_SHIPPED` nor the `DATE_ORDERED` changes. In this case, a triggering results in unnecessary computational effort, negatively influencing the performance. An unnecessary triggering of the rule could have been prevented by using `Item-based Triggers`, i.e., `WHEN-VALIDATE-ITEM` Triggers for the `DATE_SHIPPED` and the `DATE_ORDERED` Items. In this case, the evaluation would have only been triggered whenever these specific Items change. However, this would require duplicating the code that contains the validation rule,

negatively influencing the maintainability. In other words, the restrictions in the legacy environment require making cuts in the performance or maintainability, as one validation rule cannot be related to multiple attributes, i.e., *Items*. In the target environment, this restriction is not present anymore. Therefore, the restructuring addresses the *onChange* association of the *Trigger* so that it refers to both attributes (represented as *ColumnSets*) afterwards.

Second, the *Validation* that represents the condition to validate, is changed. In the legacy system, the condition is realized by an expression that checks for a failure. If the condition `DATE_SHIPPED < DATE_ORDERED` yields `TRUE`, then the rule is violated. In the target environment, the opposite needs to be expressed. Here, we do not need to (imperatively) check for a failure, but (declaratively) specify the assertion that needs to hold (cf. Section 2.3.3). In this example, the resulting assertion would be `DATE_SHIPPED ≥ DATE_ORDERED`. Therefore, the restructuring addresses the *Validation* and *Operator* class by changing the type of the former and negating the latter.

After the F-PIM has been restructured, it gets transformed into a T-PSM by performing the activity called *Concretize Summit Validation Rules*. The activity is a customization of the *Concretization* fragment and prescribes to perform forward engineering. In particular, it prescribes to express the functionality represented on a platform-independent layer by platform-dependent language constructs. Related to the Summit application, the result of the concretization can be seen in the center right of Figure 5.10. The *Summit Entity Validator Model* conforms to an SASTM [OMG11b] that has been defined for the programming language of Oracle ADF. It can be seen that for most classes a one-to-one mapping exists. For example, a *Validation Rule* is mapped to a *Compare Validation Bean*, whereby the latter represents a platform-specific construct. For some classes, a more complex mapping is required. For example, the *Error Message* is stored in separate properties file and only referenced by the validation rule. The intention is to be able to easily exchange the properties file later on, depending of the language chosen by the user.

As a last step, code is generated based on the T-PSM by performing the activity called *Generate Summit Validation Beans*. As this activity does not differ from the one described for the *Language Transformation Pattern*, we omit a description and refer to Section 5.5.1.

This concludes the description of an example for the *Conceptual Transformation Pattern*. Subsequently, we describe its generic characteristics.

Description

The *Conceptual Transformation Pattern* can be applied to transform the functionality of a legacy system into a new environment, following a conversion-based transformation strategy. The basic idea of how to transform the functionality is to explicitly represent it on a platform-

independent layer. This representation is extracted from the source code of the legacy system and mapped to source code in the target environment. More precisely, applying the pattern first requires representing the functionality to transform on the platform-specific layer as a platform-specific model of the legacy environment (L-PSM) by performing a model discovery. Then, the functionality to transform gets reverse engineered from the L-PSM and represented explicitly by a platform-independent model (F-PIM). By performing a concretization, the F-PIM is transformed into a platform-specific model of the target environment (T-PSM), which is subsequently used to generate source code.

Just like the *Language Transformation Pattern*, the transformation strategy described by the *Conceptual Transformation Pattern* is related to the mechanics of a compiler, too. This is due to the fact that a transformation method which follows one of both patterns, enables transforming functionality, written in a source language, to functionality, written in a target language. The main difference between both patterns lies in the intermediate representations used, i.e., the structure of the middle-end (cf. Section 2.3.2). In the case of the *Conceptual Transformation Pattern*, an additional intermediate representation on a platform-independent layer is used. For a compiler, the use of a platform-independent intermediate representation is not unusual, especially, if the compiler supports multiple source and/or target languages [FCL09, p.395-396]. The representation then enables the exchange of the front- or back-end, so that various combinations of source and target languages can be supported, while the middle-end can be reused. The same motivation led to the definition of the ASTM and KDM metamodels by the OMG. The use of standardized, platform-independent metamodels shall facilitate reusing of tools [UN10, pp.45-48]. In this sense, the pattern can be used to describe the mechanics of a compiler, too.

On the other side, we see another important motivation for using a platform-independent intermediate representation, namely facilitating a separation on concerns. For the *Language Transformation Pattern*, we discussed that the model transformation which transforms the L-PSM into the T-PSM needs to fulfill at least two concerns. First, the ASG needs to be interpreted to identify the functionality to transform. Second, the functionality needs to be mapped to language constructs of the target environment. Additionally, as demonstrated by the example, it can be necessary to restructure the functionality. When applying the *Conceptual Transformation Pattern*, these concerns are separated. First, the *Program Comprehension* activity addresses the interpretation. Then the functionality is *Restructured* before the *Concretization* activity addresses the subsequent mapping in the target environment.

Based on our experience, this separation of concerns is especially beneficial if the functionality to transform is only implicitly described by the source code of the legacy system, i.e., an increased degree of program comprehension is required. In the example shown in Figure 5.10,

interpreting the *If Statement* contained in the *Trigger* as a validation rule and identifying the different parts of the rule results in a complex mapping between the syntactic elements involved. Like for the *Language Transformation Pattern*, this can come at the expense of a decreased degree of automatism, but an increased degree of non-functional properties by enabling a mapping on native language constructs in the resulting system.

An application of this pattern is described in [Fle+07]. In this work, an industrial application has been transformed from COOL:Gen to Cobol, using a model-driven transformation method. The method conforms to the *Conceptual Transformation Pattern*. First, a platform-specific model of the COOL:Gen source code is discovered. Then, this model is transformed into an intermediate representation that conforms to a self-defined, platform-independent metamodel called ANT. Based on this representation some restructurings are performed, before finally source code is generated in the target environment.

Another example of a transformation method that conforms to the pattern is described in [SSG14]. In this work, graphical user interfaces developed in Oracle Forms and Borland Delphi are automatically converted to web pages. The transformation method uses platform-independent models of these user interfaces, namely Rapid Application Development (RAD) and Concrete User Interface (CUI) models as intermediate representations.

This concludes the generic description of the pattern, whereby we discussed various characteristics. However, so far we did not discuss in which situations an application of the pattern is appropriate. This discussion is part of the next section.

Suitability

In terms on influences on the effectiveness, the desired target realization needs to be considered as an influence factor. Thereby, the argumentation is the same as for the *Language Transformation Pattern*: a deviation from the desired realization can reduce the complexity of activities. In the example, using an imperative realization in the target environment would ease the reverse engineering and make the restructuring unnecessary, therefore increasing the efficiency of the method. However, this comes to the expense of a decreased performance and maintainability, as the validation framework and the IDE support of the target environment could not be used.

We want to point out that this argument applies for all patterns, also for the ones that are described subsequently. Therefore, we do not mention this factor for subsequent patterns. In general, we do not discuss factors in detail that have already been discussed for other patterns.


In terms on influences on the efficiency, we observed the complexity of the program comprehension activity to essentially influence the efficiency of the pattern. For the example, we did not provide any details of how the transformation between the L-PSM and F-PIM is realized but assume that this activity describes the execution of a model transformation.

This is, for example, possible if code conventions exist that can be used to reliably extract the required information. However, often the reverse engineering activity consists of multiple steps, i.e., it is a process itself. For example, in [Cos+12] a framework is described to extract business rules from Java source code. The reverse engineering activity is performed semi-automatically, consisting of multiple steps and intermediate results. Therefore, the way in which some functionality is realized in the legacy system as well as how it shall be realized in the target environment needs to be considered as an influence factor to assess the complexity of the program comprehension activity. Nevertheless, we assume that this pattern is particularly efficient if the functionality to transform is realized in different ways in both environments.

Apart from that, as discussed before, each method fragment that gets used when applying the method pattern needs to be evaluated to identify potential influence factors on its efficiency. As this pattern follows a conversion strategy, too, the size of the legacy system is an essential influence factor on its efficiency.

In conclusion, we assume that the pattern is particularly suitable if the functionality to transform is realized significantly different in both environments while the source code to transform is sufficiently large. If the source code is not that large, the automation of the transformation might be inefficient. Then, the method pattern we introduce in the next section can be a viable alternative, as it follows a *Reimplementation*-based transformation strategy.

5.5.3 Reimplementation (F_3)

Intent	Perform a manual transformation of the legacy system's functionality into a new environment, following a reimplementation-based transformation strategy
Strategy	Provide guidance for software developers who manually reimplement the functionality in the target environment
Structure	
Applicability	Use when automatic approaches are either inefficient or ineffective. The amount of available developers and their experience has an essential influence on the efficiency and effectiveness of the pattern

Continued on next page

Preparation	Applying this pattern essentially requires defining guidance documents to systematize the reimplementation.
Example	Figure 5.11 shows the enactment of a method which conforms to the pattern. Thereby, the internal representation of database views of the running example is transformed (cf. Section 4.3)
Known Uses	The strategy realized by the pattern is comparable to the implementation activity that is part of a software development endeavor. It can be seen as a specific type of implementation activity as some constraints need to be considered: the legacy system specifies the functionality to realize while guidance documents describe performed design decisions or required restructurings. The method described in [RM11] conforms to the pattern
Related Patterns	/

Table 5.4 Characterization of the *Reimplementation Pattern*

Example

This section is separated into two parts. First, we describe the (i) background knowledge covering technical details of the example. Then, we describe the (ii) enactment of a method that conforms to the *Reimplementation Pattern*.

Technical Background First, we need to have an understanding of the functionality to transform. In this example, we aim to transform the *view-based data access* functionality of the Summit application (cf. Section 4.3) as a whole. Primarily, this requires the transformation of the *internal representation of the views* (C3), i.e., its structure. Thereby, most attributes of a view are based on fields inside a database. However, in the Summit application, some attributes are *calculated* (C4) dynamically by an expression. We aim to transform these expressions, too. Lastly, views can be related to each other by *view relations* (C5), which cause that a selected dataset of one view determines the datasets of another view.

Second, we need an understanding of how the concept is realized in the legacy system. As described for the *Language Transformation Pattern* in Section 5.5.1, data Blocks are used within an Oracle Forms application to specify views. Usually, most Items contained in such a Block relate to an underlying field in a database table. For Items whose value is calculated

dynamically by an expression, the property `Calculation Mode` is set to `Formula`, while the `Formula` property contains the actual expression. To specify relations between views, the `Relation` language construct can be used which is also contained within a `Block`. It has several properties to specify the relation, like the `Join Condition` property.

Related to the Summit application, the realization of two views can be seen in the lower left of Figure 5.11. The `Block` called `S_ORD` defines a dialog-specific view to access ordering information, while the `Block` called `S_ITEM` defines a view on the items of a selected order. This dependency is realized by the `Relation` called `S_ORD_S_ITEM`, contained in the `S_ORD` `Block`. It defines a join-condition that is evaluated at runtime. In particular, the join is performed on the `ID Item` of the `S_ORD` `Block` and the `ORDER_ID Item` of the `S_ITEM` `Block`. In addition, the `ITEM_TOTAL` attribute that is part of the `S_ITEM` `Block` is not associated with a field in the database, but it is calculated. It represents the total costs of an ordered item, by multiplying the cost of one item with the amount ordered (cf. Figure 4.4, page 59).

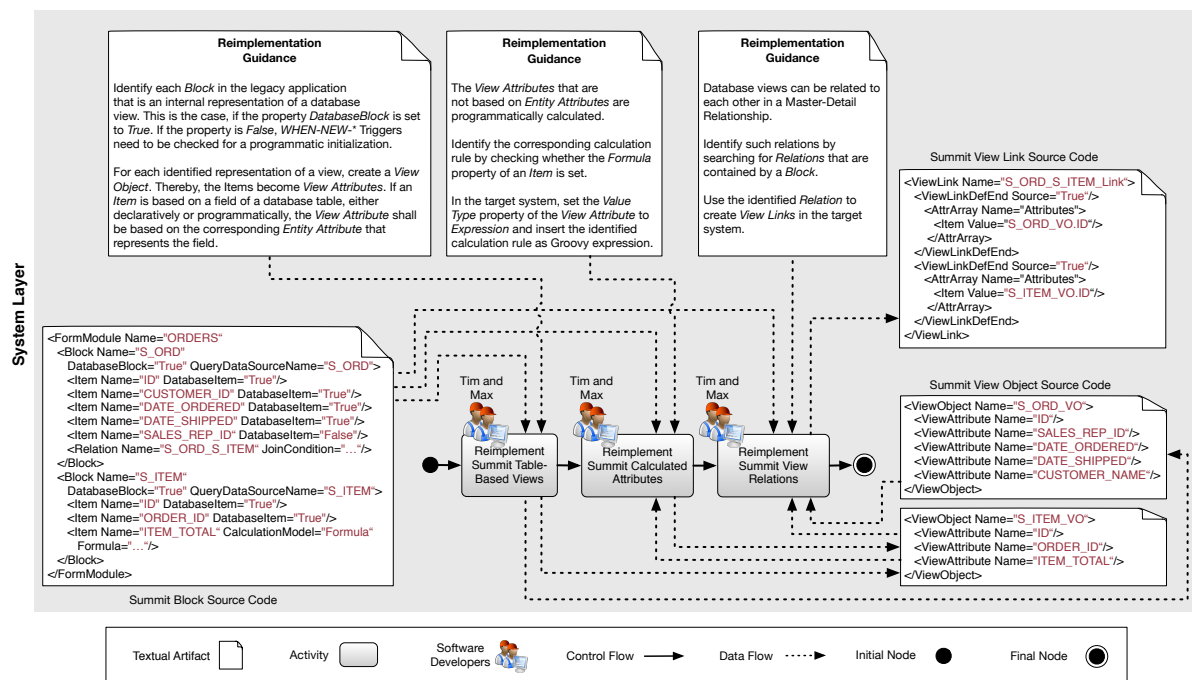


Figure 5.11 Enacting a transformation method to transform the view-based data access of the running example by using the *Reimplementation Pattern*

Besides knowledge of the concept and its realization in the legacy system, we thirdly need to have an understanding of how to realize the functionality in the target environment. Here, we will use the mapping described in [RM11, p.10-11,23]. In this case study, `View Objects` are used as an internal representation of database views. In Oracle ADF, there exist different ways on how to define `View Objects`. For example, they can be directly based on an

SQL-Query. But, ADF also provides a way to declaratively define database views, by using previously defined Entity-Objects. As discussed for the *Language Transformation Pattern* in Section 5.5.1, an Entity Object is a representation of a database table. Entity-Based View Objects are those, whose View Attributes correspond to Entity Attributes, enabling a declarative definition of database views. Apart from that, View Attributes can be based on expressions, too. The relation between View Objects is realized by so-called View Links. They enable to declaratively define the dependency between View Attributes.

On the one hand, transforming the internal representation of the database views, i.e., creating View Objects based on Blocks is not very complex. This is due to the fact that both language constructs are comparable in their structure, just like Entity Objects and Blocks (cf. Section 5.5.1). On the other hand, the transformation also comprises transforming an imperative realization of the relation between two views into a declarative one, comparable to the transformation described for the *Conceptual Transformation Pattern* (cf. Section 5.5.2). Overall, the degree of complexity when transforming the functionality as described, is varying.

Method Enactment The desired transformation is achieved by enacting the transformation method as shown in Figure 5.11. Note that most of the artifacts are presented shortened. The method consists of customized method fragments that have been introduced in Section 5.3 and is an instance of the *Reimplementation Pattern*.

In the method shown, the *Reimplementation* fragment has been customized three times. Each resulting activity is performed by assigned software developers, while associated *Reimplementation Guidance* artifacts provide instructions on what to do.

The activity called *Reimplement Summit Table-Based Views* is performed to transform the internal representation of database views contained in the legacy system into the target environment. Details on how to perform this are described in the associated reimplementation guidance, it acts as a step-by-step instruction for a developer. First, a developer shall identify the views by searching for data Blocks in the legacy system. For each instance found, a View Object with View Attributes shall be created. Related to the Summit application, the transformation of the S_ORD and S_ITEM Block is shown, both representing dialog-specific database views.

The activity called *Reimplement Summit Calculated Attributes* is performed to transform attributes of a view that are not based on fields in a database but calculated dynamically. The associated reimplementation guidance describes how to find the expression in the legacy system and where to reimplement it in the target environment. This requires transforming an expression that is written in the programming language PL/SQL into Groovy. Related to the Summit

application, the ITEM_TOTAL attribute is calculated dynamically, but its transformation is not shown in detail.

The activity called *Reimplement Summit View Relations* is performed to transform the relations between views. In particular, the reimplementation guidance describes to identify Relation constructs as part of a Block and create corresponding View Links. Related to the Summit application, the relation between the S_ORD and S_ITEM Block is shown.

This concludes the description of an example for the *Reimplementation Pattern*. Subsequently, we describe its generic characteristics.

Description

The *Reimplementation Pattern* can be applied to transform the functionality of a legacy system into a new environment, following a reimplementation-based transformation strategy. The basic idea of how to transform the functionality is to guide developers that manually develop the functionality in the target environment. More precisely, applying the pattern requires specifying the activities to be performed by developers. Thereby, the activities are associated with instructions written in a natural language. The transformation itself is performed on the system layer, i.e., developers read the source code of the legacy system and directly develop source code in the target environment.

The transformation strategy described by this pattern can be related to the process of developing software from scratch. Each software development endeavor covers some essential activities [Som10, p.28]. First, the functionality of the software to developed is *specified*, before it gets *designed* and *implemented*. The details of these activities depend on the specific method used, e.g., which artifacts result from each activity. However, in general, the implementation activity is concerned with realizing the software as specified, considering the design decisions made [Som10, p.177].

In comparison, the reimplementation activity that needs to be customized when applying the *Reimplementation Pattern* can be seen as some kind of implementation activity. As an example, consider the transformation method shown in Figure 5.11. The excerpt of the source code that is used as an input of the reimplementation activity can be seen as a way to specify the functionality to realize. The additional input in the form of guidance documents specifies the design decisions to consider, i.e., design decisions related to the mapping in the target environment. However, this analogy is not directly applicable in each situation. Basically, it depends on the differences between the legacy system and the resulting system. A set of potential differences based on [Fuh+12, p.156] is shown in Figure 5.12.

In the example, the instructions that are provided as guidance only consider a difference on the implementation level (D_I), i.e., they only describe how to map the source code into

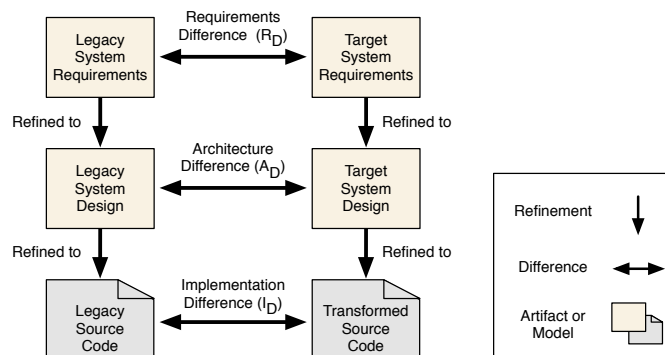


Figure 5.12 Potential differences between a legacy system and a target system, based on [Fuh+12, p.156]

the target environment. However, there might be instances in which the architecture (D_A) or even the requirements (D_R) change. In this case, such changes need to be guided, too. Related to the example, consider that we do not simply create *View Objects* per *Form Module* but per *component*, which results by *merging* multiple *Form Modules*. In this case, the merging operation, i.e., the architectural restructuring, needs to be incorporated into the guidance documents, too. In general, it can be necessary to extend or overwrite the specification of the functionality provided by the source code of the legacy system by the provided guidance.

An application of this pattern is described in [RM11] which describes the manual redevelopment of the Summit application in Oracle ADF. For each functionality of Summit, it provides a description on how to map it into the new environment. Note that we used this work throughout this chapter to derive the mapping for the running example.

This concludes the generic description of the pattern, whereby we discussed various characteristics. However, so far we did not discuss in which situations an application of the pattern is appropriate. This discussion is part of the next section.

Suitability

In terms on influences on the effectiveness, the skills of the software developers that perform the reimplementation activities need to be considered as an influence factor. This is due to the fact that the developers will always have a certain degree of freedom when implementing some functionality [Som10, p.177]. While a conversion-based transformation formally specifies the result in the form of tools, e.g., by model transformations, developers can perform micro design decisions for design issues that have not been addressed by the guidance provided. This is especially critical, if the developers are not familiar with the target environment [GW05].

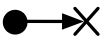
Also in terms on influences on the efficiency, the software developer plays a central role. On the one hand, the amount of available developers as part of a modernization project need to

be considered. This factor has an influence on the efficiency, in particular, it determines the speed of the transformation. On the other hand, the individual skills also have an influence on the efficiency. If a software developer is not familiar with the target environment, it might take him longer to perform the reimplementation activities. In the example, we did not instruct how to transform the calculation rule for calculated attributes from PL/SQL to Groovy. We assume that a developer knows how to do this or is able to learn it, which can take additional time.

The effort for preparing the guidance documents is the equivalence to the effort required to develop tools when applying a conversion-based method pattern. While the detail of the guidance provided can vary, depending on skills of the developers, there is an effort associated with creating it. This effort needs to be considered as an influence factor. In addition, the size of the legacy system needs to be considered, too. As discussed in [Fle+07], there exists a threshold that depends on the size of the legacy system, i.e., the size determines whether an automatic or a manual transformation is more efficient. If the legacy system is sufficiently large, an automatic transformation will be the more efficient transformation strategy.

In conclusion, we assume that the pattern is particularly suitable if the developers are familiar with the target environment and the code to transform is not too large. In some cases, it might not be necessary to even transform the source code to preserve the functionality. Such an instance and the related *Code Removal Pattern* is introduced in the next section.

5.5.4 Code Removal (F₄)

Intent	Perform no transformation of some functionality described in the legacy system to preserve it
Strategy	Ignore corresponding source code of the legacy system
Structure	
Applicability	Use when the functionality is not used by the legacy system (dead code) or if it is implicitly provided by the target environment
Preparation	/

Continued on next page

Example	Figure 5.13 shows the enactment of a method to transform the tree navigator of database views of the running example (cf. Section 4.3), which conforms to the pattern
Known Uses	The strategy realized by the pattern is comparable to the dead code elimination performed by a compiler as an optimization
Related Patterns	/

Table 5.5 Characterization of the *Code Removal Pattern*

Example

This section is separated into two parts. First, we describe the (i) background knowledge covering technical details of the example. Then, we describe the (ii) enactment of a method that conforms to the *Code Removal Pattern*.

Technical Background First, we need to have an understanding of the functionality to transform. We exemplify the *Code Removal Pattern* by transforming the *tree-based data selection* functionality of the Summit application (cf. Section 4.3). On the one hand, we show how to transform the *user interface elements* (C6) that graphically visualize a data structure as a tree. And, more importantly, we transform the functionality that enables *expanding and collapsing* (C7) parts of the tree.

Second, we need an understanding of how the concept is realized in the legacy system. In Oracle Forms, the user interface element of a tree is internally defined by an Item of the type Hierarchical Tree [Ora00c, p.59]. The type is indicated by a corresponding Item Type property. Items that represent user interface elements are assigned to a Canvas [Ora00a, p.66]. A Canvas constitutes a background on which graphical elements get drawn. The expand and collapse behavior is realized by imperative PL/SQL code. When the code is triggered, the tree gets traversed whereby the state of the nodes get changed.

Related to the Summit application the realization of the tree to select a customer can be seen in the lower left of Figure 5.13. The Item named TREE_CUST of the Block called NAVIGATOR represents the user interface element for the tree. It is placed on a Canvas called TREE. The Block called NAV_CONTROL contains four Items that constitute buttons on the same Canvas, they can be used to trigger the expand and collapse behavior. The behavior is realized at two

places: In the Trigger that is contained by the *Items*, as well as in dedicated Program Units, namely COLLAPSE_ALL and EXPAND_ALL.

Besides knowledge of the concept and its realization in the legacy system, we thirdly need to have an understanding of how to realize the functionality in the target environment. Like Oracle Forms, Oracle ADF provides a language construct to declaratively define the graphical representation of a tree, simply called Tree [Ora13]. It gets placed on a Facet, which corresponds to a Canvas element in Forms.

The distinctiveness of this mapping lies in the mapping of the expand and collapse behavior. In the scenario described, it is not necessary to transform the source code that realizes the behavior, as it is already implicitly provided by the Tree construct. In particular, a context menu is provided at runtime that enables expanding and collapsing the tree automatically. Whether this menu is provided or not can be controlled by setting the associated property called `expandAllEnabled` to true or false. However, as it is enabled by default, this property is not visible on the system layer, i.e., it is not part of the source code.

Method Enactment The desired transformation is achieved by enacting the transformation method as shown in Figure 5.13. Thereby we want to point out that the method does not conform to the *Code Removal Pattern*. Instead, the combined use of two method patterns is shown: The user interface elements are transformed by applying the *Language Transformation Pattern* (cf. Section 5.5.1), while the expand and collapse behavior is transformed by applying the *Code Removal Pattern*. We do not solely show a method that applies to the *Code Removal Pattern* for two reasons. On the one hand, the application of the pattern can be observed best when additionally describing how another functionality is transformed. On the other hand, it is an example for a composed pattern, i.e., a method pattern that arises when combining multiple basic patterns. These types of patterns are introduced in the next section.

Related to the transformation of the user interface elements we do not provide a detailed description of the method parts but refer to the example provided in Section 5.5.1. First, a syntactic and semantic analysis is performed by enacting the activity called *Discover Summit Blocks and Canvases* to retrieve the ASG of the source code. Then, by performing the activity called *Transform Summit Canvases and UI Elements*, a model transformation is executed that transforms the L-PSM into the T-PSM. In particular, the *Item* that constitutes the tree is transformed to a *Tree*, while the *Canvas* is transformed to a *Facet*. As a last step, performing the activity called *Generate Summit User Interfaces* generates corresponding source code for the target environment.

The *Code Removal Pattern* is special in the sense that it does only prescribe to customize one method fragment, namely the *Legacy Source Code* fragment (cf. Figure 5.6). This is due

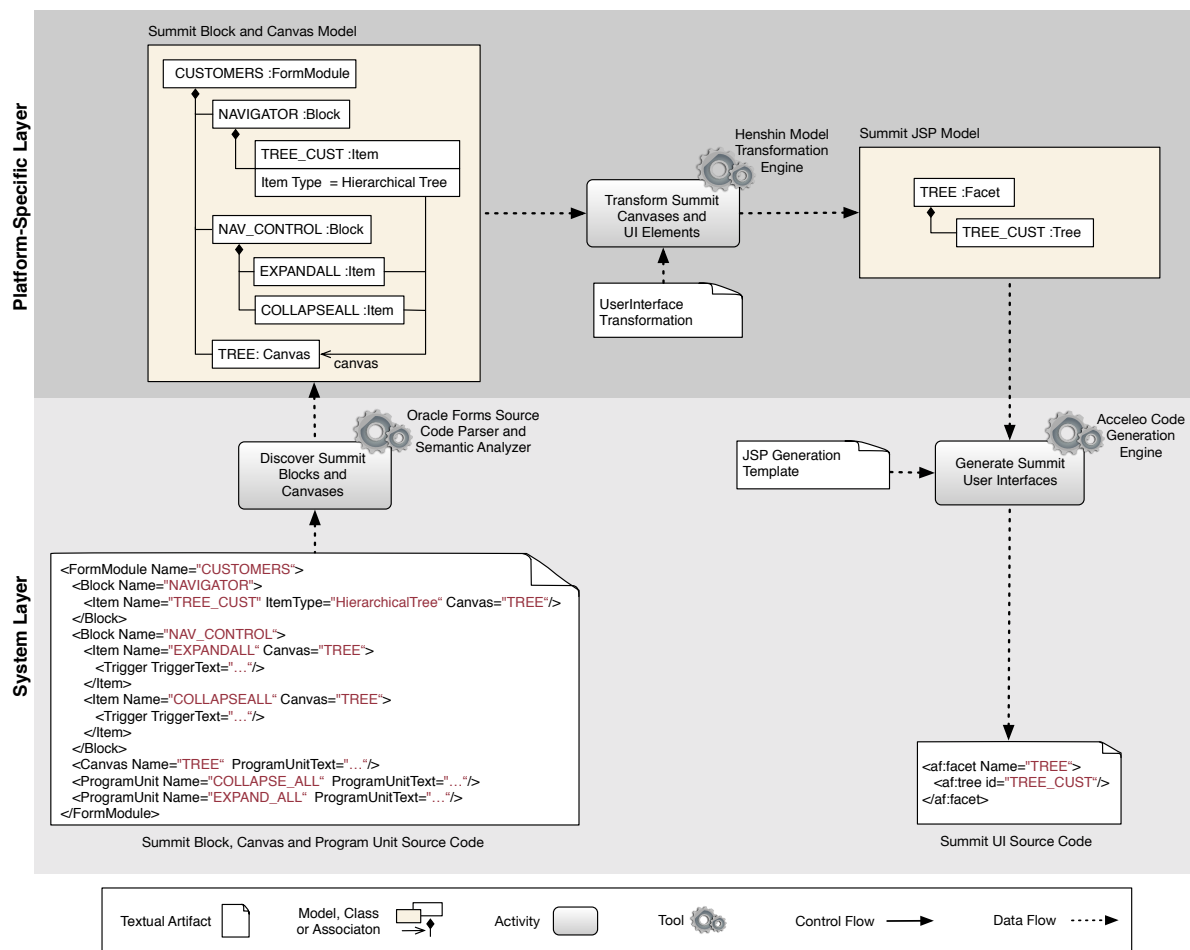


Figure 5.13 Enacting a transformation method to transform the tree-based data selection of the running example by using the *Code Removal Pattern*

to the fact that the affected parts of the source code do not need to be considered during the transformation. Therefore, the *Code Removal Pattern* can only be seen implicitly in Figure 5.13. It is applied on those parts of the source code that are not transformed. In this example, this comprises the Triggers contained by the Items called COLLAPSEALL and EXPANDALL, as well as related Program Units.

This concludes the description of an example for the *Code Removal Pattern*. Subsequently, we describe the generic characteristics of this method pattern.

Description

The *Code Removal Pattern* can be applied to preserve the functionality of a legacy system by not transforming it. This is achieved by ignoring corresponding parts in the source code that represent the targeted functionality.

The transformation strategy described by this pattern is related to the mechanics of a compiler. In particular, we refer to a common optimization performed by compilers, i.e., the *elimination of dead code* [Aho+06, pp.591-592]. In this step, a compiler tries to identify parts of the source code that cannot be executed and are therefore not compiled. In certain instances this can be deduced by considering knowledge gained during the compilation, i.e., the use or the value of variables.

Related to the development of transformation methods we need to essentially distinguish two cases, i.e., we need to distinguish for which reason it is not necessary to transform the code. On the one hand, the code might *not be used* in the legacy system, which can result when a system got continuously advanced. In fact, the presence of dead code is a common problem for legacy systems, wherefore it is usually addressed in a dedicated preparation phase [SWH10, pp. 106-107]. If some code gets removed during the transformation for that reason, then it is comparable to the *elimination of dead code* that we described.

On the other hand, the code might *not be necessary* in the target environment, as the environment provides the functionality implicitly. An example for this type of code can be seen in Figure 5.13. As the target environment already provides the functionality to expand and collapse a tree, the source code that realizes this functionality imperatively needs not to be transformed. One could argue that the functionality is defined by the `Tree` construct so that the language constructs, e.g., the `Program Units`, are not removed but mapped to the `Tree`. However, we would argue that this is not the case as the model transformation that transforms the L-PSM into the T-PSM does not need to consider the `Program Units`. In the example, it only transforms `Items` of the type `Hierarchical Tree`. As a result, we are not able to continuously trace the `Program Units` from their definition in the source code of the legacy environment to the source code of the target environment. In general, this can be an indicator for the removal of code.

In addition, one could argue if this pattern is required at all since it only prescribes to customize one method fragment that constitutes an artifact. In particular, no activities are defined. While this is true, the pattern enables some kind of *marking* source code that shall not be transformed. We assume that this is essential to enable the assessment of developed methods. In particular, it would not be possible to evaluate whether a transformation method is complete, i.e., whether it covers the whole legacy system.

This concludes the generic description of the pattern, whereby we discussed various characteristics. However, so far we did not discuss in which situations an application of the pattern is appropriate. This discussion is part of the next section.

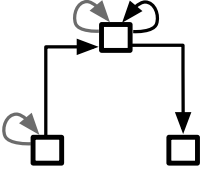
Suitability

In terms on influences on the effectiveness, the necessity of the source code on which the pattern is applied needs to be considered as influence factor. In fact, the pattern can only be effective if it is applied on source code that is not necessary. Otherwise, it will always be ineffective as functionality will be lost during the transformation

Also in terms on influences on the efficiency, the necessity of the source code is the determining influence factor. If the code is not necessary, we assume the pattern to be the most efficient one as it requires the least effort, i.e., no effort. When applying the pattern on necessary code, its efficiency does not need to be considered as it yields an erroneous transformation.

In conclusion, we assume that the pattern shall be applied if it is not necessary to transform the functionality, since it is not used by the legacy system or since it is implicitly provided by the target environment. So far, we focused on patterns to transform some functionality. In the next section, we introduce a pattern to restructure the architecture of a legacy system during its transformation.

5.5.5 Platform-Dependent Architecture Restructuring (A_1)

Intent	Perform an automated restructuring of a legacy system's architecture during its transformation into a new environment
Strategy	An intermediate representation of the system's architecture is used on a platform-independent layer. The representation is reverse engineered from an ASG ⁴ of the legacy system on a platform-specific layer. After a restructuring, it is transformed into an ASG ⁴ of the target environment
Structure	 <pre> graph TD A[] --> B[] A --> C[] A --> A B --> B </pre>

Continued on next page

Applicability	Use when the architecture of the system to transform differs in the source and in the target environment and if the difference is too significant to perform the restructuring implicitly when converting the functionality. In addition, the structures that imply the architecture of the system and those that are affected by the restructuring need to reside in models on the platform-specific layer
Preparation	Applying this pattern requires realizing model transformations rules. In addition, it can be necessary to realize dedicated architecture recovery algorithms or model views
Example	Figure 5.14 shows the enactment of a method which conforms to the pattern. Thereby, the modularization of the running example is changed (cf. Section 4.3)
Known Uses	The strategy realized by the pattern is comparable to the strategy followed by architectural reengineering tools. The method described in [Hec+08] conforms to the pattern
Related Patterns	Similar to A ₂

Table 5.6 Characterization of the *Platform-Dependent Architecture Restructuring Pattern***Example**

This section is separated into two parts. First, we describe the (i) background knowledge covering technical details of the example. Then, we describe the (ii) enactment of a method that conforms to the *Platform-Dependent Architecture Restructuring Pattern*.

Technical Background First, we need to have an understanding of the concept on which this pattern shall be applied. In contrast to the other patterns introduced, this one does not enable to transform some functionality but to restructure the architecture of a system during its transformation. Therefore, we need to have an understanding of the architectural structure we aim to change. Related to the Summit application (cf. Section 4.3), we aim to change its *modularization* (C8). In addition, we demonstrate the impact of this change on the transformation of the *internal representation of the tables* used (C1).

⁴Depending on the situation, using an AST can be sufficient

Second, we need an understanding of how the concept is realized in the legacy system. Any Forms-based system consists of a set of modules. While there are different types of modules, each module is stored in a dedicated file. In the example, we only consider *Form Modules* which contain the definition of user interfaces, source code routines as well as data connections. Related to the Summit application, we use the `ORDERS` and `CUSTOMERS` Form Modules to exemplify the application of the pattern. An excerpt of their source code can be seen in the lower left of Figure 5.14. Note that we do not discuss the realization and mapping of the *internal representation of the tables* used. Instead, we refer to Section 5.5.1 where this has been discussed in great detail.

Besides knowledge of the concept and its realization in the legacy system, we thirdly need to have an understanding of how to realize the functionality in the target environment. In Oracle ADF, the modularization can be defined for each layer separately (cf. Section 3.1), e.g., for the data as well as for the view layer. In the example, we focus on the modularization of the data layer and use a `Model Project` for this purpose. This is a simplification, as the equivalent to a module in Oracle Forms would rather be an `Application Module` than a `Modeling Project`. However, we assume that all data connections contained in that project are later on bundled as one `Application Module`, which acts as a container for defined data connections [Ora13]

Finally, we need to have an understanding of the way in which the architecture shall be restructured. In the example, we apply the mapping described in [RM11, p.8] which suggests merging all modules of the application. It is stated that larger applications might require a more complex restructuring, but merging is suitable for the Summit application. For a more complex restructuring, we refer to the feasibility study described in Section 7.2.

Method Enactment The desired transformation is achieved by enacting the transformation method as shown in Figure 5.14. Thereby we want to point out that the method does not conform to the *Platform-Dependent Architecture Restructuring Pattern*. Instead, the combined use of two method patterns is shown: The internal representation of database tables is transformed by applying the *Language Transformation Pattern* (cf. Section 5.5.1), while the modularization is changed by applying the *Platform-Dependent Architecture Restructuring Pattern*. The reasons for that equal the reasons for the *Code Removal Pattern* (cf. Section 5.5.4). On the one hand, the application of the pattern can be observed best when demonstrating its impact on other patterns. On the other hand, it is another example for a composed pattern.

Related to the transformation of the internal representation of database tables, we do not provide a detailed description of the method parts but refer to the example provided in Section 5.5.1. First, a syntactic and semantic analysis is performed by enacting the activities

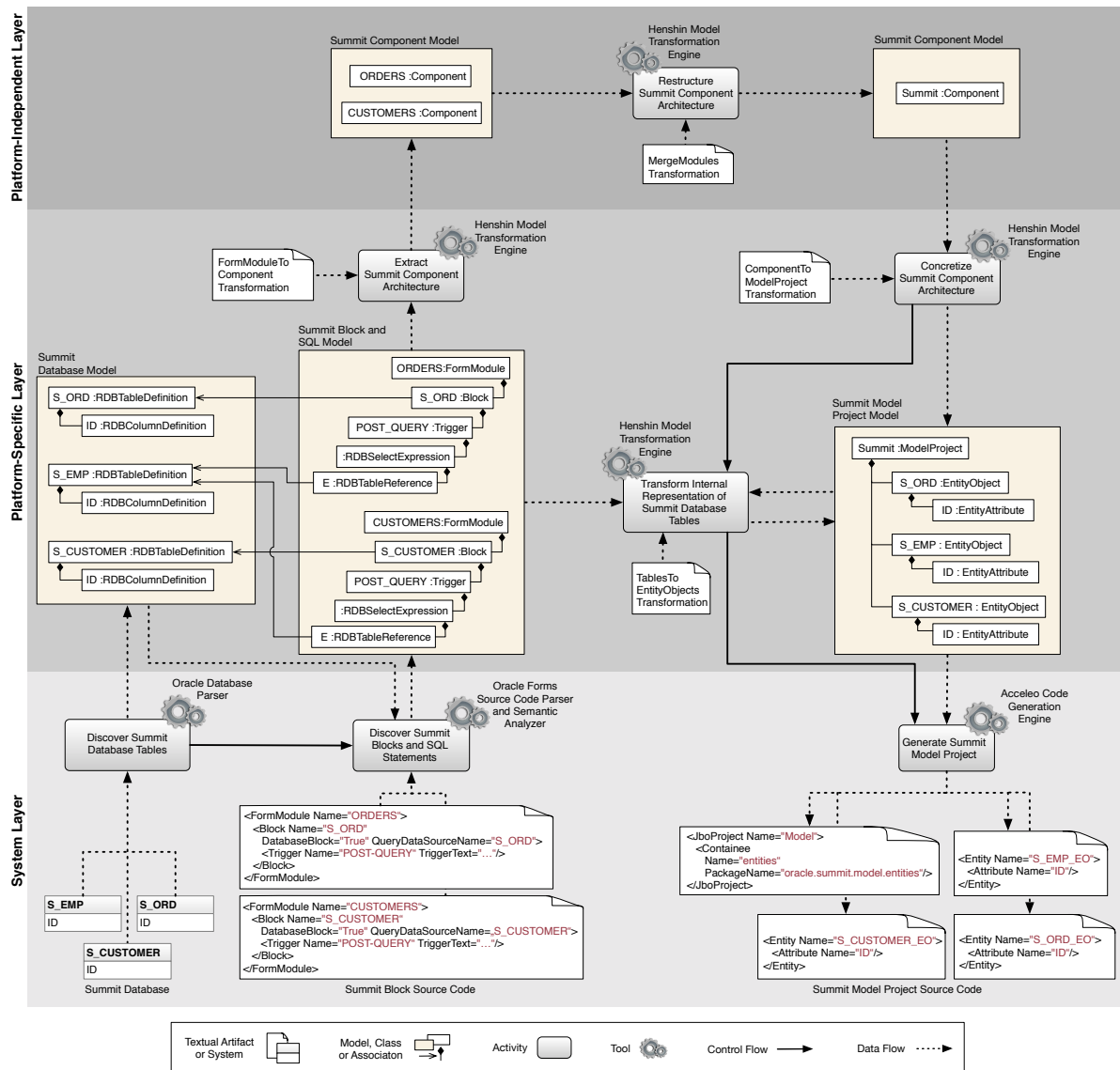


Figure 5.14 Enacting a transformation method to transform the architecture of the running example by using the *Platform-Dependent Architecture Restructuring Pattern*

called *Discover Summit Database Tables* and *Discover Summit Blocks and SQL Statements* to retrieve a model of the database and the ASG of the source code. Then, by performing the activity called *Transform Internal Representation of Summit Database Tables*, a model transformation is executed that transforms the L-PSM into the T-PSM. In particular, the tables and columns of the database model are transformed to Entity Objects and Entity Attributes. As a last step, performing the activity called *Generate Summit Model Project* generates corresponding source code for the target environment.

The three activities shown in the upper part of Figure 5.14 result from applying the *Platform-Dependent Architecture Restructuring Pattern*. At first, an A-PIM gets created by enacting the activity called *Extract Summit Component Architecture*. This activity is a customization of the *Architecture Recovery* fragment and prescribes to perform reverse engineering. In particular, it prescribes how to reconstruct the architecture of the application by interpreting the L-PSM and representing the result on a platform-independent layer. Related to the Summit application, we extract the architecture by transforming each `Form Module` instance into an architectural `Component` which represents a unit of composition [Szy02, p.548]. In general, components can have dependencies to each other. In the case of the example, this could be navigation flows between contained dialogs (cf. Section 4.3). However, to realize the intended restructuring, it was not required to extract them. The resulting model can be seen in the upper left of Figure 5.11, it conforms to the KDM [OMG11b].

As a second step, the architecture gets restructured by modifying the content of the A-PIM using an endogenous model transformation. This is achieved by enacting the activity called *Restructure Summit Component Architecture*, which is a customization of the *Architecture Restructuring* fragment. Related to the Summit application, the restructuring consists of merging all `Components`. The resulting model can be seen in the upper right of Figure 5.11.

Finally, the changed architecture of the application gets concretized in the new environment. In particular, the A-PIM is used to create parts of the T-PSM by applying an exogenous model transformation. This is achieved by enacting the activity called *Concretize Summit Component Architecture*, which is a customization of the *Architecture Concretization* fragment. Related to the Summit application, a model transformation creates a `Model Project` instance in the T-PSM for each `Component` that is contained in the A-PIM. The resulting model can be seen in the middle right of Figure 5.11.

Note that the T-PSM plays an important role in the integration of both patterns, as it is an input to the activity called *Transform Internal Representation of Summit Database Tables*. This data flow dependency makes the *Language Transformation Pattern* (cf. Section 5.5.1) dependent on the (result of the) *Architecture Restructuring Pattern*. From a functional perspective, this is due to the fact that each `Entity Object` needs to be associated to a `Model Project` which is only created by the architecture concretization activity.

This concludes the description of an example for the *Platform-Dependent Architecture Restructuring Pattern*. Subsequently, we describe its generic characteristics.

Description

The *Platform-Dependent Architecture Restructuring Pattern* can be applied to change the architecture of a legacy system during its transformation into a new environment. The basic

idea of how to restructure the architecture is to use an explicit representation of it on a platform-independent layer. This representation is extracted from the source code of the legacy system and mapped to source code in the target environment. More precisely, applying the pattern first requires extracting the architecture of the system to transform on the platform-independent layer as a platform-independent model (A-PIM) by performing an architecture recovery. Then, an architecture restructuring changes the architecture as desired by modifying the A-PIM. By performing an architecture concretization, the A-PIM is transformed into a platform-specific model, i.e., the architecture is realized by platform-specific concepts.

The transformation strategy described by this pattern is related to the strategy followed by architectural reengineering tools. In particular, these tools usually realize three activities, namely *architecture recovery*, *architecture transformation* and *architecture design* [KWC98]. Since the intentions of the processes correspond to the intentions of the three activities that are prescribed by the pattern, the pattern can be used to describe the mechanics of an architectural reengineering tool.

The pattern is similar to the *Platform-Independent Architecture Restructuring Pattern* (A₂). Both patterns only differ in the source of the architecture restructuring activity, i.e., whether a platform-specific or platform-independent representation of the legacy system is used. Thereby, the abstraction layer to use depends on where the information is described that is required to extract the architectural model. In the example, the `Components` are extracted directly from the `Form Modules` that are represented in the L-PSM. Assume that an F-PIM would be used in which `Form Modules` are represented by platform-independent `Modules`. Then, the architectural model could be extracted from the F-PIM instead.

We want to point out that the relation between the A-PIM and models on a lower level of abstraction is slightly different than the relation between the F-PIM and models on a lower level of abstraction. If an instance of some functionality to transform is represented in the L-PSM and F-PIM, then the only information that gets lost is how the functionality was realized in the legacy environment. But, the functionality described is the same in both models, i.e., it is redundant. In contrast, this is not the case for the content of the architectural model. The architectural entities contained do not represent the functionalities described by the models on a lower level of abstraction but aim to represent greater coherences. This can, for example, be achieved by aggregating information contained in platform-specific models. This can also be seen in the example when considering the difference between the `Form Modules` and the `Components` shown. While both seem to be similar, a `Form Module` is solely a representation of a source file. The `S_ORD Block` is contained by it, but the `Form Module` does not represent it. In contrast, the corresponding `Component` is a representation of the entire `Form Module`.

An application of this pattern is described in [Hec+08]. In this work, a Java-based application that has a two-tier architecture is transformed to conform to a three-tier architecture. This is achieved by extracting an A-PIM that represents the layer each code block belongs to. A restructuring activity on this model introduces a new layer and moves code blocks based on defined rules. Finally, the changes are reflected back to the source code.

This concludes the generic description of the pattern, whereby we discussed various characteristics. However, so far we did not discuss in which situations an application of the pattern is appropriate. This discussion is part of the next section.

Suitability

In terms of influences on the effectiveness, we need to consider that this pattern differs from the other patterns by the fact that it is always optional. Transforming a legacy system as part of a software modernization essentially requires retaining its functionality, which can be achieved by applying any of the patterns introduced earlier. In contrast, the *Platform-Dependent Architecture Restructuring Pattern* enables performing some restructurings explicitly on an architectural layer, which is not required to preserve the functionality.

However, if an architectural restructuring is desired, transformations that preserve the functionality need to implicitly consider this restructuring. For example, in the case of a *Language Transformation Pattern*, the direct model transformation between the L-PSM and T-PSM needs to realize this restructuring, in addition to the mapping between the programming languages used. As discussed in Section 5.5.1, this would increase the complexity of the model transformation which can, in turn, decrease the effectiveness. By using the *Architecture Restructuring Pattern*, the concern of extracting and restructuring the architecture can be separated from the preservation of the functionality. Intuitively, the resulting architectural model can be used to parameterize transformations on a lower level of abstraction, possibly making them effective. Therefore, influences on the effectiveness are implicitly given by the influence factors of the other patterns.

In terms of influences on the efficiency, we observed the complexity of the architecture recovery and restructuring activities to essentially influence the efficiency of the pattern. In the example, we used a simple one-to-one mapping between `Form Modules` and `Components` to extract an architectural model while subsequently merging all resulting `Components`. However, in general, architecture recovery and restructuring are complex endeavors that can require applying pattern-matching or clustering techniques.

In conclusion, we assume that the pattern is particularly suitable if the architecture of the system in the source and in the target environment differs significantly, while large parts of the functionality are converted so that dependent transformations benefit from the architectural

model. As can be seen in the example shown in Figure 5.14, patterns can be related to each other. In the next Section, we introduce a set of composed patterns that result by combining two or more of the method patterns introduced so far.

5.6 Composed Transformation Patterns

In this section, we introduce a set of composed patterns, shown in Figure 5.15. A composed pattern results from combining two or more basic patterns. An example is the transformation method described in section 5.5.5. The method is a result of applying the *Platform-Dependent Architecture Restructuring Pattern* and the *Language Transformation Pattern*.

Based on the example, it could be seen that some integration effort is necessary so that an integrated transformation method results. In particular, the data flow needed to be precisely defined as part of the integration in order to ensure that activities which originate from different patterns are performed in a correct order. In general, the use of a composed pattern always requires integration of the method fragments that originate from different patterns. Therefore, such integration is an essential part of the method engineering process of MEFiSTo. The integration is addressed in detail in Section 6.4.2.

To understand the usefulness of this additional set of patterns, we need to distinguish two cases: a composed pattern can either *emerge implicitly* or be *applied explicitly*. A composed pattern emerges implicitly if a legacy system is divided into distinct parts, whereby different basic patterns are applied on different parts to transform the contained functionality. Then, the integration needs to be addressed retrospectively.

Providing a set of composed patterns as part of the method base enables considering the combined use of basic patterns prospectively. Based on our observations, this offers at least two benefits. On the one hand, it prevents the need for a fine-granular description of the functionality of the system to transform. In this sense, it provides an additional degree of freedom in the specification of the functionality. On the other hand, it enables considering integration effects during the development of a method explicitly. This allows a fine-granular adaptation of the method to the situation at hand.

Subsequently, we introduce the composed patterns in the same way as we introduced the basic patterns in Section 5.5. However, in contrast to the basic patterns, we do not describe each composed pattern but only selected ones, i.e., patterns F₅ and F₇. This is due to the observation that the difference in the characteristics between the basic and the composed patterns only arises due to integration effects. Therefore, we focus on covering all integration effects but not all patterns. Nevertheless, the characteristics of patterns that have not been described can be found in the appendix (cf. Section A).

The pattern F_5 results when combining the *Language Transformation Pattern* F_1 and the *Reimplementation Pattern* F_3 . In this instance, the integration effect arises due to the integration of an automated conversion and a manual reimplementation. As this effect equals the one for pattern F_6 , we do not describe F_6 separately. The pattern F_7 results when combining the *Language* and *Conceptual Transformation Pattern*. As this results in a unique integration effect, we describe this pattern explicitly, too.

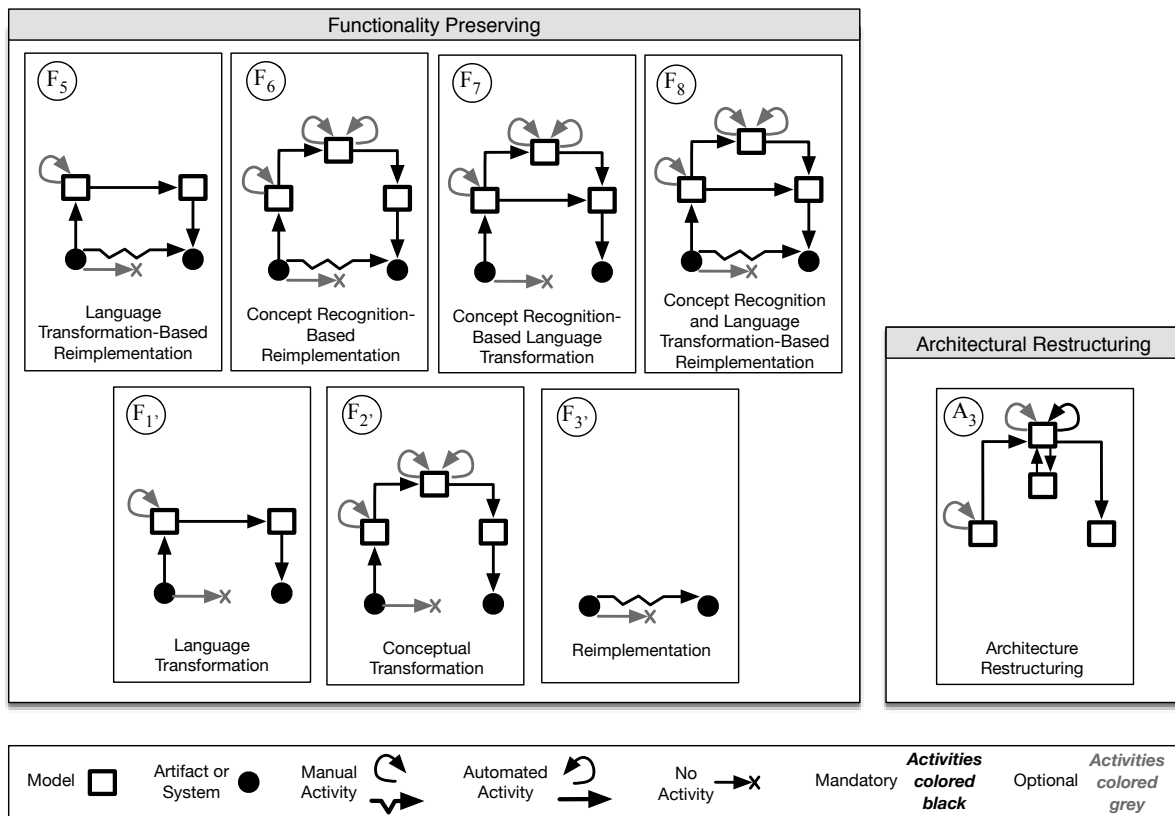


Figure 5.15 Composed transformation method patterns

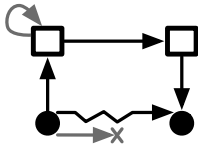
We do not describe patterns $F_{1'}$, $F_{2'}$ and $F_{3'}$ which are a result of combining the basic patterns F_1 , F_2 and F_3 with the *Code Removal Pattern* F_4 . An example for the pattern $F_{1'}$ is the method that has been described in Section 5.5.4. As the removal of code is integrated as an optional part, the characteristics of patterns F_1 , F_2 and F_3 still apply. In this case, the integration effect consists of a reduced specification effort, i.e., it is not necessary to separately specify parts that shall not be transformed.

The pattern F_8 is a result of combining patterns F_1 , F_2 and F_3 . As the integration effects are also a combination as the effects described for patterns F_5 , F_6 and F_7 , we do not discuss this pattern separately. Also, we do not describe method pattern A_3 . The pattern is a result of

combining patterns A_1 and A_2 which can reduce the specification effort but does not result in any additional integration effects.

Subsequently, we describe the first composed pattern, namely the *Language Transformation-Based Reimplementation Pattern*. Its methodological solution to preserve functionality consists of integrating manual reimplementation activities with the results of an automatic conversion.

5.6.1 Language Transformation-Based Reimplementation (F_5)

Intent	Perform a semi-automatic transformation of the legacy system's functionality into a new environment by combining a conversion- and a reimplementation-based transformation strategy
Strategy	Definition of a direct mapping between the programming languages of the environments involved to transform parts of the functionality, while remaining parts are reimplemented manually. This is realized by representing the legacy system as a model of its ASG ⁵ on a platform-specific layer. A model transformation that transforms this model into an ASG ⁵ of the target environment is a realization of the mapping between the programming languages involved. Subsequently, code is generated in the target environment that forms the basis for reimplementation activities performed by software developers
Structure	
Applicability	Use when large parts of the functionality to transform are realized comparably in the legacy and target environment and the legacy system has a sufficient size. The difference in the realization determines the complexity of the mapping between the programming languages involved, influencing the efficiency and effectiveness of the pattern. Those parts, which are realized significantly different, shall be reimplemented. Thereby, the amount of available developers and their experience form influence factors to consider

Continued on next page

Preparation	Applying this pattern essentially requires realizing a parser, model-to-model transformations, code generation rules and guidance documents to systematize the reimplementation. In addition, it can be necessary to realize a semantic analyzer or model views
Example	Figure 5.16 shows the enactment of a method which conforms to the pattern. In particular, it demonstrates the transformation of the internal representation of database tables as well as attribute validation rules of the running example (cf. Section 4.3)
Known Uses	The strategy realized by the pattern is comparable to the strategy followed by an MDA-based development process that starts on the platform-specific layer. While parts of the functionality to transform are automatically generated using a model-driven tool chain, remaining parts are completed manually. The method described in [Fuh+12, pp.170-174] to some degree conforms to the pattern
Related Patterns	Combines F ₁ , F ₃ and F ₄ ; Similar to F ₆

Table 5.7 Characterization of the *Language Transformation-Based Reimplementation Pattern*

Example

This section is separated into two parts. First, we describe the (i) background knowledge covering technical details of the example. Then, we describe the (ii) enactment of a method that conforms to the *Language Transformation-Based Reimplementation Pattern*.

Technical Background First, we need to have an understanding of the functionality to transform. In this example, we aim to transform a part of the *table-based data access* functionality realized in the Summit application (cf. Section 4.3). More specifically, we demonstrate the transformation of the *internal representation of the tables* (C1) as well as related *attribute validation rules* (C2).

We do not go into detail on how these functionalities are realized in the legacy system and shall be realized in the target environment. Instead, we refer to Sections 5.5.1 and 5.5.2 where this has been discussed in great detail. In short, it is appropriate to transform the internal

⁵Depending on the situation, using an AST can be sufficient

representations of database tables by applying a *Language Transformation Pattern*, as the functionality is realized comparably in both environments. For this example, we assume that *Blocks* and *Items* need to be mapped to *Entity Objects* and *Entity Attributes*, which is a simplification. The same pattern is not appropriate to transform the attribute validation rules, as they are realized imperatively in the source environment but declaratively in the target environment. In particular, imperative PL/SQL source code that is part of a *Trigger* needs to be mapped to a *Validation Bean* that is part of an *Entity Object*. In Section 5.5.2 we showed how the *Conceptual Transformation Pattern* can be applying to realize the transformation by using program comprehension. In this example, we use the *Reimplementation Pattern* instead.

Method Enactment The desired transformation is achieved by enacting the transformation method as shown in Figure 5.16. The method consists of customized method fragments that have been introduced in Section 5.3 and is an instance of the *Language Transformation-Based Reimplementation Pattern*.

The activities shown in the upper two layers of Figure 5.16 originate from the *Language Transformation Pattern*, while the activity called *Reimplement Summit Attribute Validation Rules* that is shown in the lower layer originates from the *Reimplementation Pattern*. As the enactment of exemplary methods for both patterns has already been discussed in Sections 5.5.1 and 5.5.3, we do not go into detail on every activity but focus on those parts that are affected by the integration of both patterns.

An example for an affected activity is the activity called *Annotate Summit Attribute Validation Rules*, shown in the upper left of Figure 5.16. This activity is a customization of the *Enrichment* fragment which describes to add additional information to an existing model. It is an optional fragment of the *Language Transformation Pattern*. In the example, this activity is performed manually by a system expert who annotates parts of the L-PSM using key-value pairs. In particular, she annotates all statements that represent an attribute validation rule by adding a *Concept = Attribute Validation Rule* annotation. In addition, she annotates to which *Attributes*, i.e., to which *Items*, this rule refers to.

The subsequent activity called *Transform Summit Table-Based Data Access* is a customization of the *Language Transformation* fragment. Here, a set of model transformation rules is executed that transform the L-PSM into a T-PSM. The resulting model can be seen in the upper right of Figure 5.16. Thereby, the annotations are used to partially realize the attribute validation rules in the resulting model. For each annotated statement a *Compare Validation Bean* gets instantiated, whereby the source code of the statement is stored in text form as part of the *Note* attribute. In addition, the annotations that specify to which attributes a rule refers to are used to instantiate a *Trigger* with references to the corresponding *Entity Attributes*.

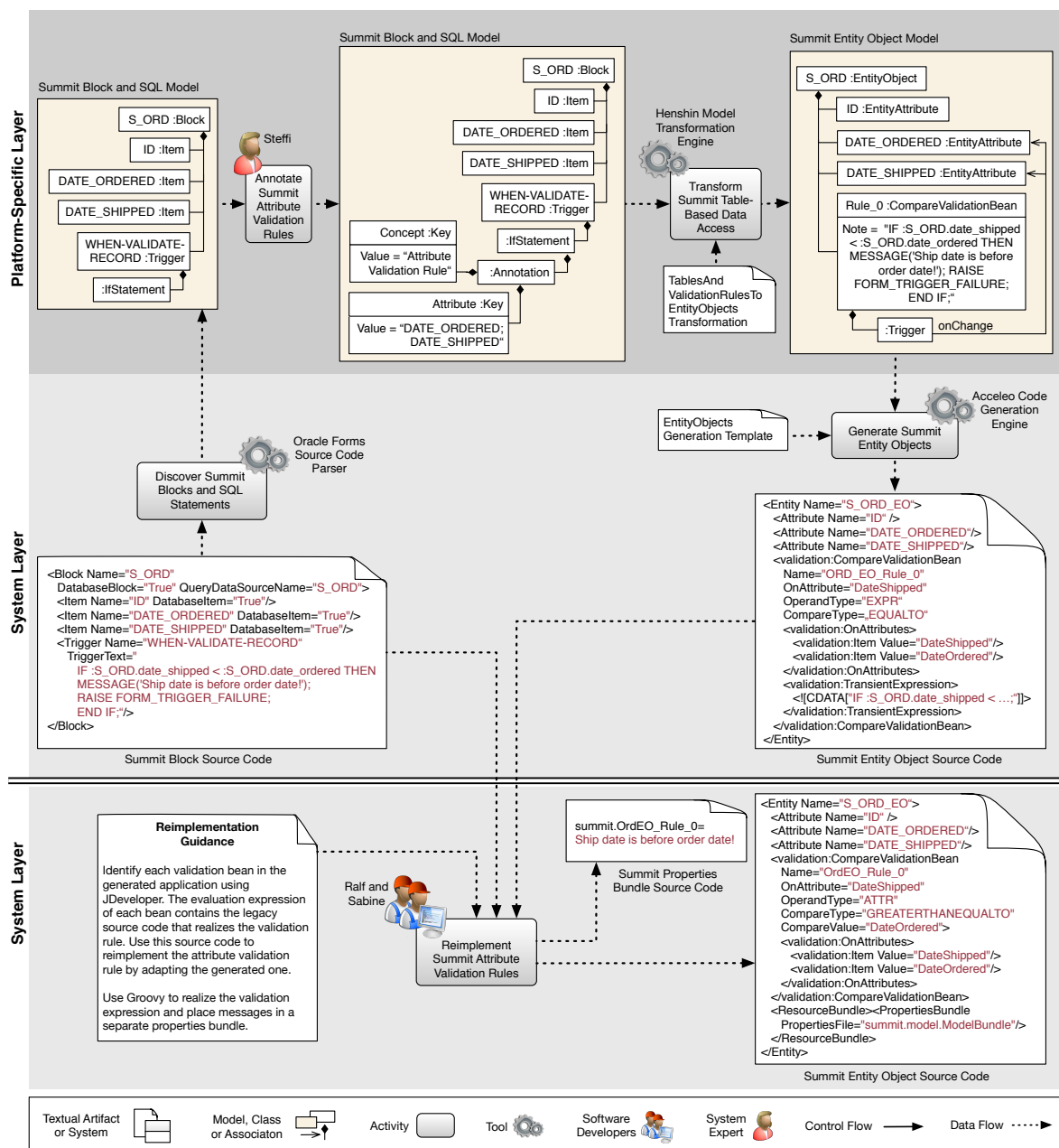


Figure 5.16 Enacting a transformation method to transform the internal representation of database tables and associated attribute validation rules of the running example by using the *Language Transformation-Based Reimplementation Pattern*

By enacting the last activity that is a result of applying the *Language Transformation Pattern*, named *Generate Summit Entity Objects*, source code is generated in the target environment. The result can be seen in the right side of Figure 5.16. In the example, the internal representation of database tables, realized by Entity Objects and corresponding Entity Attributes, are generated completely. In contrast, the attribute validation rules, realized by Compare

Validation Beans, are only generated incompletely. In particular, an implementation of the validation expression is missing as the generated expression only contains the legacy source code of the validation rule as text.

The missing parts are completed by subsequently enacting the activity called *Reimplement Summit Attribute Validation Rules*. The activity is a customization of the *Reimplementation* fragment, it results from applying the *Reimplementation Pattern*. The activity can be seen in the lower layer of Figure 5.16. In the example, two software developers perform the reimplementation manually. As an input, they use the generated *Summit Entity Object Source Code*, the legacy *Summit Block Source Code* as well as a *Reimplementation Guidance* artifact. Note that the data flow dependency of this activity on the transformed source code realizes the integration of both patterns. The guidance artifact specifies what the developers are supposed to do. In this case, they shall browse the generated source code in their default development environment and identify attribute validation rules. For each rule, they shall understand the validation expression by reading the legacy source code and reimplement it as a groovy expression. Also, messages shall be placed in a separate properties bundle. This results in the completed source code that is an output of this activity.

This concludes the description of an example for the *Language Transformation-Based Reimplementation Pattern*. Subsequently, we describe its generic characteristics.

Description

The *Language Transformation-Based Reimplementation Pattern* can be applied to transform the functionality of a legacy system into a new environment, following a conversion- and reimplementation-based transformation strategy. The basic idea of how to transform the functionality is to define a mapping between the programming languages of both environments for large parts of the functionality. Remaining parts, for which a direct mapping becomes too complex, are subsequently reimplemented manually. More precisely, applying the pattern requires representing large parts of the functionality to transform on the platform-specific layer as a model. Initially, the functionality is represented by a platform-specific model of the legacy environment (L-PSM) that gets created by performing a model discovery. A model transformation transforms the L-PSM into a platform-specific model of the target environment (T-PSM). The model transformation is a realization of the mapping between the programming languages involved. Based on the content of the T-PSM, source code is generated in the target environment. Those parts of the functionality that have not been transformed automatically are subsequently reimplemented manually by software developers.

The transformation strategy described by this pattern is comparable to a model-driven development process, in particular, to the Model-Driven Architecture (MDA) (cf. Section 2.1.2).

In MDA, models are used on various levels of abstraction during the development, whereby the lowest level is the platform-specific layer [OMG14]. The PSM on that layer is interpreted and executed directly, or, it is used to generate code. Code can be completely or partially generated. In the last case, developers implement missing parts manually [BCW12, p.29].

In comparison, the method that results when applying the *Language Transformation-Based Reimplementation Pattern* only differs in two main aspects. First, the instantiation of the PSM differs. While in an MDA-based process a PSM results from transforming a PIM, the pattern prescribes to instantiate it from another PSM. Intuitively, a compiler-based approach is applied to enable a lateral entry into an MDA-based development process. From this point on the process is the same except for, secondly, the way in which the developers are guided. In particular, the information source differs. While in an MDA-based process the PSM shall contain all information required for reimplementation [BCW12, p.41], the legacy system always takes this role in case of a modernization.

Note that the information source can be shifted, which can be seen based on the method shown in Figure 5.16. In the example, it would actually not be necessary to access the source code in order to reimplement the validation rules, as all required information are contained. This can be beneficial if a context change for the software developers is prevented, i.e., they do not need to leave their development environment but find all relevant information there. On the one hand, this can be seen as a synergy effect resulting from the integration of both pattern. On the other hand, this creates a dependency of reimplementation activities on the output of the language transformation. If the source code of the legacy system is directly used, reimplementation activities could also be performed in parallel. However, integration always needs to be considered, either implicitly or explicitly.

To some degree, an application of this pattern is described in [Fuh+12, pp.170-174]. In this work, a model-driven transformation from a client-server to a Service Oriented Architecture (SOA) has been performed as a case study. First, the legacy system was represented as a platform-specific model. However, no language transformation was performed but some kind of restructuring as corresponding parts of the model have been identified that form a service. For each part identified, code has been generated based on the PSM that is not complete but reimplemented manually by developers afterwards.

This concludes the generic description of the pattern, whereby we discussed various characteristics. However, so far we did not discuss in which situations an application of the pattern is appropriate. This discussion is part of the next section.

Suitability

In terms of influences on the effectiveness, we have to consider the influence factors of the basic patterns that result when decomposing this composed one. For those parts of the functionality that shall be automatically converted, we need to consider the realization in the legacy system as well as the desired realization in the target environment as essential influence factors. As discussed in Section 5.5.1, these factors influence the complexity of the model transformation between the L-PSM and T-PSM. This, in turn, can lead to a deviation from an initially desired realization, negatively influencing the effectiveness.

For those parts of the functionality that shall be reimplemented, the skills of the software developer that perform the reimplementation activity need to be considered as an essential influence factor. As discussed in Section 5.5.3, the developers always have a certain degree of freedom when performing reimplementation activities. Therefore, the effectiveness can depend on the micro design-decisions they perform.

Also in case of influence factors on the efficiency we need to consider the influence factors of the underlying basic patterns. For those parts of the functionality that shall be automatically converted, the effort to develop a syntactic and/or semantic analyzer, metamodels and code generation rules needs to be considered. In addition, the complexity of the model transformation on the platform-specific layer as well as the size of the functionality to convert are essential influence factors.

For those parts of the functionality that shall be reimplemented, the amount of developers available as well as their skills form essential influence factors. Also, the size of the part of the functionality to reimplement needs to be considered. It should be sufficiently small so that an automatic conversion would be inefficient.

In conclusion, we assume that the pattern is particularly suitable if large parts of the functionality to transform are realized comparably in both environments while the overall source code to transform is sufficiently large. That parts of the functionality that are realized significantly different should not be too large while developers should be familiar with the target environment. Based on the description of this pattern, it could be seen that the integration of an automated conversion and a manual reimplementation needs to be carefully considered. Subsequently, we describe another special integration effect that can arise when integrating conversion-based transformation strategies that rely on the use of different levels of abstraction.

5.6.2 Concept Recognition-Based Language Transformation (F₇)

Intent	Perform an automated transformation of the legacy system's functionality into a new environment, following a conversion-based transformation strategy
Strategy	Use an intermediate representation of parts of the functionality to transform on a platform-independent layer to enhance a direct mapping between the programming languages of the environments involved. The intermediate representation is reverse engineered from an ASG ⁶ of the legacy system on a platform-specific layer. When the ASG ⁶ of the legacy system is transformed into an ASG ⁶ of the target environment, the model transformation used is dependent on the transformation of the intermediate representation
Structure	
Applicability	Use when the functionality to transform is realized significantly different in the legacy and target environment and when the complexity of a direct transformation becomes low if parts of the functionality are made explicit. Also, the legacy system needs to have a sufficient size. The use of an intermediate representation can reduce the complexity of a direct transformation by separating the concerns of reverse engineering, restructuring and mapping the functionality. The manifestation of these concerns essentially influences the efficiency and effectiveness of the pattern
Preparation	Applying this pattern essentially requires realizing a parser, model-to-model transformations and code generation rules. In addition, it can be necessary to realize a semantic analyzer, dedicated reverse engineering algorithms or model views

Continued on next page

Example	Figure 5.17 shows the enactment of a method of a method. Thereby, the internal representation of database views of the running example is transformed (cf. Section 4.3)
Known Uses	The strategy realized by the pattern is comparable to the strategy followed by a compiler that uses an abstract representation of the source code to compile in order to optimize the result. The method described in [San14, pp.197-199] conforms to the pattern
Related Patterns	Combines F_1 , F_2 and F_4

Table 5.8 Characterization of the *Concept Recognition-Based Language Transformation Pattern***Example**

This section is separated into two parts. First, we describe the (i) background knowledge covering technical details of the example. Then, we describe the (ii) enactment of a method that conforms to the *Concept Recognition-Based Language Transformation Pattern*.

Technical Background First, we need to have an understanding of the functionality to transform. In this example, we aim to transform the *view-based data access* functionality (cf. Section 4.3). In particular, we show how the pattern can be used to transform the *internal representation of the views* (C3).

We do not go into detail on how these functionalities are realized in the legacy system and shall be realized in the target environment. Instead, we refer to Section 5.5.3 where this has been described in great detail. In short, database views are defined by `Blocks` and `Items` in Oracle Forms, while they are mapped to `View Objects` and corresponding `View Attributes` in Oracle ADF.

While the pattern exemplifies the transformation of the internal representation of database views, we additionally assume how the internal representation of database tables are transformed. In particular, we assume that the mapping that has been described in detail in Section 5.5.1 is applied. In short, database tables that are referenced by `Items` or imperative source code in Oracle Forms are represented by `Entity Objects` and corresponding `Entity Attributes` in Oracle ADF.

⁶Depending on the situation, using an AST can be sufficient

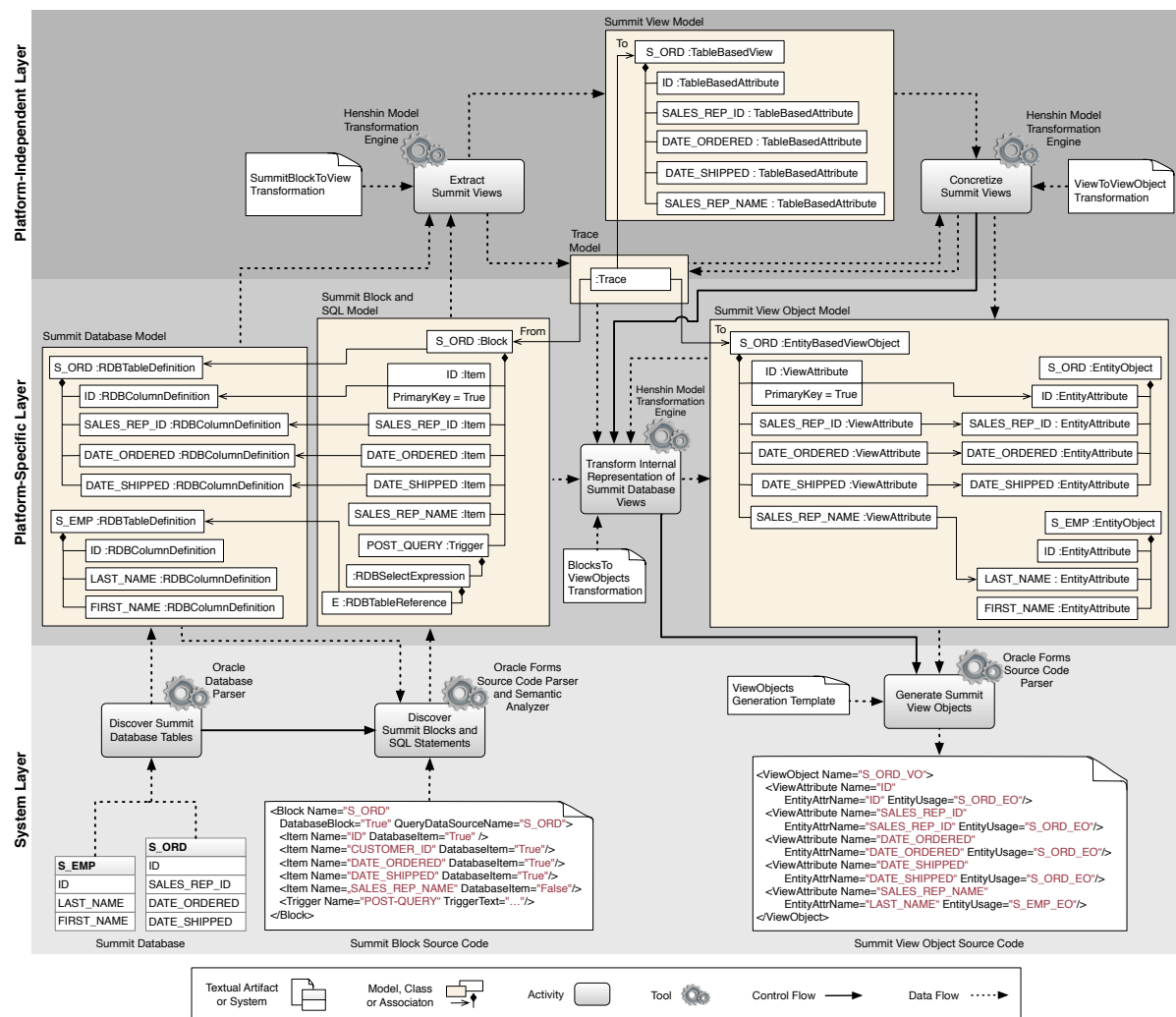


Figure 5.17 Resulting models when enacting a transformation method to transform the internal representation of database views of the running example by using the *Concept Recognition-Based Language Transformation Pattern*

Method Enactment The desired transformation is achieved by enacting the transformation method as shown in Figure 5.16. Note that most of the artifacts are presented shortened. The method consists of customized method fragments that have been introduced in Section 5.3 and is an instance of the *Concept Recognition-Based Language Transformation Pattern*.

The two activities shown in the highest layer of Figure 5.17 originate from the *Conceptual Transformation Pattern*, while the activity on the middle layer originates from the *Language Transformation Pattern*. In contrast, the activities on the lower layer are part of both patterns. As the enactment of exemplary methods has already been discussed in Sections 5.5.1 and 5.5.2, we do not go into detail on every activity but focus on those parts that are affected by the integration of both patterns.

An example for an affected activity is the activity called *Extract Summit Views*, shown in the upper left of Figure 5.10. The activity is a customization of the *Program Comprehension* fragment, its purpose is to extract the database views defined within the Oracle Forms application and represent it as an F-PIM. When applying the *Concept Recognition-Based Language Transformation Pattern*, the F-PIM does not completely represent the functionality to transform but only a part of it. That part is used to enhance a dependent language transformation. Related to the Summit application, the extraction of the view called S_ORD is shown, which is defined by the equally named Block. The resulting F-PIM does only represent the structure of the view, i.e., the view and its fields, but does not model technical details. For example, key-relations are not modeled on the platform-independent layer as well as the information to which database field a Table Based Attribute corresponds to.

Since the missing information is contained in the L-PSM, a *Trace Model* that links both models is an additional output of the activity. In the example, the contained Trace element links the S_ORD Table Based View element and its origin, namely the S_ORD Block. While the figure only shows one Trace element, we assume that each part of the *Summit View Model* is traced. For example, each Table Based Attribute is linked to an Item and its data source, i.e., to the associated RDB Column Definition from which the data is fetched.

The subsequent activity called *Concretize Summit Views* is a customization of the *Concretization* fragment. By enacting the activity, the F-PIM gets transformed into the T-PSM, shown in the middle right of Figure 5.17. In the example, enacting the activity invokes the execution of a model transformation that creates the Entity Based View Object element as well as its contained View Attributes. However, properties are not created, like the Primary Key value of the View Attribute called ID, as well as references to Entity Objects. The *Trace Model* is updated to reference elements of the *Summit View Object Model*.

The missing technical details are added by enacting the activity called *Transform Internal Representation of Summit Database Views*, which is a customization of the *Language Transformation* fragment. Thereby, the model transformation that is executed when performing the activity considers the trace information provided by the *Trace Model* to retrieve the coherences between the involved models. Note that the data flow dependency of the activity on the *Trace Model* and the T-PSM realizes the integration of the *Conceptual Transformation* and the *Language Transformation Pattern*. As a last step, code is generated in the target environment by enacting the activity called *Generate Summit View Objects*.

This concludes the description of an example for the *Concept Recognition-Based Language Transformation Pattern*. Subsequently, we describe its generic characteristics.

Description

The *Concept Recognition-Based Language Transformation Pattern* can be applied to transform the functionality of a legacy system into a new environment, following a conversion-based transformation strategy. The basic idea of how to transform the functionality is to explicitly represent a part of it on a platform-independent layer. This representation is extracted from the source code of the legacy system and used to enhance a mapping between the programming languages of both environments. More precisely, applying the pattern first requires representing the functionality to transform on the platform-specific layer as a platform-specific model of the legacy environment (L-PSM) by performing a model discovery. Then, a part of the functionality to transform gets reverse engineered from the L-PSM and represented explicitly by a platform-independent model (F-PIM). By performing a concretization, the F-PIM is transformed into a platform-specific model of the target environment T-PSM. Trace information is maintained so that a subsequent language transformation can add missing parts in the T-PSM by transforming the L-PSM while using trace links. Finally, source code is generated based on the T-PSM.

The transformation strategy described by this pattern can be related to the mechanics of a compiler. Besides using an intermediate representation that holds all information of the code to compile, some compilers use an additional representation to enhance the code generation process [Aho+06, p.525-542]. For example, a Control Flow Graph (CFG) can be used that represents an abstraction of the code, i.e., it does not contain all information but focuses on control flow relations. Optimizations can be defined purely on the CFG that are then reflected back on the intermediate representation. This is comparable to the strategy described by the pattern. In the example, the F-PIM abstracts from technology-specific terminology and explicitly represents the structure of the underlying functionality. The pattern enables to perform restructurings on this platform-independent representation, so that it could be used to describe the mechanics of a compiler, too.

On the other side, the representation by an F-PIM does not need to be restructured in order to enhance a language transformation. In fact, we assume that it can already be beneficial to just explicitly represent parts of the functionality on a platform-independent layer. We apply the same argument as for the *Conceptual Transformation Pattern* (cf. Section 5.5.2), namely the fact that the use of a platform-independent layer separates the concerns of interpreting the L-PSM and mapping the result in the target environment. Note that, when applying this pattern, the mapping occurs as part of the concretization and the language transformation. Thereby, the amount of information being transformed per activity can vary.

This concludes the generic description of the pattern, whereby we discussed various characteristics. However, so far we did not discuss in which situations an application of the pattern is appropriate. This discussion is part of the next section.

Suitability

In terms of influences on the effectiveness, we have to consider the influence factors of the basic patterns that result when decomposing this composed one. For the *Language* as well as for the *Conceptual Transformation Pattern*, the realization of the functionality to transform in the legacy system and the desired realization in the target environment are essential influence factors. The difference in the realization will determine the complexity of the *Language Transformation*, which can be counteracted by using a platform-independent layer. This has been discussed in detail in Sections 5.5.1 and 5.5.2.

Also, in case of influence factors on the efficiency we need to consider the influence factors of the underlying basic patterns. As both patterns follow a conversion-based transformation strategy, the effort to develop a syntactic and/or semantic analyzer, metamodels and code generation rules needs to be considered. In addition, we assume the effort to develop the model transformation that transforms the L-PSM into the T-PSM as well as the effort to realize the program comprehension activity to be significant. The effort is determined by the differences in the realization.

Also, the size of the source code to transform needs to be considered as an influence factor. Only if the code is sufficiently large, the benefits of automating the transformation will compensate the effort for developing tools. In particular, for this pattern, the effort for the development of tools needs to be carefully evaluated. As the pattern relies on the use of trace information between different, possibly large models, developing scalable tools is critical.

In conclusion, we assume that the pattern is particularly suitable if the functionality to transform is realized differently in both environments and when the complexity of a direct transformation becomes low if parts of the functionality are made explicit, while the source code to transform is sufficiently large. This concludes an informal description of the composed patterns. In the next section, we describe how we formalized them.

5.7 Formalization

In this section, we describe how we formalize the content of the method base, i.e., the proposed fragments and patterns. An overview of the different artifacts involved in the formalization as well as their relations is shown in Figure 5.18.

The figure is separated into two layers. Artifacts belonging to the upper layer are *project-independent* artifacts. They are defined as part of this thesis and support the development of transformation methods. In contrast, artifacts belonging to the lower layer are *project-specific* ones. They are created when enacting the method engineering process of the MEFiSTo framework, described in Section 6.

In MEFiSTo, we do not directly specify the developed transformation method in SPEM but initially use an intermediate representation instead. That intermediate representation is automatically transformed into SPEM by executing a model transformation. The reasons for this are twofold: First, as SPEM is a comprehensive and therefore complex language, the specification can become a cumbersome task. By using an intermediate representation that only supports specifying essential constituents of a method by abstracting from SPEM-specific constituents, we aim to reduce this complexity. Second, the use of an intermediate language eases the replacement of SPEM. If another language to specify methods shall be used, it is only required to develop a new model transformation that transforms the intermediate representation into that language.

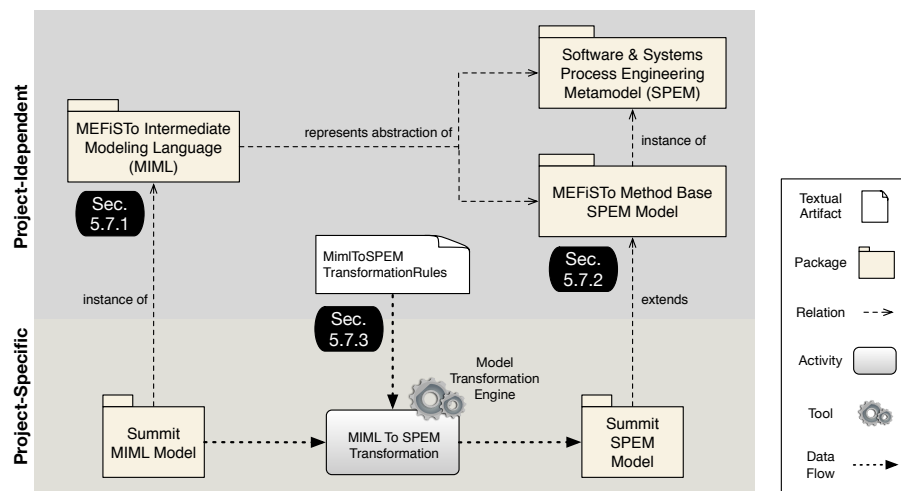


Figure 5.18 Overview of packages and their relations when formalizing a method in MEFiSTo

The *MEFiSTo Intermediate Modeling Language (MIML)* artifact that is shown on the upper left side of Figure 5.18 represents the intermediate language of MEFiSTo. An instance of this metamodel describes a project-specific method, an example is the *Summit MIML Model*. We go into detail on these two artifacts and their content in Section 5.7.1.

The three artifacts in the right side of Figure 5.18 constitute a formal description of a transformation method using SPEM. On the top, the *Software And Systems Process Engineering Metamodel* resides as defined by the OMG [OMG08a]. The *MEFiSTo Method Base SPEM Model* is an instance of that metamodel, it specifies the generic method fragments as proposed in Sections 5.3 and 5.4. A project-specific model can extend these method fragments to specify customized ones. An example for such a model is the *Summit SPEM Model*. We go into detail on these three artifacts and their content in Section 5.7.2. In addition, we discuss the model transformation to transform MIML into SPEM in Section 5.7.3.

5.7.1 MEFiSTo Intermediate Modeling Language (MIML)

In MEFiSTo, a transformation method gets initially specified by a model of an intermediate language called MIML. In Figure 5.19, an excerpt of the metamodel of MIML and a corresponding model of a project-specific method is shown.

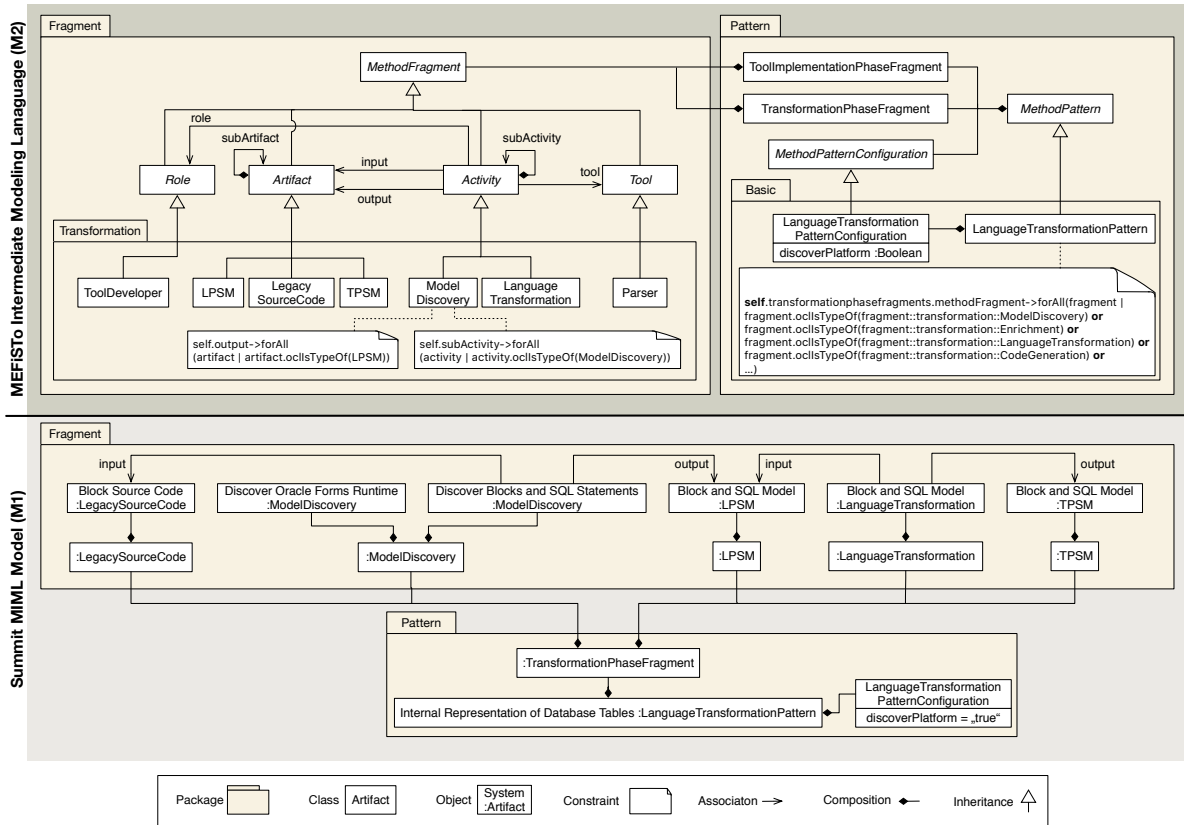


Figure 5.19 Model of the transformation method introduced in Section 5.5.1 in MIML (excerpt)

The upper layer shows the content of the MIML metamodel. The package called *Fragment* provides elements to specify the essential constituents of a method, namely roles, artifacts, activities and tools. Sub-packages provide explicit language elements for all proposed method fragments by refining the generic constituents. For example, the package named *Transformation* provides elements to specify fragments of the equally named phase, e.g., the *LPSM* artifact or *Model Discovery* activity (cf. Sections 5.3 and 5.4).

Elements of the *Fragment* package are associated with constraints that specify invariants of method fragment instances. Those invariants are defined in the Object Constraint Language (OCL). For example, an instance of the *Model Discovery* fragment can only have sub-activities of the same type while the output has to consist of *LPSM* instances.

The Pattern package provides means to specify Method Patterns. A Method Pattern is associated with a Configuration, as well as with a set of Tool Implementation- and Transformation Phase Fragments. Equivalent to the Fragment package, sub-packages provide explicit language elements for all method patterns proposed (cf. Sections 5.5 and 5.6). For example, the package named Basic provides elements to specify an application of the Language Transformation Pattern. In addition, constraints are used to specify invariants of Method Patterns. In the figure, an excerpt of the constraint for the Language Transformation Pattern is shown. The constraint specifies which fragments can only be part of the pattern.

The lower layer shows a model which is an instance of MIML. It specifies an excerpt of the transformation method introduced in Section 5.5.1. When using the MEFiSTo framework, such a model would be systematically instantiated by enacting the method engineering process described in Chapter 6. At first, the pattern to apply would be specified. In the example, we specify to use the Language Transformation Pattern element and provide its Configuration, shown at the bottom of Figure 5.19. Then, an initial set of method fragments corresponding to the pattern would be instantiated automatically. On the one hand, the manifestation of that set depends on the constraints specified in the metamodel. On the other hand, it depends on the configuration of the pattern. Here, the pattern is configured to include the discovery of a platform which results in the instantiation of an additional *Model Discovery* activity.

This concludes a brief description on how we specify transformation methods in MIML. We want to point out that we implemented the MIML metamodel as part of this thesis. For this purpose, we used the *Eclipse Modeling Framework*⁷. In the next section, we discuss how to specify the same method in SPEM.

5.7.2 SPEM

When formalizing transformation methods in SPEM, we distinguish three artifacts: The SPEM metamodel, a project-independent model of the content of the method base as well as a model of the project-specific method. In Figure 5.20, an excerpt of these artifacts is visualized by layers, whereby the content of each layer constitutes the content of the corresponding artifact.

The highest layer shows the content of the SPEM metamodel. While SPEM consists of various packages (cf. Section 2.2.2), we focus on four of them to exemplify the formalization of a method. The package named Managed Content provides means to manage the content of a method, e.g., by defining categories. The actual content of a method is defined by elements of

⁷<https://eclipse.org/modeling/emf/> (accessed March 22th, 2016)

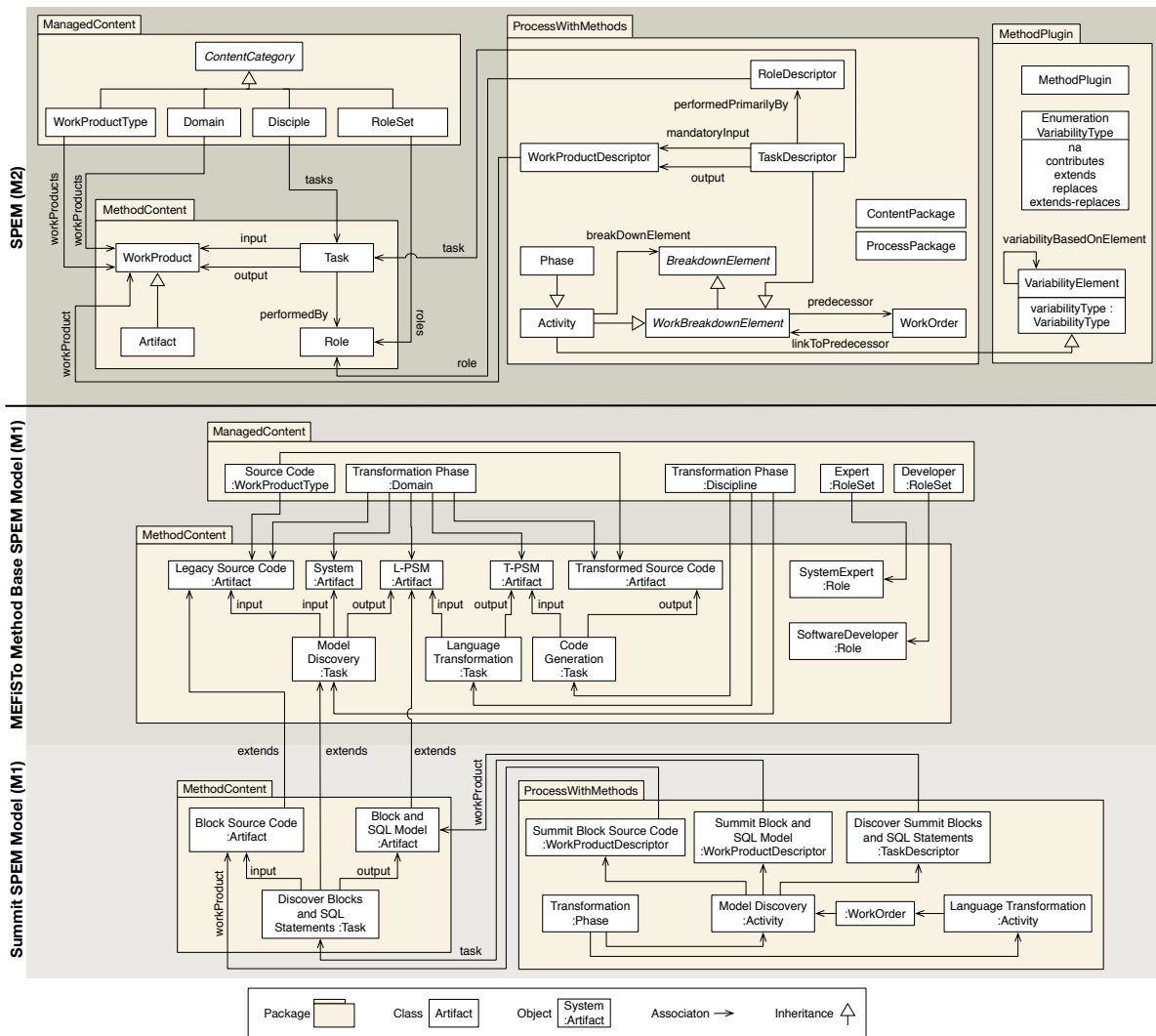


Figure 5.20 Model of the transformation method introduced in Section 5.5.1 in SPEM (excerpt)

the Method Content package. It enables specifying method fragments, like artifacts, activities (called Tasks) or roles.

SPEM follows a clear separation between reusable method content and its use as part of a process within a project. This separation can be seen by the relation between the Method Content and the Process With Method packages. While elements of the former one can be used to define fragments like Tasks, there are no means provided to specify a temporal order between them. This is enabled by elements of the Process With Method package, e.g., by using a Task Descriptor and an associated Work Order element. A Task Descriptor thereby references a Task, so that the same Task can be reused in different processes.

The Method Plugin package provides means to manage whole methods modeled with SPEM and to define relations between them. In this context, a Method Plugin provides

a container for specified elements, it can be used to encapsulate a whole method. Method Plugins can be related to each other, in the sense that one plugin replaces or extends parts of another Method Plugin.

We want to point out that the metamodel shown can differ slightly from the one defined in [OMG08a]. On the one hand, this is due to the fact that the metamodel has been simplified to increase the readability. On the other hand, we use the metamodel that is defined within the Rational Method Composer (RMC). The RMC has been developed by IBM and provides a reference implementation of SPEM [OMG08a, p.22]. The use of the metamodel of RMC is motivated by the fact that we provide an implementation of the described formalization to demonstrate the feasibility.

On the middle layer, the content of the *MEFiSTo Method Base SPEM Model* is shown. The model is an instance of SPEM and specifies the method fragments proposed in Sections 5.3 and 5.4 by using elements of the Method Content package. For example, the *Model Discovery* activity becomes a Task which is related to Artifacts that serve as input or output. Although not shown in the figure it is also associated with a Tool, namely a Parser.

To structure the method fragments, various categories are defined by using elements of the ManagedContent package. For example, a Discipline named Transformation Phase is specified which aggregates all Tasks associated with this phase.

On the lowest layer, the content of a project-specific method is shown, namely the *Summit SPEM Model*. As can be seen, a project-specific method defines its own method fragments as part of the MethodContent package. Thereby, each fragment is an extension of a fragment defined within the *MEFiSTo Method Base SPEM Model*. The extension can be seen as some kind of inheritance [OMG08a, p.139], we use it to model the project-specific customization of a generic fragment. The transformation method specified is an excerpt of the one introduced in Section 5.5.1. In this method, an activity called *Discover Blocks and SQL Statements* is performed. That activity is specified as a Task which is an extension, i.e., customization, of the Model Discovery Task.

In addition to the customized fragments, the project-specific method also defines a process by using elements of the Process With Method package. For example, the Task Descriptor called Discover Summit Blocks and SQL Statements specifies an activity that needs to be performed when enacting the method to transform the Summit legacy system. As can be seen, the Task Descriptor is embedded in a process structure according to the *Language Transformation Pattern*. It is referenced by a ModelDiscovery Activity which describes a super-process and, in turn, is referenced by a Transformation Phase.

This concludes a brief description on how we specify transformation methods in SPEM. In the next section, we discuss the transformation from MIML to SPEM.

5.7.3 Transforming MIML to SPEM

Transformation Methods that are specified using MIML can be automatically transformed into SPEM by executing a set of model transformations rules. The transformation of an exemplary model is shown in Figure 5.21.

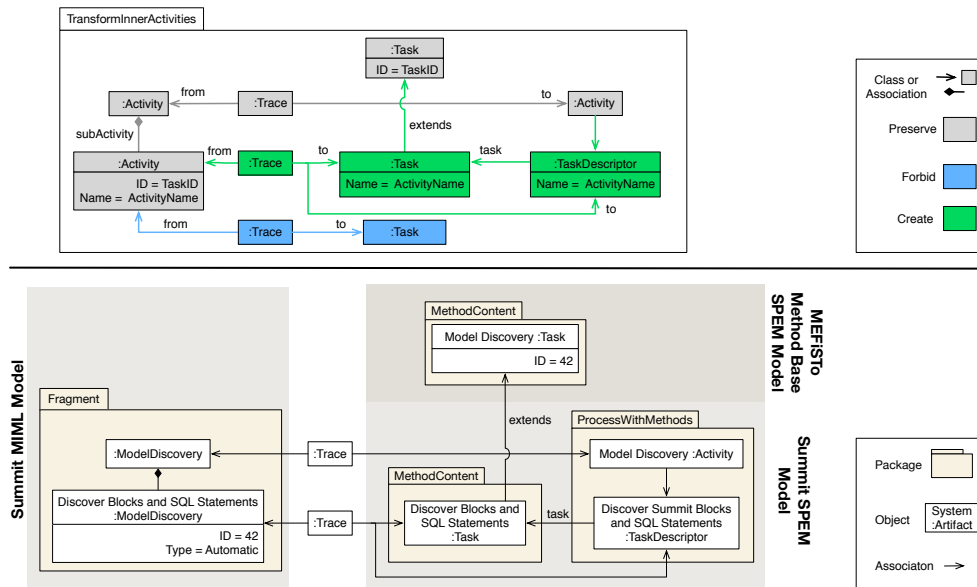


Figure 5.21 Transforming a MIML model to SPEM (bottom) using model transformations (top)

The set of model transformation rules we defined enables transforming MIML models into SPEM, according to the schema described in the previous section. In the upper part of the figure, an excerpt of a single rule is shown which transforms an Activity specified in MIML to a Task and a corresponding Task Descriptor in SPEM. More precisely, the transformation of an *inner* Activity is shown, as only those are mapped to Tasks, while *outer* Activities are mapped to the equally named language construct in SPEM (cf. Section 6.4.1). In the lower part of the figure, the rule is applied on a concrete instance. The models shown have been introduced in the previous two sections.

Based on the example, it can be seen how the Model Discovery activity is specified in MIML and SPEM respectively. The example suggests, why MIML reduces the complexity of specifying a transformation method. On the one hand, MIML provides explicit types for each method fragment proposed on the M2 layer. In contrast, SPEM specifies this information on the M1 layer, requiring an interpretation of the *extends* relation. Among other things, this complicates the expression of constraints. On the other hand, the MIML model does not distinguish between reusable method content and processes, which reduces the amount of modeling elements required. Considering the separation when developing a transformation

method would only be beneficial if we expect to reuse fragments within one project. However, we only expect to reuse fragments between projects. Therefore, we realize the separation as part of the model transformation.

This concludes a description how we formalize transformation methods by using an intermediate representation called MIML that gets transformed to SPEM by model transformations.

5.8 Summary

The MEFiSTo framework enables the modular construction of transformation methods by reusing methodological knowledge stored in a method base. In this chapter, we introduced the content of the method base, namely a set of method fragments and method patterns.

First, we discussed requirements related to the method base in Section 5.1 and refined its structure in Section 5.2. We described how we derived the method fragments proposed and introduced a schema to characterize the method patterns.

In Sections 5.3 and 5.4, we proposed a set of method fragments that are stored in the method base of MEFiSTo. In MEFiSTo, a method fragment is an atomic constituent of a method, namely an artifact, activity, role or tool. We classified the fragments based on the phase they belong to, namely the tool implementation or transformation phase (cf. Section 4.1.3).

In Sections 5.5 and 5.6, we proposed a set of method patterns that are stored additionally in the method base of MEFiSTo. In MEFiSTo, a method pattern encodes methodological knowledge of transformation methods, i.e., each pattern prescribes a transformation strategy by indicating which method fragments to customize and how to assemble them. We gave detailed examples for the proposed patterns and discussed their characteristics.

In the last Section 5.7, we described a formalization of the proposed content of the method base. We introduced an intermediate language called MIML to specify transformation methods. Models of this language can automatically be transformed into SPEM by executing a set of model transformation rules.

In the next chapter, we will go into detail on the method engineering process of MEFiSTo. The process guides the systematic development of transformation methods, using the content of the method base that has been introduced in this chapter.

MEFiSTo Method Engineering Process

In the previous chapter, the content of the method base as part of the MEFiSTo framework has been described. In this chapter, we introduce the corresponding method engineering process that uses the method base to develop and enact situation-specific transformation methods. First, we discuss requirements that specifically address the method engineering process in Section 6.1. In Section 6.2, we give an overview of the core activities of the process. Essentially, each activity can either be assigned to one of two disciplines: method development or method enactment. We introduce method development activities in Sections 6.3 and 6.4, while activities belonging to the method enactment discipline are introduced in Sections 6.5 and 6.6. The findings of this chapter are summarized in Section 6.7.

6.1 Requirements

Before introducing the method engineering process, we discuss related requirements. First, we want to discuss the main purpose of the method engineering process of MEFiSTo. In general, there exist three different concerns that Situational Method Engineering (SME) processes may address, namely *method development*, *method tailoring*, and *method enactment* [HS+14, p.20]. Method tailoring is the discipline to adapt an already existing method to the current situation, which should not be necessary when developing a method from scratch but only when an existing method is reused [HS+14, p.169]. As we do not consider reuse, the method engineering process of MEFiSTo does not cover method tailoring.

Activities belonging to the method development discipline are concerned with developing a situation-specific method in the form of a specification. We aim to provide detailed guidance for this endeavor mainly for two reasons: First, developing a method in MEFiSTo not only requires assembling but also customizing method fragments stored in the method base, which is

a complex task. Second, a rigorous specification is an essential input to support the subsequent method enactment.

Method enactment is concerned with the performance of the method as specified [HS+14, p.223], which can, for example, be supported by using a process engine that assigns task to participants [DF94]. However, while we address the procedure of enacting a method, we do not focus on this discipline, leaving support open for future work. Therefore, Requirement 1 claims that the primary focus of the method engineering process of MEFiSTo is the development of a transformation method.

Process Requirement 1 (*Focus on method development*)

The development of a transformation method is the primary focus of the method engineering process of MEFiSTo.

When specifying a method, an important characteristic of the specification is its granularity. Thereby, we need to distinguish two views on the granularity: the technical and functional viewpoint. Several works address the technical granularity of a method that is determined by the underlying metamodel [HSGP11]. As we intend to use the SPEM metamodel as defined by the OMG (cf. Requirement 5, page 68), we do not address this kind of granularity.

However, even when using a fixed metamodel like SPEM, the functional granularity can vary. This refers to the level of detail in which the specification determines a functional aspect. Related to a transformation method, we can, for example, specify one activity to transform the whole legacy system, or one activity for each language element of the programming language used. In the latter case, the specification will be more fine-granular. As another example, we can decide to guide reimplementations tasks in detail, or not, leaving more degrees of freedom during the enactment. We do not aim deciding on the granularity a priori but leave this open for an expert to decide during method development. Therefore, Requirement 2 claims that the functional granularity of the transformation method shall be adaptable.

Process Requirement 2 (*Adaptable specification granularity*)

The functional granularity of the transformation method specification shall be adaptable.

Developing a situation-specific transformation method is a complex and knowledge-intensive task which requires performing various decisions. Related to MEFiSTo, it needs to be decided, for example, which method patterns to apply. This decision can influence the overall efficiency and effectiveness of the underlying modernization project.

To avoid poor decisions, we could aim to develop a solution concept that enables finding the best solution for each decision point. For example, we could determine the efficiency and effectiveness of each pattern and select the best one automatically. However, such a quantifica-

tion is an open research challenge, e.g., determining the effectiveness of a transformation in terms of software quality [Pan+14]. In general, we assume the quantification to be a complex and error-prone task, wherefore it is not our objective.

Instead, we aim to center our solution concept, i.e., MEFiSTo, around decisions performed by experts. This is based on our observation in practice that nowadays, situation-specific transformation methods are developed by experts, too. However, decisions are often not comprehensible, i.e., the decision rationale is missing. We intend to improve the status quo by the framework defined in this thesis. In particular, we aim to enable a systematic exploration of the situational context to make informed and comprehensible decisions, possibly enabling quantification in subsequent projects. Therefore, Requirement 3 claims that decisions performed during the development of a transformation method shall be comprehensible.

Process Requirement 3 (*Comprehensibility of decisions*)

The decisions performed during the development of a transformation method shall be comprehensible.

6.2 Overview of the Process

In this section, we give an overview of the method engineering process as part of the MEFiSTo framework. The purpose of the process is to guide the systematic development and enactment of a transformation method specification. In Section 4.1.2, we briefly introduced the core activities of each phase of the method engineering process. These activities are also shown in Figure 6.1. In this section, we refine each activity by describing its emphases.

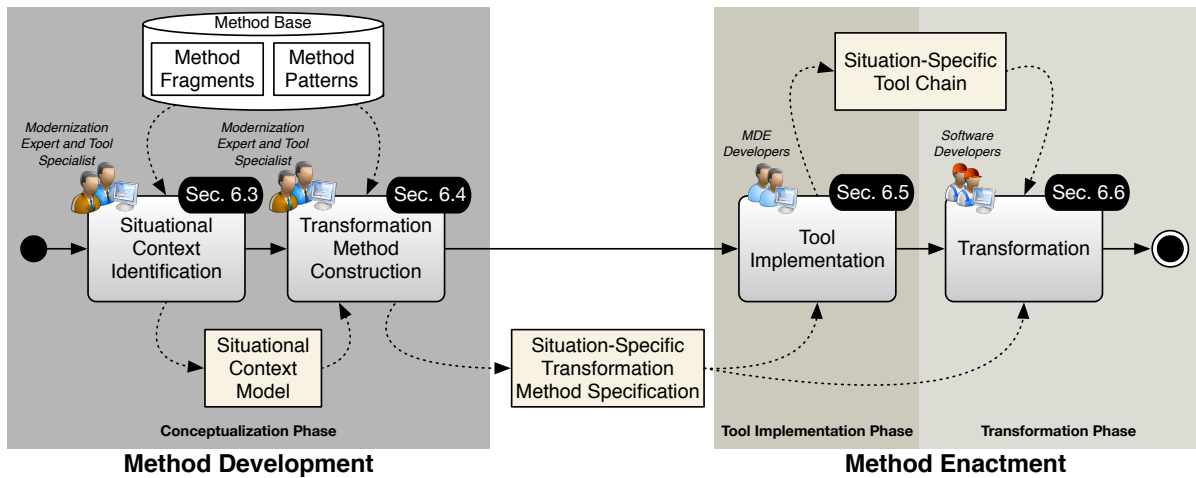


Figure 6.1 Core activities of the method engineering process of MEFiSTo

The first core activity is named *Situational Context Identification*, it is described in detail in Section 6.3. Its purpose is to systematically discover the situational context of the software modernization project. Related to this core activity, we set an emphasis on the following two aspects. First, we focus on decomposing the legacy system into distinct parts. This is based on our observation in practice that applying a single method pattern on a legacy system may not yield a situation-specific transformation method. In contrast, it is often desired to apply different patterns on different parts of a legacy system. To facilitate this, we aim to guide the development of a model that describes a decomposition of a legacy system. Second, we focus on systematically identifying and modeling the influence factors on the efficiency and effectiveness of a method pattern. As exemplified in Sections 5.5 and 5.6, it is essential to be aware of influence factors when assessing the suitability of a pattern. We aim to guide their systematic identification and modeling, to enable comprehensible decisions during the development of a transformation method (cf. Requirement 3, page 143).

The second core activity is named *Transformation Method Construction*, it is described in detail in Section 6.4. Its purpose is to guide the pattern-based development of a transformation method. Related to this core activity, we set an emphasis on the following two aspects. First, we focus on systematically customizing the method fragments. As can be seen based on Sections 5.3 and 5.4, the fragments stored in the method base are not associated to a specific technology but described generically. As a result, they need to be customized when developing a situation-specific method. We aim to guide their systematic but adaptable customization (cf. Requirement 2, page 142). Second, we focus on integrating different method patterns. By simply applying different patterns on different parts of the legacy system, various unconnected methods would result. To avoid this, we aim to guide the systematic integration of patterns.

The first two activities of the method engineering process form the *Conceptualization Phase* which is concerned with the development of a transformation method specification. An example for a specification can be seen in the upper part of Figure 6.2. The specification is a model that formally specifies the developed transformation method by defining which activities to perform or artifacts to generate in order to realize the transformation. In the example, the model is based on the MEFiSTo Intermediate Modeling Language (MIML) (cf. Section 5.7.1), whereby we use the Software and Systems Process Engineering Metamodel (SPeM) in the general case.

While the first two core activities are concerned with the development of a transformation method specification, the last two core activities are concerned with enacting it. Note that the manifestations of both activities are defined by the developed specification, i.e., each activity can be seen as being parameterized over the developed specification.

The third core activity is named *Tool Implementation*, it is described in Section 6.5. Its purpose is to enact those parts of the specification that describe the development of required

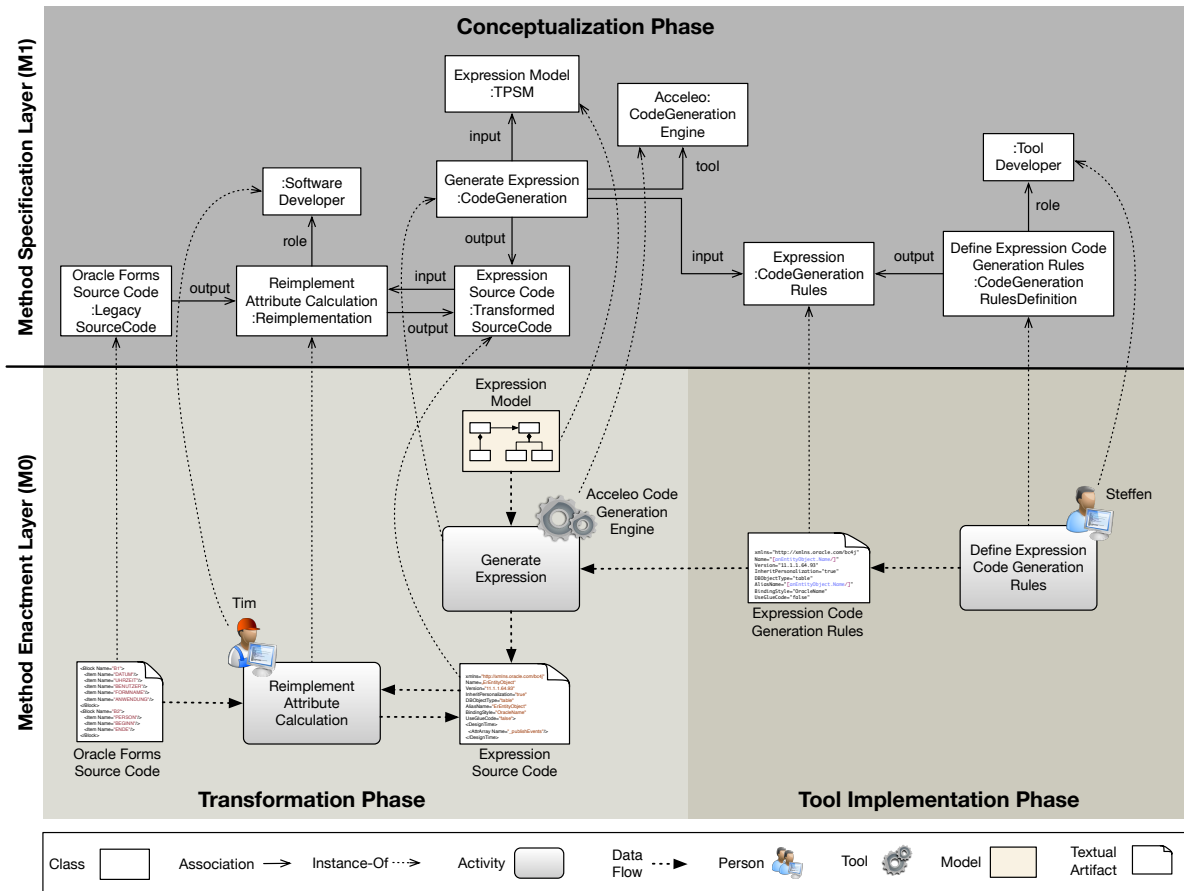


Figure 6.2 Exemplification of the relationships between the different phases of the method engineering process of MEFiSto

tools. Those tools are required to automate (parts of) the transformation. This endeavor is exemplified in the bottom right of Figure 6.2. Related to this core activity, we focus on discussing required capabilities of a tool infrastructure. We assume that such a generic tool infrastructure is in place when using the MEFiSto framework. It shall provide a foundation for the development of project-specific tools.

The last core activity is named *Transformation*, it is described in Section 6.6. Its purpose is to enact those parts of the specification that describe actual transformation of a legacy system, using the tools developed. This endeavor is exemplified in the bottom left of Figure 6.2. Related to this core activity, we focus on discussing challenges when enacting the activities incrementally. In practice, an incremental transformation is required to control the complexity arising from transforming a large legacy system.

Subsequently, we describe the sub-activities of each core activity separately, using the running example introduced in Section 4.3. An integrated view on all activities can be found in the appendix (cf. Section B). We begin with the activity called *Situational Context Identification*.

6.3 Situational Context Identification

In this section, we refine the *Situational Context Identification* activity, whose purpose it is to systematically discover the situational context of the modernization project. It consists of two sub-activities that are shown in Figure 6.3.

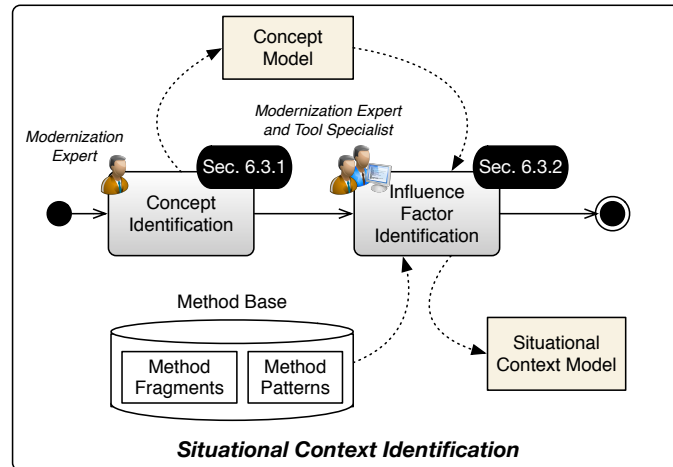


Figure 6.3 Situational context identification process

In the beginning, the *Concept Identification* activity is performed by a person in the role of a *Modernization Expert*. Its purpose is to describe the legacy system to transform as a set of concepts, effectively decomposing it into distinct parts. The resulting *Concept Model* serves as an input for the subsequent *Influence Factor Identification* activity. It is performed by persons in the role of a *Modernization Expert* and *Tool Specialist*, respectively. For each concept, they systematically identify a set of suitable method patterns and related influence factors that affect their efficiency or effectiveness. Taken together, the resulting *Influence Factor Model* and the previously defined *Concept Model* form the *Situational Context Model*.

We describe each sub-activity separately, beginning with the *Concept Identification* activity.

6.3.1 Concept Identification

To model a decomposition of a legacy system into distinct parts, we use principles of the well-established technique of *Concept Modeling* [KNE92]. In particular, we reuse the way in which the functionality of a software system is described and transfer this way to the domain of software modernization. For a detailed description of how to model systems by concepts we refer to the foundations of this thesis, described in Section 2.3.4. Subsequently, we provide a brief recap before describing how to transfer the approach.

In [KNE92], the functionality of a software system is described by a set of *concepts*, whereby different types are distinguished. The first distinction is made between *language concepts* and *abstract concepts*. Language concepts (cf. Notation 8, page 33) directly correspond to syntactic elements of the programming language used and are defined by them. In contrast, abstract concepts represent language-independent ideas of computation and problem solving principles (cf. Notation 9, page 33). They can be further classified into *architectural concepts* or *programming concepts*, whereby the latter ones include general coding strategies, data structures and algorithms (cf. Notations 10 and 11, page 34). Usually, abstract concepts are not directly associated to one syntactic element but to multiple ones. Therefore, they might not be related to consecutive parts of the source code but can be scattered across it.

Concepts can be related to each other by two types of relations, namely *is-a* and *consists-of* relations. The *is-a* relation is used to express a hierarchy between different concepts, while *consists-of* is used to express dependencies between concepts. Intuitively, if the identification of a concept requires identifying others first, then that concept consists of the others.

When transferring the idea of concept modeling to the domain of software modernization, we can distinguish three classes of concepts. These classes are shown in Figure 6.4.

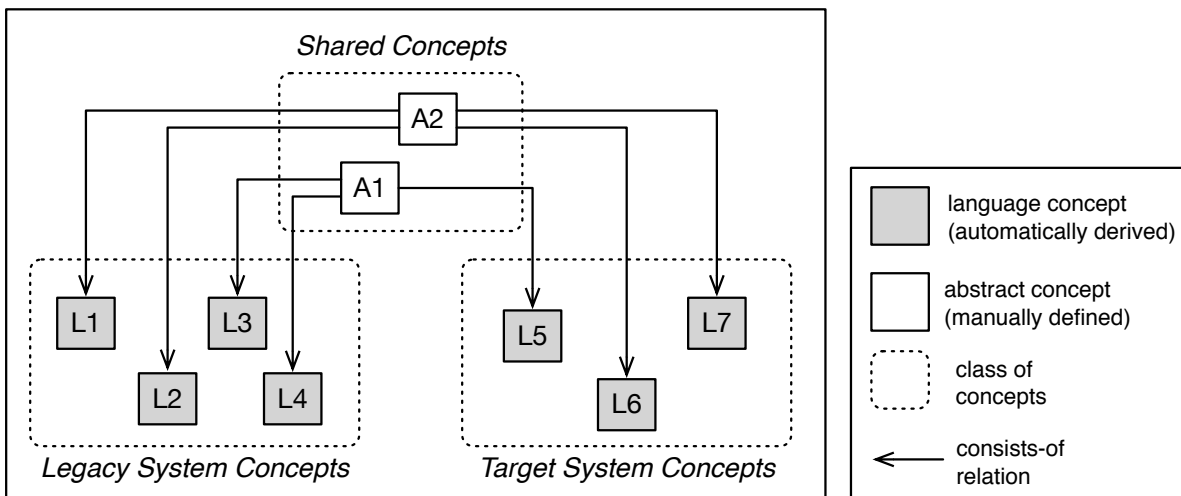


Figure 6.4 Classes of concepts involved in a software modernization scenario

The first two classes are named *Legacy System Concepts* and *Target System Concepts*, they are associated with the corresponding systems. Both classes contain a set of language concepts. Related to the legacy system, these concepts are determined by the language elements that are already used by the implemented system. In contrast, language concepts of the target system are those that will be used after the transformation. In addition, both classes may contain abstract concepts that are specific to one system. In this context, we mean that they cannot be expressed in the other environment. For example, the legacy system might enable to realize graphical

user interfaces (GUIs), but the target environment does not. Instead, the target environment could be centered around text-based user interfaces, maybe even missing a mouse as an input device. In this case, the abstract concept of GUIs is specific to the legacy environment. This can also be the case the other way around. In general, in such an instance one environment is less expressive than the other.

According to this definition, a transformation of an abstract concept that is specific to one environment is not possible, i.e., the functionality cannot be preserved. We neglect such cases and focus on concepts that are expressible in both environments, i.e., which belong to the class of *Shared Concepts*. In this thesis, we define them as follows:

Notation 13 (*Shared Concept*)

A shared concept is an abstract concept of the legacy system that can be realized in the target environment.

To describe the functionality of a legacy system, one or multiple shared concepts can be used. Taken together, these concepts form the *Concept Model*. In this thesis, we define a concept model as follows:

Notation 14 (*Concept Model*)

The concept model is a directed, acyclic and connected graph. Its nodes are shared concepts, while edges between them represent is-a or consists-of relations.

Running Example

Before going into details on the characteristics of a concept model, we provide an exemplary model based on the running example of this thesis. The model is shown in Figure 6.5, it specifies most of the functionalities described in Section 4.3.

We propose to use *Concerns* within the concept model that contain concepts. Concerns can be seen as some kind of packaging mechanism to aggregate concepts that belong together. As illustrated in Figure 6.5, we defined two concerns for the running example, namely *Model* and *Modularization*. Concepts contained in the former concern are related to data-access functionalities realized within the Summit application, while concepts of the latter concern are related to the overall architecture of the application.

The *Model* concern contains only programming concepts. In the left part, concepts related to the *table-based data access* are modeled. As described in detail in Section 5.5.1, the application implements an internal representation of *Tables*. We distinguish two types of them: those that are explicitly specified by Blocks (*Primary Table*) and those that are implicitly referenced by imperative source code (*Foreign Table*). This type-hierarchy is modeled by *is-a* relations. Each

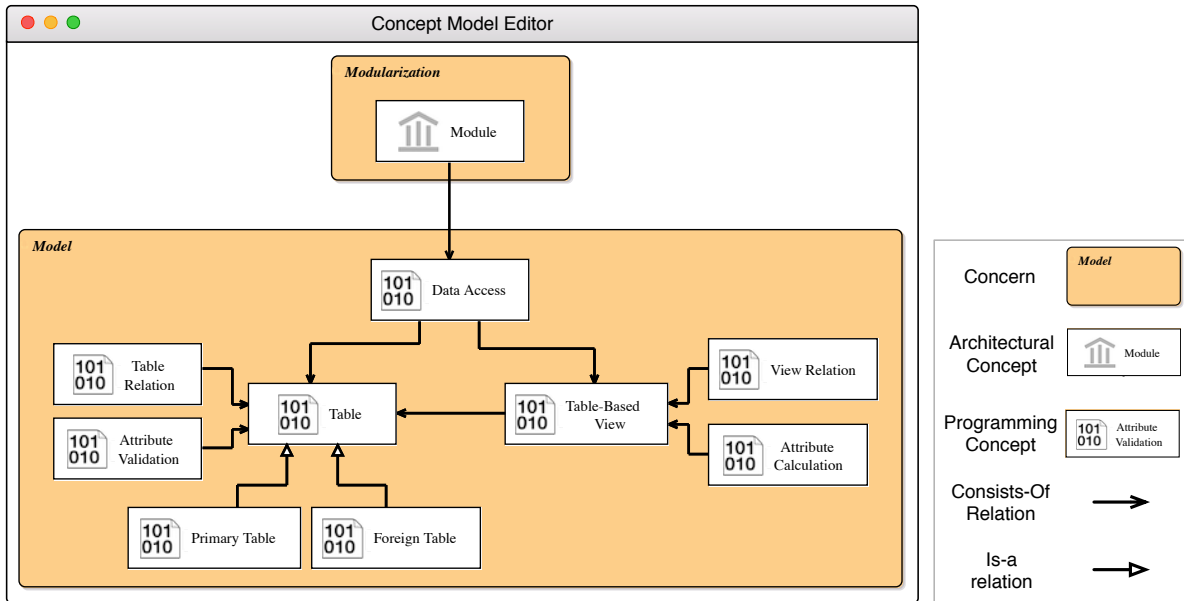


Figure 6.5 Concept model of the running example, specified in MIML

table contains a set of attributes which can be associated to validation rules (cf. Section 5.5.2). This is modeled by a *consists-of* relation, i.e., the *Attribute Validation* concept consists of the *Table* concept whose attributes are validated by rules. In addition, tables can be related to each other by foreign-key relations (*Table Relation*)

In the same way, concepts related to the *view-based data access* are modeled in the right part. As described in detail in Section 5.5.3, Summit implements an internal representation of views that are based on tables (*Table-Based View*). Some views contain attributes that are calculated dynamically (*Attribute Calculation*). Views can be related to each other by master-detail relationships (*View Relation*).

We want to point out that Figure 6.5 actually shows a screenshot of a graphical editor that has been implemented as part of this thesis in *Eclipse Sirius*¹. The editor enables defining a concept model interactively by providing a view on an underlying MIML model.

Characteristics of Concept Models

While the running example gives an intuitive idea of a concept model, we assume that specifying such a model is a hard but critical task. To guide this endeavor, we discuss certain desired characteristics of the model based on our observations in practice. As the model is a starting point to derive a transformation method, the characteristics can also be seen as quality criteria for the resulting method. Therefore, we classify them based on the quality characteristics

¹<https://eclipse.org/sirius/> (accessed March 22th, 2016)

for methods described in [Har97, pp.227-243]. In particular, we discuss the *Completeness* of the model, as well as a three characteristics related to its *Soundness*, namely *Abstraction*, *Connectivity* and *Cycle-Freeness*.

Completeness We consider a concept model to be complete, if its concepts fully describe the functionality of the legacy system. In MEFiSTo, transformation methods are constructed by selecting method patterns for each concept identified. As a result, if the concept model is incomplete, the transformation method becomes incomplete, too. This implies that some functionality of the legacy system is not transformed.

Validating the completeness is the task of the modernization expert. However, since the concept model is based on the work of [KNE92], it is possible to give a necessary but not itself sufficient criteria of completeness. In particular, this work demonstrates how to define abstract concepts by so called *term plans* which can be automatically recognized by a corresponding engine. Intuitively, term plans are formal descriptions of how to recognize instances of a concept. When providing such plans, it would be possible to detect instances of concepts and therefore validate that a concept model covers the complete source code. Note that this would not ensure completeness as it is only a sign of syntactical but not semantical completeness. However, in this thesis we do not follow this direction by rely solely on the expert.

Abstraction (Soundness) A concept model can describe a software system on varying level of abstraction. Related to the Summit application, we proposed the set of abstract concepts shown in Figure 6.5 to describe its functionality. However, we could have just used one abstract concept called *Summit* to describe its functionality. Although this is an edge case, it should give an idea of varying granularity.

Choosing the level of abstraction is the task of the modernization expert, it will at least have two implications. First, it determines the *granularity* of the method which directly depends on the amount of concepts specified. This is due to the fact that a method pattern gets chosen for each concept separately, leading to a customization of a set of method fragments. Therefore, specifying more concepts leads to more method fragments, i.e., a more detailed method specification. We foresee two main reasons for using a fine-granular set of concepts. On the one hand, it enables choosing different patterns for different functionalities of the legacy system. This enables adapting the method flexibly to the situation encountered. On the other hand, it enables providing detailed guidance. As each method fragment specifies a part of a transformation method, the more method fragments are used, the more detailed the method specification becomes.

Second, if we consider the concept model as some kind of specification, the ability to choose the level of abstraction enables adapting the functionality to preserve. As an example, consider the concept model shown in Figure 6.6. In this example, we assume that the legacy system uses text-based user interfaces, while we aim to use graphical user interfaces in the target system. Both realizations are modeled by environment-specific abstract concepts, while the shared concept abstracts from these technology-specific realizations. Instead, it models the intersection of the functionality, i.e., the definition of user interfaces independent of the way in which they are realized.

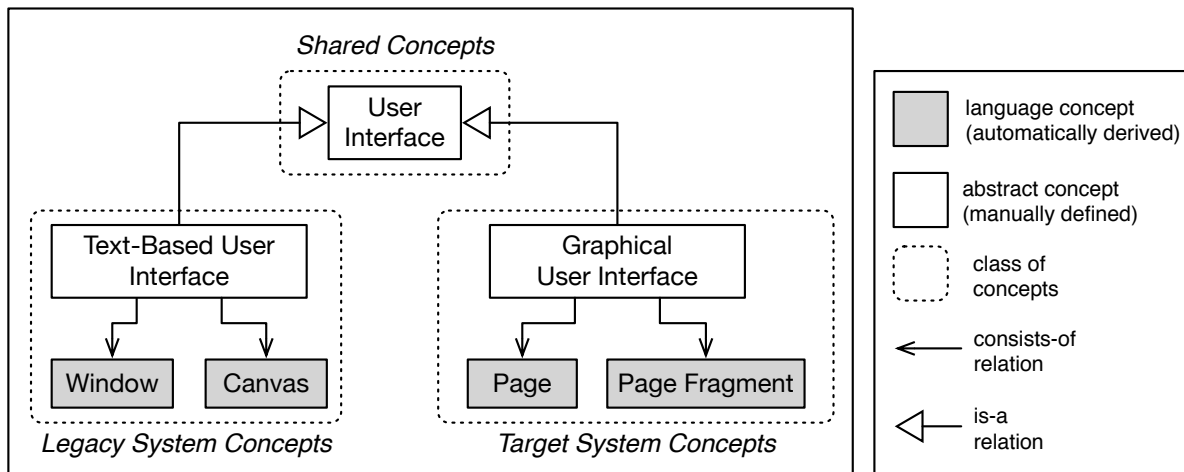


Figure 6.6 Using concepts to abstract from technology-specific realizations

Assuming that the *User Interface* concept specifies the functionality to preserve, we intentionally lose some kind of information during the transformation. Here, we aim to lose the way in which the functionality has been realized. We only aim to preserve the interface itself, e.g., the information shown or input fields provided. Then, a user which uses this interface to fulfill a task, gets the same information or can provide the same input in both systems, but maybe in a different way. We assume that such an abstraction is essential for enabling a true adaption of the functionality to the new environment. Note that the level of abstraction chosen can influence our perception of when we consider the legacy system and the target system to be functional equivalent. As discussed for the example, the user interface may look different after the transformation, but also enable the same interaction. We assume that this is tolerable in a software modernization scenario.

Connectivity (Completeness) We assume that the model itself is *connected*, i.e., the set of shared concepts forms a connected graph. Thereby, connectivity is based on *is-a* as well as on *consists-of* relations. Assume that the concept model contains two unconnected sets of

concepts S_1 and S_2 . Then, there does not exist a relation between the instances of both sets in the legacy system, i.e., no control or data flow dependency. In such an instance, the code of the legacy system is either incomplete, the system consists of two distinct systems or it contains dead code. We neglect those cases.

Cycle-Freeness (Soundness) We assume that the model itself is *acyclic*, i.e., the graph formed by the set of concepts does not form a cycle. On the one hand, this enables using the dependencies in the model to derive the order in which the functionalities are transformed. If a cycle exists, we cannot decide which functionality to transform first. On the other hand, this supports an incremental definition of the method as concepts can be gradually added.

Target-Driven Concept Identification Process

So far, we gave an idea of what concept models are. To guide their specification, we refine the concept identification activity (cf. Figure 6.3, page 146). To do so, we first describe our perspective on this endeavor, before introducing the actual process.

Using concepts to describe functionalities of software systems is common in the field of software reengineering [RW02]. In general, two different strategies can be distinguished to identify concepts: *bottom-up* and *top-down*. When using a *bottom-up* strategy, the starting point would be the source code of the legacy system which needs to be iteratively condensed to derive abstract concepts. We assume that using such a strategy would increase the risk of preserving legacy concepts, preventing an adaptation of the system to the new environment. Therefore, we suggest to use a *top-down* strategy instead.

When using a *top-down* strategy, hypotheses about the functionality are generated first and evaluated subsequently [KR91]. A hypothesis in our context is a concept, which is evaluated by searching for so-called *beacons* [Bro83] in the source code, i.e., structures that give a hint on its presence. Most notably, experts can create such hypotheses solely based on their experiences without needing to review the source code [MV93]. Intuitively, experts can guess the functionalities that systems of a certain domain typically implement.

Based on this perspective, we refined the concept identification activity into the process shown in Figure 6.7. The proposed process can be seen as being *target-driven*, meaning that the desired outcome of the transformation drives the identification of concepts. This already becomes clear by the first activity named *Concern Identification based on Target Architecture*. Its purpose is to identify a set of concerns to provide a coarse-grained structuring of the concept model. The activity is target-driven as we identify concerns based on the desired structure of the resulting system, i.e., the target architecture. Related to the running example whose concept model is shown in Figure 6.5, it can be seen that we identified two concerns, namely

Model and *Modularization*. In the target system, we intend to realize the data access by ADF Business Services and ADF Model components (cf. Section 3.1). By abstracting from technology-specific terminology we aggregate the concepts of both layers in the shared concern named *Model*. The concern named *Modularization* does not directly relate to any component in the target system but shall contain concepts related to the overall architecture of the system.

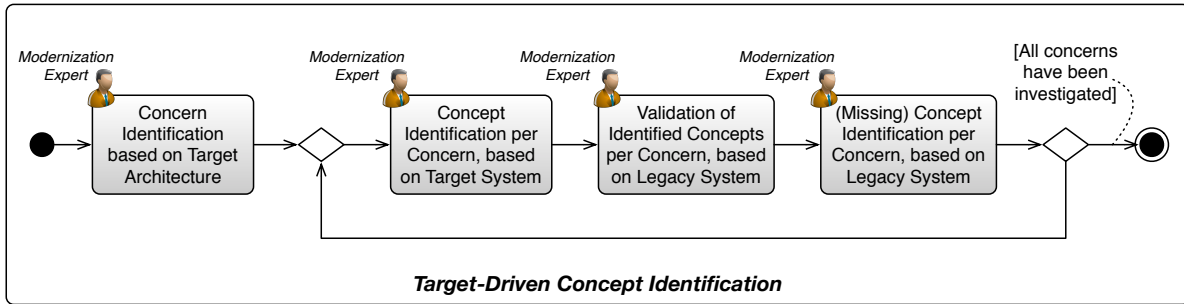


Figure 6.7 Target-driven concept identification process

The remaining activities are performed repeatedly for each concern, until all concerns have been investigated. First, concepts for the current concern are identified, i.e., hypothesized. This can be based solely on the experience of the expert or by evaluating supporting materials like development tutorials. Related to the example, Oracle provides a tutorial on how to develop Business Services which, among other things, proposes to create `EntityObjects`². By abstracting from technology-specific terminology, we derive the *Table* concept which represents the implementation of internal representations of *Tables*.

Second, the identified concepts are validated by performing a superficial analysis of the legacy system. This is necessary as the concepts have so far been identified without considering the legacy system, i.e., it is not said that the legacy system actually realizes the concepts. Related to the example, an expert can quickly validate that the Summit application realizes the *Table* concept by searching for one *Block* that uses a database table as a data source.

Third, the legacy system is analyzed to identify additional concepts, i.e., a concluding, legacy-driven identification takes place. This activity enables adding concepts that are specific to the legacy system. Related to the example, the differentiation between *Primary* and *Foreign Tables* results from this activity. Those concepts represent different ways on how the Summit application realizes internal representations of tables (declarative vs. imperative). Distinguishing those concepts is not relevant for the target system but can be relevant to define a situation-specific method, as the complexity to transform each concept can differ.

²http://docs.oracle.com/cd/E18941_01/tutorials/jdtut_11r2_55/jdtut_11r2_55_1.html (accessed March 22th, 2016)

This concludes the description of the target-driven concept identification process which guides the definition of a concept model. Subsequently, we describe how we formalize concept models in MEFiSTo.

Formalization in MIML

To formalize concept models, we extend the MEFiSTo Intermediate Modeling Language (MIML) by a package called *Concept* that contains required language elements. The package is shown in Figure 6.8.

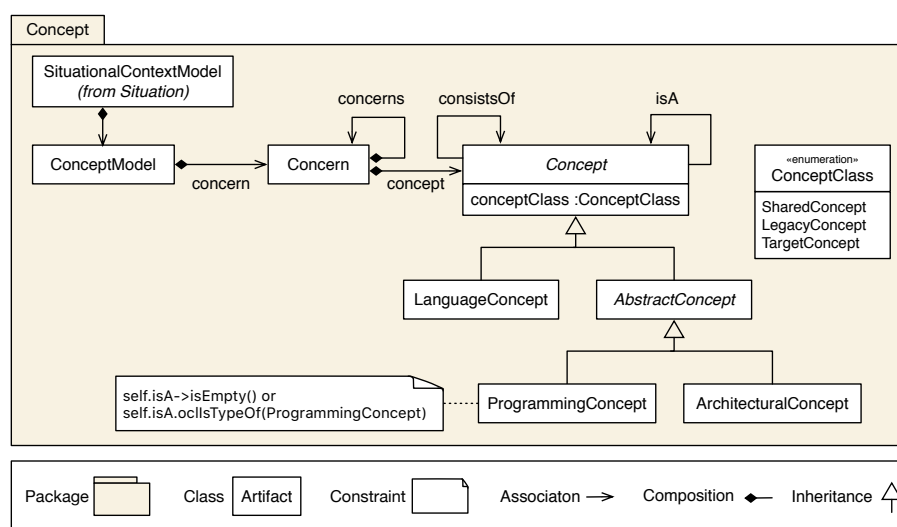


Figure 6.8 Excerpt of the MIML metamodel to formalize concept models

As can be seen in the upper left of Figure 6.8, we consider the *Concept Model* to be a part of the *Situational Context Model*. The latter model is the overall output of the *Situational Context Identification* activity (cf. Figure 6.3, page 146). The *Concept Model* can contain a set of *Concerns* which in turn can contain sub-*Concerns* and a set of *Concepts*. The different types of *Concepts* are represented by subclasses. In addition, each *Concept* belongs to one out of three classes, as described in the beginning of this section. Also, *Concepts* can be related to each other by *consists-Of* and *is-A* relations. OCL constraints ensure invariants of concept model instances. For example, the OCL constraint shown defines that the target of an *is-A* relation for a *Programming Concept* can only be another *Programming Concept*.

6.3.2 Influence Factor Identification

So far, a set of concepts have been identified and modeled as a concept model. In MEFiSTo, a transformation method gets constructed by choosing a method pattern for each concept. To

perform an informed and comprehensible decision on which method pattern to use, we need to systematically search for characteristics that influence their efficiency or effectiveness first. This is the purpose of this activity.

We assume that the problem of deciding for a method pattern during the design of a transformation method is comparable to performing architectural design decisions during software development. As the latter area is an established field of research, we aim to reuse principles from it. Subsequently, we first introduce a work that focuses on modeling architectural design decisions, before discussing why and how to transfer its principles to the domain of MEFiSTo.

In particular, we reuse principles defined in the work of [Zim09] which introduces a framework that is centered around architectural design decisions. The work is motivated by the fact that it is common practice to capture and document decisions only retrospectively after they have been performed, leading to incomprehensible, uninformed decisions in the first place and hindering reuse of knowledge [Zim09, p.3-4]. The introduced framework reverses the role of decisions by considering them as the main driver during the architectural design process [Zim09, p.55-61]. To achieve this, the process begins with identifying design issues (*decisions required*) that need to be addressed. For each issue, design alternatives (*potential solutions*) are identified. From the entire set of design alternatives, those are selected that seem to be relevant (*considered solution*). For each alternative of this subset, advantages and disadvantages are identified to perform an informed decision on which one to choose and apply (*outcome*). Separating these concerns is a key factor of the framework.

	[Zim09] Principle	MEFiSTo Principle
Decision Required	Design Issue	Concept
Potential Solution	Design Alternative	Method Pattern
Considered Solution	Design Alternative	Suitable Method Pattern
Outcome	Decision Outcome	Applied Method Pattern

Table 6.1 Transferring principles from [Zim09] to the domain of MEFiSTo

The findings of this work coincides with our observations in practice: The design of transformation methods is a knowledge-intensive task performed by experts. Nevertheless, the design rationale is usually not captured. This leads to the same aforementioned problems, i.e., incomprehensible decisions related to the design of a transformation method and lost knowledge for similar, subsequent modernization projects. Therefore, we aim to improve

this state by including principles of the framework in MEFiSTo. A mapping of the different principles of the framework to the domain of MEFiSTo is shown in Table 6.1.

We put a design issue on the same level with a concept. By this we mean that each identified concept represents a decision that needs to be performed, i.e., how to transform the concept. Following this direction, each method pattern stored in the method base represents a potential solution for the decision by encoding construction guidelines for a transformation method. Note that the type of pattern and the type of concept determines, whether a pattern describes a solution: functionality preserving patterns describe potential solutions for language- and programming concepts, while architectural restructuring patterns describe potential solutions for *architectural concepts* (cf. Sections 5.2 and 6.3.1).

If the entire set of method patterns describes the potential solutions, then those that are assessed as being suitable become considered solutions. When we introduced the method patterns in Chapter 5, we discussed the suitability of each pattern for the example given. Thereby, the suitability was based on identified influence factors that affect the efficiency or effectiveness of a pattern. These factors can constitute positive or negative influences on the resulting transformation method, i.e., they determine advantages or disadvantages of the considered solution. We define the influence factors in this thesis as follows:

Notation 15 (*Influence Factor*)

An influence factor is a characteristic of a software modernization project that has an impact on the efficiency or effectiveness of a transformation method.

We put an emphasis on the influence factors as we assume that the outcome of a decision, i.e., whether to choose a pattern or not, depends on them. In other words, the rationale for choosing a pattern is based on the influence factors present. Therefore, we aim to explicitly capture them as a model, enabling comprehensible decisions based on reusable knowledge. We define the influence factor model in this thesis as follows:

Notation 16 (*Influence Factor Model*)

The influence factor model is a model which describes of a set of influence factors and affected method patterns.

Based on the discussion of the suitability provided in Sections 5.5 and 5.6, we conclude that we can classify the influence factors in a software modernization scenario into four distinct categories. The classification is shown in Figure 6.9.

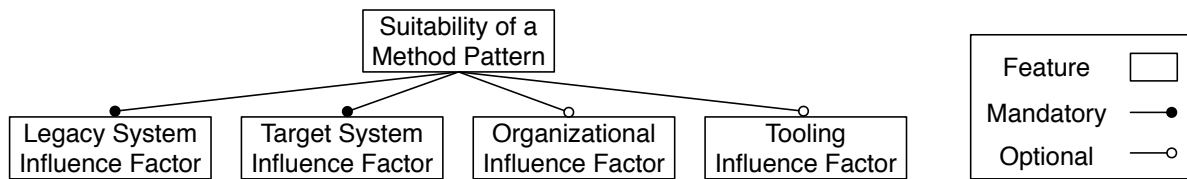


Figure 6.9 Classification of influence factors

Assessing the suitability of method patterns always requires an assessment of the realization of a concept. This applies to the current realization in the legacy system as well as the desired realization in the target system. Therefore, at least one influence factor of each category shall be identified for each method pattern. In addition, method patterns that involve manual activities are subject to organizational influence factors. An exemplary influence factor is the amount of developers available. In the same way, method patterns that encode automated conversions are subject to tool-related ones. For example, the availability of metamodels or parsers falls into this category.

Running Example

Before going into details on how to guide the specification of an influence factor model, we provide an exemplary model based on the running example of this thesis. The model is shown in Figure 6.10, it contains a set of influence factors for the *Attribute Validation* concept that has been exemplified in Section 5.5.2.

	Name	Description	Implication
▼ Programming Concept Attribute Validation			
▼ Language Transformation			
Legacy System Influence Factor	Amount of Validation Rules	The Summit application only contain	An automated transformation will only be eff
Legacy System Influence Factor	Realization of Validation R	Validation rules are realized impera	Reliably distinguishing validation rules from
Target System Influence Factor	Realization of Validation R	Validation rules shall be realized de	As Forms does not support a declarative def
Tooling Influence Factor	PL/SQL Parser	A parser for the language PL/SQL ca	The parser costs 299\$, the license allows cc
Tooling Influence Factor	Coding Convention	Validation rules are consistently loca	The coding conventions enable to narrow d
▼ Conceptual Transformation			
Legacy System Influence Factor	Amount of Validation Rules	The Summit application only contain	An automated transformation will only be eff
Legacy System Influence Factor	Realization of Validation R	Validation rules are realized impera	Reliably distinguishing validation rules from
Target System Influence Factor	Realization of Validation R	Validation rules shall be realized de	As Forms does not support a declarative def
Tooling Influence Factor	PL/SQL Parser	A	
Tooling Influence Factor	Coding Convention	V	
Tooling Influence Factor	Metamodel for Validation R	N	
▼ Reimplementation (chosen)			
Legacy System Influence Factor	Realization of Validation R	V	
Target System Influence Factor	Realization of Validation R	V	
Tooling Influence Factor	Coding Convention	V	
Organizational Influence Factor	Amount of Developers	T	
Organizational Influence Factor	Experience of Developers	Developers are not experienced in d	Reimplementation tasks shall be compreh

Figure 6.10 Influence factors related to the *Attribute Validation* concept of the running example, specified in MIML

As can be seen, we provide a tabular editor to specify influence factors. The editor provides a view on an underlying MIML model. It can be instantiated for each concept identified. In the most left column, the concept for which the view has been instantiated forms the root of a tree. Below that root resides a set of suitable method patterns which have been added by the modernization expert. In the example shown, the set consists of the *Language Transformation*, *Conceptual Transformation* and *Reimplementation Pattern*. Influence factors are specified for each pattern separately. Thereby, the different classes shown in Table 6.9 are distinguished.

Each element of the tree contains various attributes that capture essential knowledge about the decision point. They are based on the ones identified in [Zim09] and [KLV06]. For example, each influence factor is associated with the attributes *Name*, *Description* and *Implication*, as can be seen in the figure. In the example shown, a *Legacy System Influence Factor* named *Amount of Validation Rules* is specified, whereby the description hints that *only few validation rules exist in the application*. This will have an *impact* on patterns that encode an automated conversion, as *an automated conversion will only be efficient the functionality to transform is sufficiently large*. Note that not all attributes are shown. For example, each suitable method pattern is associated with attributes called *Pros* and *Cons* which summarize the advantages and disadvantages when choosing the patterns in a textual form.

Process

To guide the instantiation of an influence factor model, we refine the influence factor identification activity (cf. Figure 6.3, page 146). The resulting process is shown in Figure 6.11.

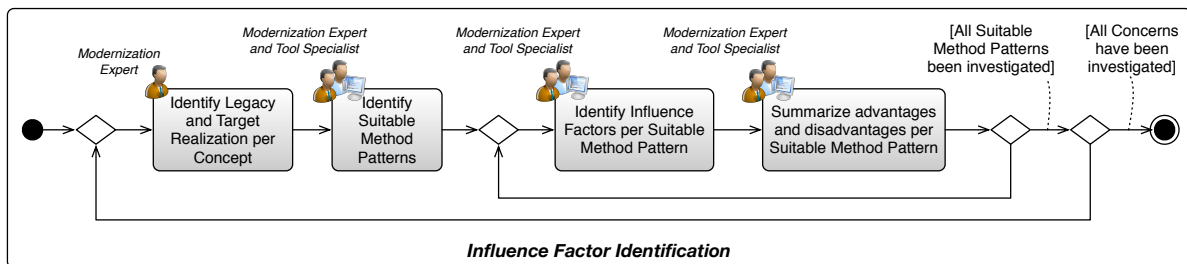


Figure 6.11 Influence factor identification process

The process begins with the identification of influence factors related to the realization of a concept in the legacy system as well as the desired realization in the target system. In contrast to the analysis performed during the concept identification activity, we do not aim for a superficial analysis of a concept but a fine-granular one. For example, this comprises identifying the amount of implementation variants of a concept. The intention to start with the analysis of these particular types of influence factors is motivated by the fact that it is relevant

for all patterns and might allow an early exclusion of some. For example, if the identification indicates that a concept is realized significantly different in both environments, the experts might conclude that patterns which do not consider a conceptual transformation are not suitable. We assume that this can reduce the overall analysis effort.

After a set of suitable method patterns has been identified, influence factors are identified and described for each pattern separately. The identification itself can be systematized by successively considering the method fragments that are prescribed by a pattern, i.e., searching for related influence factors. For example, the *Language Transformation Pattern* prescribes to use the *Model Discovery* fragment which relies on the use of a parser. Based on this connection, we assume that an expert can derive the fact that the availability of a parser is an influence factor. It can be seen that two types of experts are responsible for the identification, namely a *Modernization Expert* and a *Tool Specialist*. The latter expert is especially responsible for influence factors related to tooling.

After the influence factors for one suitable method pattern have been identified, the pattern itself needs to be assessed. Here, we require the experts to provide a textual summary of the advantages and disadvantages when applying the pattern. After those activities have been repeated for all suitable method patterns and concepts, an influence factor model results. Subsequently, we describe how we formalize that model in MEFiSTo.

Formalization in MIML

To formalize influence factor models, we extend the MEFiSTo Intermediate Modeling Language (MIML) by a package called *Influence* that contains required language elements. The package is shown in Figure 6.12.

As can be seen in the upper left of Figure 6.12, we consider the Influence Factor Model to be a part of the Situational Context Model. The latter model is the overall output of the *Situational Context Identification* activity (cf. Figure 6.3, page 146). The Influence Factor Model can contain a set of Influence Factors, whereby we distinguish the different types identified (cf. Figure 6.9, page 157).

The right part of the metamodel is derived from the one defined in [Zim09, p.88]. A Concept is associated with a set of suitable Method Pattern Alternatives. One of them can be chosen to be applied, indicated by the Applied Method Pattern class. Each Method Pattern Alternative can be influenced by Influence Factors. Note that an Influence Factor can be related to multiple Method Pattern Alternatives. For example, the realization of a concept in the legacy system will influence all suitable method patterns. However, the Influence Factor only needs to be specified once and can be linked to all affected Method Pattern Alternatives. OCL constraints are used to ensure invariants. For exam-

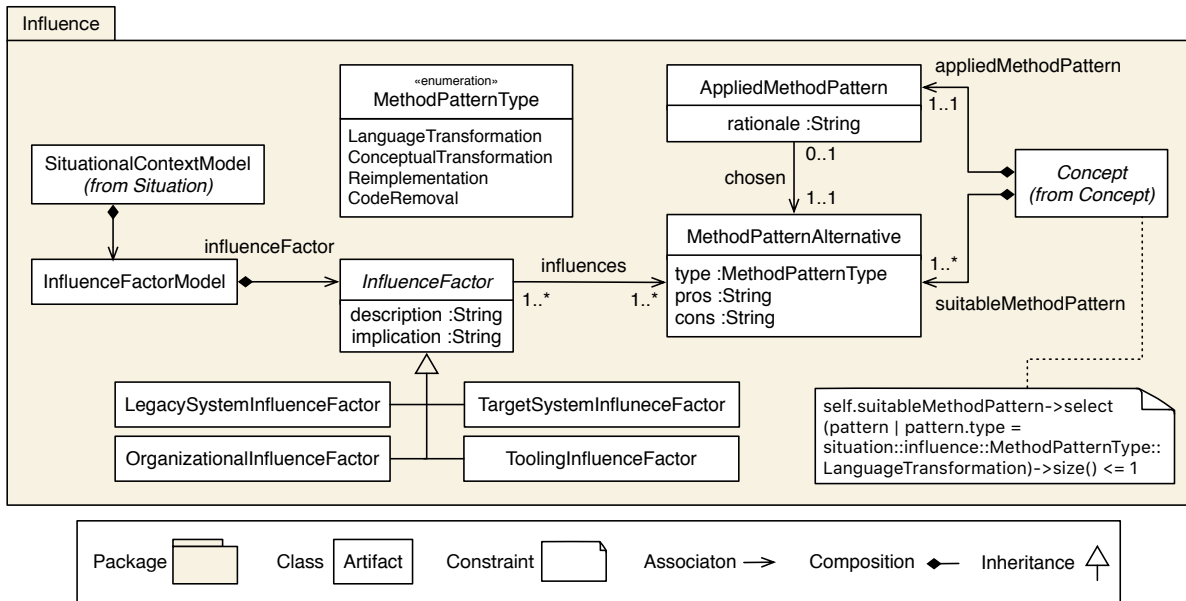


Figure 6.12 Excerpt of the MIML metamodel to formalize influence factor models

ple, the OCL constraint shown defines that each Method Pattern Type can occur only once as suitable Method Pattern Alternative (shown for the *Language Transformation* type).

This concludes a description of the *Situational Context Identification* activity. The first sub-activity named *Concept Identification* outputs a *Concept Model*, while the latter sub-activity named *Influence Factor Identification* outputs an *Influence Factor Model*. Taken together, both models form the *Situational Context Model*. Note that so far, a decision on which method pattern to choose in order to transform a concept has not been performed, i.e., the edge between the Applied Method Pattern and one of the suitable Method Pattern Alternatives has not been established. This is one of the purposes of the next activity, namely the *Transformation Method Construction*.

6.4 Transformation Method Construction

So far, the situational context of the modernization project has been discovered. In this section, we refine the *Transformation Method Construction* activity, whose purpose it is to construct a transformation method by considering the discovered context. It consists of five sub-activities that are shown in Figure 6.13.

In the beginning, the *Method Pattern Selection and Configuration* activity is performed by persons in the role of a *Modernization Expert* and *Tool Specialist*, respectively. The purpose of the activity is to select a method pattern for each concept, i.e., deciding on how to transform

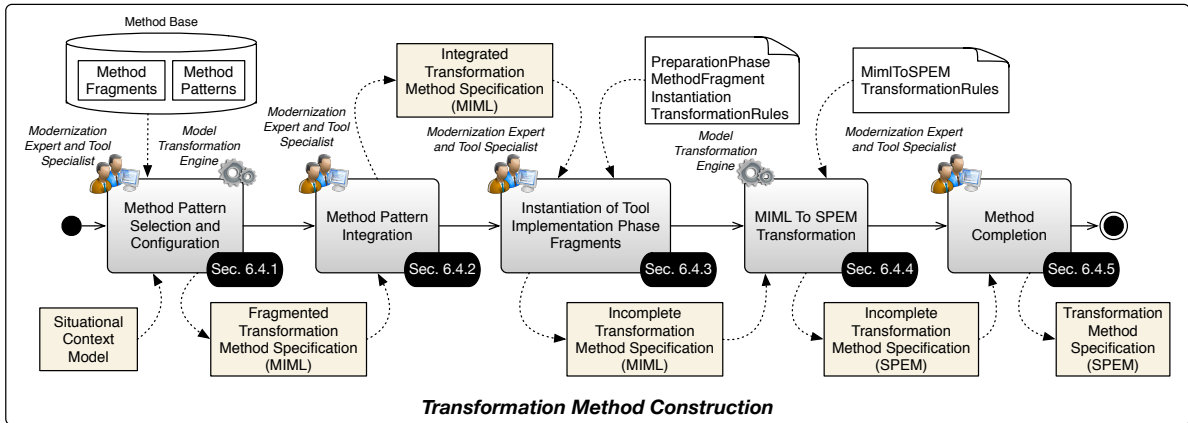


Figure 6.13 Transformation method construction process

each concept. Besides selecting a pattern, it also needs to be configured. Then, for each configured method pattern a set of method fragments is automatically instantiated. At this point, the method fragments that originate from different patterns have not been integrated, wherefore a *Fragmented Transformation Method Specification* results. Integrating the method fragments is the purpose of the second activity. The resulting *Integrated Transformation Method* does only contain customized fragments of the transformation phase but not of the tool implementation phase. Automatically deriving these fragments is the purpose of the third activity. At this point, the transformation method is specified in MIML but gets automatically transformed to SPEM by the fourth activity. The purpose of the last activity named *Method Completion* is to complete the *Method Specification* in SPEM. As MIML does not provide the same extent to specify methods as SPEM, some parts have not been specified until this point.

Subsequently, we describe each sub-activity separately, beginning with the *Method Pattern Selection and Configuration* activity.

6.4.1 Method Pattern Selection & Configuration

So far, knowledge about the modernization project has been identified and persisted in the form of a situational context model. We aim to use this knowledge as a basis to systematically derive a situation-specific transformation method. As a first step into this direction, we propose to select and configure a method pattern for each identified concept. This is the purpose of this activity, its core idea is visualized in Figure 6.14.

The starting point for the *Method Pattern Selection and & Configuration* activity is the situational context model. As described in Section 6.3, it consists of a concept and an influence factor model. The concepts contained in the concept model decompose the legacy system into distinct parts so that each part can be transformed by applying a (possibly different) method

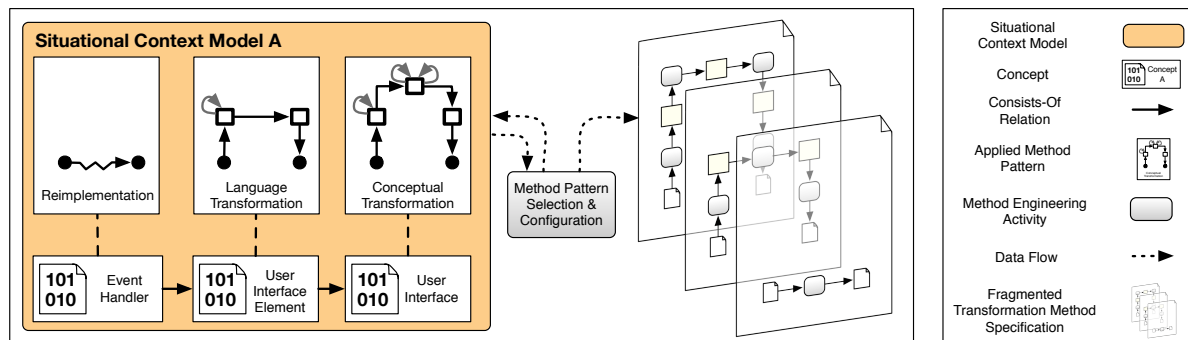


Figure 6.14 Instantiating a fragmented transformation method specification from a situational context model

pattern (cf. Section 6.3.1). To perform an informed decision on which method pattern to choose, a set of suitable patterns has been identified for each concept (cf. Section 6.3.2). Each identified pattern is associated with influence factors that can be interpreted to determine its efficiency and effectiveness.

The activity itself has two emphases: First, a method pattern needs to be *selected* for each concept contained in the situational context model, i.e., a decision on how to transform a concept needs to be performed. The decision shall be based on the situational context, i.e., based on the findings that are summarized by the influence factor model. The result is exemplified in the left side of Figure 6.14. As can be seen, each concept contained in the situational context model is associated with an *Applied Method Pattern* after performing the selection.

Second, a configuration needs to be performed in order to derive an initial transformation method specification. We foresee two means to perform the configuration: First, each applied pattern can be configured (*coarse-granular configuration*). Among other things, this comprises deciding on optional parts for each pattern, e.g., whether to use additional enrichment activities, or not. Based on this configuration, a set of customized method fragments gets automatically instantiated, as prescribed by the pattern. Second, the resulting set of fragments can be configured too, e.g., by renaming or removing existing fragments, or by adding additional ones (*fine-granular configuration*).

When instantiating customized fragments for one concept according to the method pattern applied, the resulting set of fragments will form a horseshoe model. Therefore, a horseshoe model is defined within MEFiSTo as follows:

Notation 17 (*Horseshoe Model*)

A horseshoe model is a model which consists of a set of customized method fragments and conforms to a method pattern. The fragments specify a method to transform a concept.

The overall output of the activity can be seen in the right side of Figure 6.14. It consists of a *fragmented* transformation method specification, i.e., a method specification which can be separated into distinct horseshoe models. We define a fragmented transformation method specification as follows:

Notation 18 (*Fragmented Method Specification*)

A fragmented transformation method specification is a specification that consists of a set of horseshoe models that have not been integrated.

As integrating the different patterns is a comprehensive task, we separately address this concern by the next activity of the method engineering process named *Method Pattern Integration*.

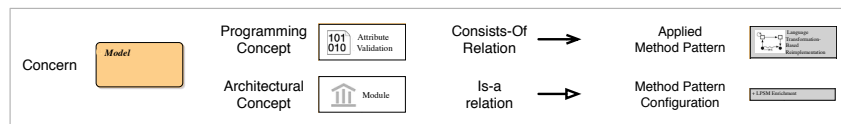
Running example

Before going into details on the selection of method patterns, we exemplify the activity based on the running example that has been introduced in Section 4.3. In Figure 6.15, a concept model is shown for which method patterns have been selected. It is a refinement of the concept model that has been introduced in the previous section (cf. Figure 6.5, page 149).

It can be seen that a method pattern has been selected for each identified concept, whereby each pattern prescribes how to transform the associated concept. For example, it has been decided to transform the *Table-Based View* concept by applying a *Concept Recognition-Based Language Transformation Pattern*. Note that the application of that specific pattern on this concept has already been described in detail Section 5.6.2.

Not only have the method patterns been selected, but also configured in a coarse-granular way. On the one hand, every pattern has some parts that are optional (cf. Sections 5.5 and 5.6). This comprises, for example, an inclusion of intermediate activities, like restructurings or enrichments, or additional activities, like the discovery of an external system. The configuration of those optional parts can be seen in Figure 6.15. Related to the *Table-Based View* concept, it has been decided to include a *System Discovery*. If no optional part has been included, then the *Default Configuration* is used.

On the other hand, although not shown in the figure, notations have been provided for each pattern in order to perform a meaningful instantiation of customized method fragments. For example, when a pattern has been selected that includes the *Model Discovery* fragment, a notation needs to be provided that describes what to discover. If the discovery is related to a concept, then this comprises a notation for the technical realization of the concept in the legacy system. Otherwise, if it is related to the discovery of a system, then this comprises the name of the system. Related to the *Table-Based View* concept, we provided *Block* as the name for



the realization of the concept in the legacy system, while we refer to the system to discover as *Oracle Database*.

The horseshoe view visualizes the customized artifacts and activities, as well as the data flow between them. For each activity, it can be specified whether it shall be performed *automatically*, *semi-automatically* or *manually*. For an automated activity, a tool can be associated from a predefined list, i.e., an activity can be associated with a *parser*, a *model transformation engine*, a *code generation engine* or a *custom* tool. As discussed in Section 5.3, the former three tools are essential ones for model driven tool chains. However, we also foresee to use custom tools,

³<https://eclipse.org/sirius/> (accessed March 22th, 2016)

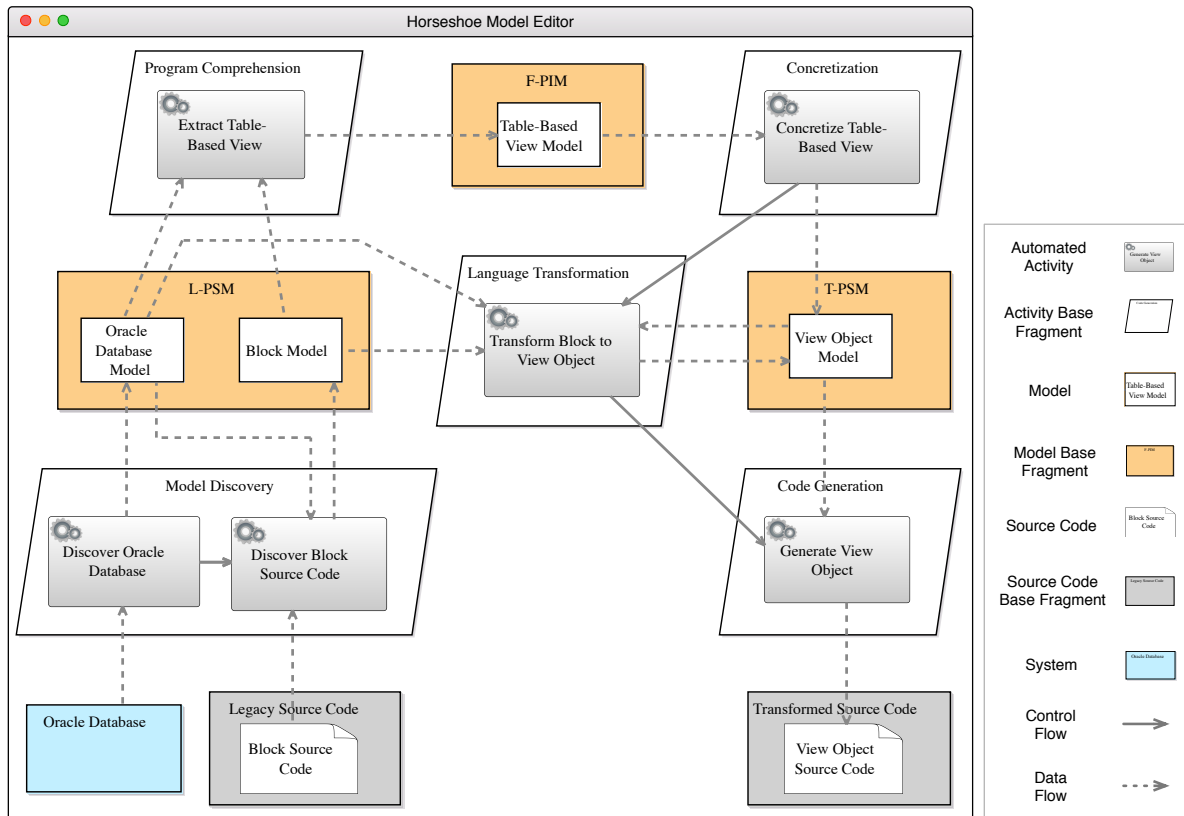


Figure 6.16 Horseshoe model showing customized method fragments to transform the table-based view concept of the running example, specified in MIML

e.g., tools that realize custom reverse engineering algorithms (cf. Section 5.4). In contrast, for each semi-automatic or manual activity, a role needs to be associated.

We distinguish between *inner* and *outer* fragments. Inner fragments represent actual customizations of fragments. In contrast, outer fragments can be seen as container elements to group inner fragments that share a common base fragment. Among other things, this enhances the visual representation of the fragments as the applied pattern becomes directly visible. Related to activities, outer fragments can be seen as a representation for the process that is formed by the inner activities. For example, *Discover Oracle Database* and *Discover Block Source Code* are inner fragments. They are customizations of the *Model Discovery* fragment, which is represented by an outer fragment called *Model Discovery*. The outer fragment represents the equally named process which is formed by the contained, inner fragments.

While the fragments shown in Figure 6.16 have been automatically generated, operations are provided to enable a fine-granular configuration. In particular, activities or artifacts can be renamed, added or removed. The addition is only possible per outer fragment, i.e., a new inner fragment can be added to an outer fragment whereby it must be of the corresponding type. This

enables refining activities or artifacts further to provide a more fine-granular guidance during the enactment. Take the *Generate View Object* activity as an example. In its current state, the activity represents a generation of the *View Objects* itself as well as all of its contained *View Attributes* (cf. Section 5.5.3). By adding another *Code Generation* activity called *Generate View Attributes* these two concerns could be separated into two sequential activities to provide more fine-granular guidance. Note that we do not foresee a refinement of the *System* fragment, as it is out of scope.

Characteristics of Concept Models

The running example only gives an intuitive idea of the selection of method patterns within a concept model. To guide this endeavor, we discuss certain desired characteristics of the resulting concept model. We classify them based on the quality characteristics for methods described in [Har97, pp.227-243]. In particular, we discuss the *Completeness* of the model as well as a characteristic related to its *Soundness*, namely *Graduated Abstraction Levels*.

Completeness At this point in the method engineering process of MEFiSTo, we consider the concept model to be complete if a method pattern has been selected and configured coarsely-granularly for each contained concept.

An exception arises if an *is-a* relation is present. In such an instance, we consider the concept model to be complete in two cases: On the one hand, a pattern can solely be selected and configured for the root concept in the tree that is formed by the *is-a* relations. An example for this case is the *Table* concept shown in Figure 6.15. In this case, no method fragments get instantiated for the more specific concepts like *Primary Table* or *Foreign Table*. Rather, they can be seen as special cases that need to be considered during the transformation of the root concept. On the other hand, patterns can solely be selected and configured for all leaf concepts within the tree structure.

Graduated Abstraction Levels (Soundness) So far, we discussed that influence factors are the main driver when selecting a method pattern for an identified concept. However, such factors are local, i.e., limited to one concept only. As soon as multiple concepts exist within the concept model, we additionally need to consider interdependencies between the decisions. This is due to the fact that the patterns get integrated subsequently to form a consistent transformation method specification.

Take for example the concept model shown in the left side of Figure 6.17. In the example shown, an *Event Handler* concept consists of a *User Interface Element* concept, which in turn consists of a *User Interface* concept. We assume that the dependencies formed by these

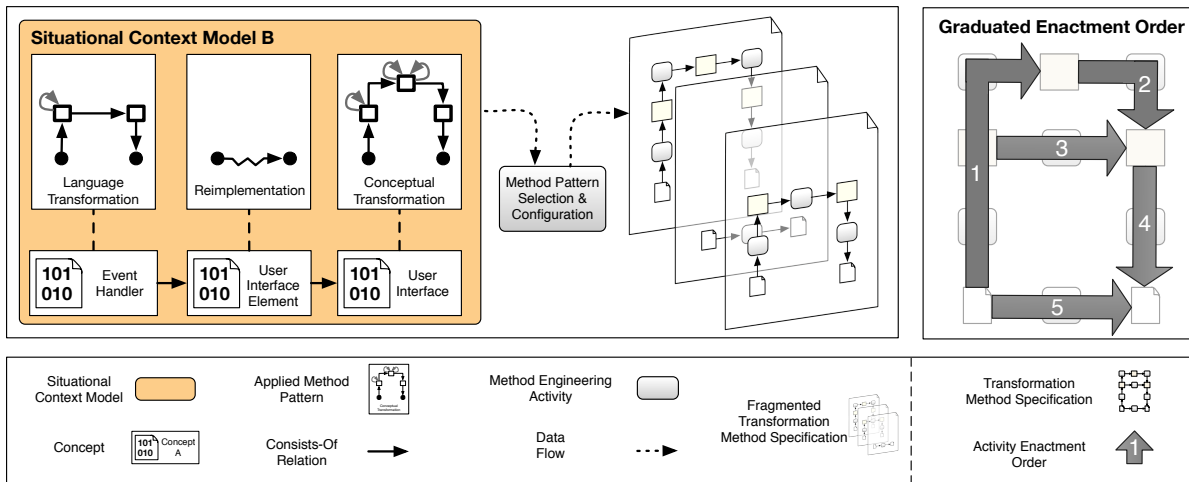


Figure 6.17 Example for an interrupted transformation method specification (left), graduated enactment of an integrated transformation method (right)

consists-of relations imply the order of transformations. Related to the example, the user interface needs to be transformed first before its elements are added. In the end, event handlers are transformed which relate to certain user interface elements.

When integrating the method patterns, these dependencies will manifest themselves by data flow dependencies. In the example shown, the reimplementation activity that reimplements *User Interface Elements* will use the generated *User Interfaces* as an input. Examples for such data flow dependencies when integrating different patterns can also be seen in the examples described in Sections 5.5.5 and 5.6.1.

Considering the resulting data flow dependencies during the selection of the method patterns is critical, as it will influence the overall efficiency of the resulting method. For example, the selection of method patterns as shown in Figure 6.17 might not result in an efficient transformation method as the two automated transformations are interrupted by a manual reimplementation. More precisely, inefficiency may arise from the fact that the automated transformation of the *Event Handler* needs to consider the output of the manual reimplementation of *User Interface Elements*. In the worst case, it might only be possible to develop model transformations for the applied *Language Transformation Pattern* after the reimplementation took place, slowing down the overall transformation.

Based on these findings, we assume that an efficient transformation method will arise if the concept model fulfills the following constraint: If one method pattern (MP_1) depends on another one (MP_2), i.e., the concept related to MP_1 *consists-of* the concept related to MP_2 , then MP_1 needs to be located on the same or a lower level of abstraction than MP_2 . An example for a concept model that fulfills this constraint is shown in Figure 6.14 on page 162. In this case, a

graduated enactment of the method fragments as indicated in the right side of Figure 6.17 is fostered. At first, reverse engineering-based method fragments are enacted to reach the highest level of abstraction. Thereafter, the level of abstraction is progressively lowered by iteratively performing concretizations, possibly preceded by a restructuring.

We assume that aligning the abstraction levels as described to foster a graduated enactment is beneficial for two reasons. On the one hand, all (semi-)automatic activities are enacted before the manual reimplementation takes place. Therefore, an interruption of automated activity by manual reimplementations is ruled out. On the other hand, adhering to a global sequencing that is aligned with the abstraction levels of the transformation enables controlling the complexity of the overall transformation. In particular, we assume that this enables generating meaningful intermediate results, in terms of models, that can be automatically or manually validated.

To evaluate whether the transformation method indicated by a configured concept model fosters a graduated enactment, or not, we express the observations as patterns and anti-patterns. Thereby, anti-patterns represent configurations that should be avoided. We identified three classes of patterns and anti-patterns, whereby a member of each class is shown in Figure 6.18.

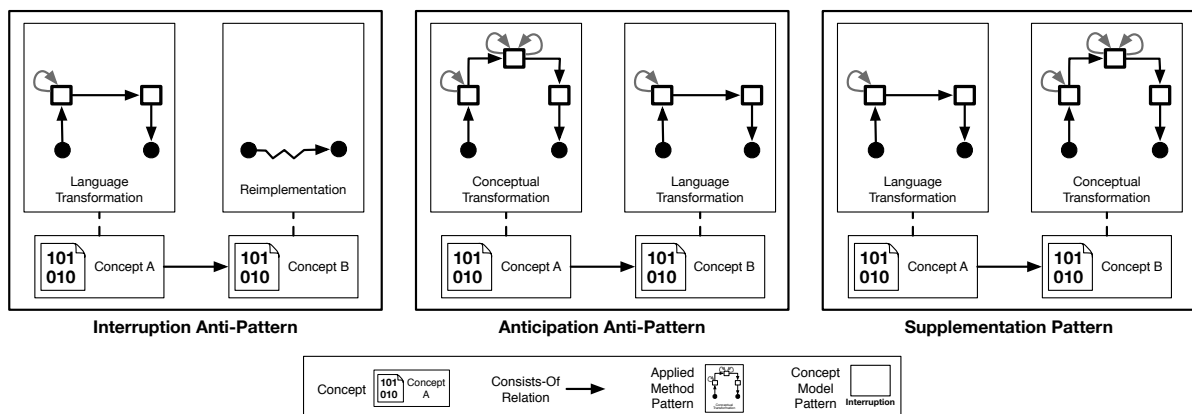


Figure 6.18 Excerpt of patterns and anti-patterns within a concept model

An *Interruption Anti-Pattern* represents the case that an automated transformation depends on the output of a manual one, its presence can be an indicator that an inefficient transformation method will arise. An *Anticipation Anti-Pattern* is comparable to the *Interruption Anti-Pattern* as the dependent method pattern is also located on a higher level of abstraction. However, in this case the kind of transformation is not changed, i.e., both method patterns represent automated transformations. Ideally, the concept model only contains a set of *Supplementation Patterns* which represents the case that the dependent pattern is located on the same level or a lower level of abstraction. Intuitively, the dependent pattern supplements an existing transformation.

A graduated enactment is not fostered if an instance of an anti-pattern exists within the concept model, i.e., if a dependent method pattern MP_2 is transformed on a lower level of

abstraction than the pattern MP_1 on which it depends. In this case, two solution strategies can be applied: either, the level of abstraction of MP_2 is decreased, or the level of MP_1 is increased.

In general, a decision between both strategies requires evaluating the resulting implications in terms on the efficiency and effectiveness. However, based on our observations in practice we assume that the latter strategy, i.e., increasing the level of abstraction, is preferable in the presence of an *Anticipation Anti-Pattern*. This is due to the fact that it is often not necessary to transform the whole concept on a higher level of abstraction, but only a placeholder for it.

To make this clearer, assume the exemplary concept model shown in the left side of Figure 6.19. In this example, the *User Interface Element* concept is dependent on the *User Interface* concept. This dependency arises from the fact that user interface elements *are part of* a user interface. Since user interfaces are transformed on a lower level of abstraction than the user interface elements, an anti-pattern arises. In particular, the model on the highest level of abstraction will not be meaningful as it only represents a set of user interface elements. These elements cannot be aligned with the overall structure of the user interface as the structure only resides on a lower level of abstraction.

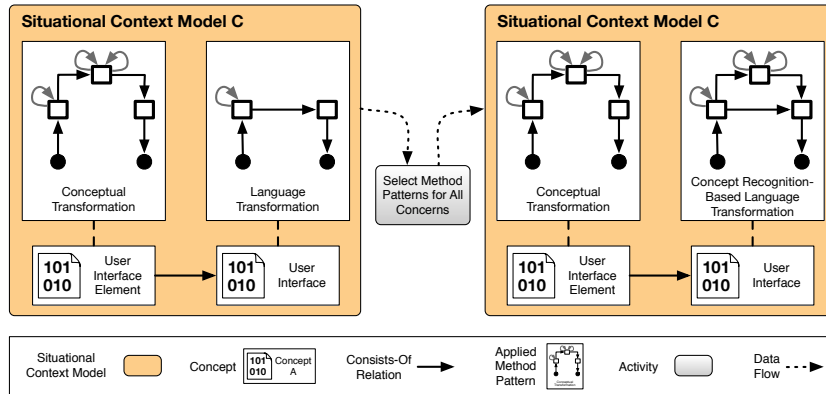


Figure 6.19 Removal of an anticipation anti-pattern instance within a concept model

Here, we decide to eliminate the anti-pattern by raising the level of abstraction for the *User Interface* concept by applying the *Concept Recognition-Based Language Transformation Pattern*, shown in the right side of Figure 6.19. Usually, this pattern is used to transform parts of a concept on a platform-independent layer to improve an underlying language transformation (cf. Section 5.6.2). However, in this case we use the pattern to enable an integration by only transforming those parts of the concept on the platform-independent layer that are required for the integration. In particular, we extract a placeholder for each user interface present. This enables an integration on the platform-independent layer, while the majority of the user interface can still be transformed on the platform-specific layer, e.g., the pixel-precise layout.

We frequently used this particular resolution strategy in practice. In fact, even the example described for the *Conceptual Transformation Pattern* in Section 5.5.2 is a result of applying this strategy. The example describes a conceptual transformation of the *Attribute Validation Rule* concept. However, as this concept depends on the *Table* concept (cf. Figure 6.15, page 164) which shall be transformed by applying the *Language Transformation Pattern* (cf. Section 5.5.1), an anti-pattern arises. The resolution strategy can be seen in the upper part of Figure 5.10 on page 95. The *RelationalTable* instance is the placeholder for the *Table* concept which enables an integration on the platform-independent layer.

Characteristics of Horseshoe Models

So far, we discussed characteristics of concept models which are used to derive a transformation method specification that is fragmented into different parts. Each part consists of a horseshoe model which can be fine-granularly configured. To guide the configuration of horseshoe models, we discuss their desired characteristics. We classify them based on the quality characteristics for methods described in [Har97, pp.227-243]. In particular, we discuss the *Completeness* of the horseshoe model as well as two characteristics related to its *Soundness*, namely *Unambiguous Generation* and *Dependency Specification*.

Completeness We consider a horseshoe model to be complete if the customized fragments conform to the constraints described by the method pattern applied. On the one hand, each pattern specifies which types of fragments to use, i.e., to customize. On the other hand, each pattern specifies how to customize them, e.g., which data or control flow relations are valid.

Initially, the conformity can be ensured when assuming that the set of model transformations, which is used to derive the customized method fragments, is correct. Then, the initial MIML model is correct, too. To ensure the correctness of the model after a change, i.e., a fine-granular configuration, we could follow two strategies: either each change could be correctness-preserving, or we define a validation of the model.

The tooling realized, follows both approaches. The operations provided ensure certain properties of the model. In particular, as we do not provide operations to add or remove outer fragments, it is not possible to add new fragments that do not belong to the pattern applied.

Nevertheless, the inner fragments might be in an incorrect state after changes have been applied, wherefore we require their validation. We need to ensure the following constraints, whereby some of them are already defined in the MIML metamodel using OCL (cf. Section 5.7):

- Each outer artifact or activity contains at least one inner artifact or activity
- Each artifact is an input and/or output of an activity

- Each activity only consumes and/or produces artifacts according to the definition of the base fragment
- Each activity that represents a model from/to model/text transformation is either performed automatically or semi-automatically
- Each activity that represents a reimplementation is performed manually
- Each activity that is performed automatically or semi-automatically is related to a tool
- Each activity that is performed manually or semi-automatically is related to a role

Unambiguous Order (Soundness) If an artifact is produced by multiple activities, we require that the horseshoe model specifies an unambiguous order of activities that are related with that artifact. Such instances occur if intermediate activities like *Restructurings* or *Enrichments* are used, if multiple models are merged (cf. Section 6.4.2) or if a composed pattern is applied.

To make this clearer, take for example the horseshoe model shown in Figure 6.16 on page 165, which represents the application of such a composed pattern. Thereby, the *View Object Model* artifact is changed by two activities, namely the *Concretize Table-Based View* activity as well as the *Transform Block to View Object* activity. In addition, it is consumed by the *Generate View Object* activity. Assume that solely the data flow would be specified. Then, it would be clear that the *Concretize Table-Based View* activity has to be performed first, as it is the only activity that does not require the *View Object Model* as an input. However, it would not be clear which one of the remaining two activities to perform next. In this example, the *Generate View Object* activity actually depends on the changes performed by the *Transform Block to View Object* activity, but this dependency would not become clear by the data flow dependencies solely. We assume that such an ambiguous order is an indication for an inaccurate method specification.

The reason for the necessity to provide an unambiguous order as soon as an artifact is produced by multiple activities is due to the *lifecycle* of the artifact. Whenever an artifact is only produced by a single activity, its lifecycle only consists of one state, namely *created*. However, as soon multiple activities produce an artifact, its lifecycle will consist of more states. In this instance, activities may depend on a specific state of the artifact, like the *Generate View Object* activity in the example described.

One solution to this problem would be to explicitly model the lifecycle of an artifact, i.e., all states of the artifact. Then, whenever an activity consumes an artifact, a guard can be used that specifies in which state the artifact needs to be. However, MIML currently does not support the specification of an artifact lifecycle. Instead, we explicitly specify the control flow between

associated activities to express dependencies between them. As future work, we envision that the lifecycle of an artifact is derived automatically based on the current state of the specification. We want to point out that we did not realize a validation of this characteristic but only sketch the idea. In the current state, we assume that the model is validated manually.

Dependency Specification (Soundness) We require that for each customized activity its dependencies in terms of control and data flow are specified. Note that this is not ensured by the two characteristics that have been introduced. On the one hand, even if a method specification is complete according to the definition given, dependencies might be missing. This is due to the fact that the *Completeness* characteristic only defines syntactical constraints, but dependencies are of semantical nature. On the other hand, the *Unambiguous Order* characteristic can be seen as one specific kind of dependency that needs to be specified. We put an emphasis on this characteristic, as it commonly arises.

An example for a dependency that would be missing even if the method specification is complete and specifies an unambiguous order can be seen in the horseshoe model shown in Figure 6.16 on page 165. If the activity called *Discover Oracle Database* would not be a predecessor of the activity called *Discover Block Source Code* activity and if the latter activity would not consume the artifact called *Oracle Database Model*, the set of method fragments would still adhere to these two characteristics. However, the dependency between those two activities is essential. In this example, the *Discover Block Source Code* activity performs a semantic analysis of the source code, i.e., references from the source code to the database used are identified. These references are established in form of links between the *Block Model* and the *Database Model*. Note that these references have been described and exemplified in Detail in Section 5.5.1.

We assume that such dependencies are explicitly specified manually. As future work, it could be sufficient to define relations between involved artifacts and derive data and control flow dependencies automatically. Related to the example, it could be expressed that the *Block Model* will refer to the *Oracle Database Model*, which would imply a data flow dependency of the *Discover Block Source Code* on the *Oracle Database Model*. This, in turn, would imply the control flow dependency on the *Discover Oracle Database* activity.

Process

So far, we gave an idea of the artifacts involved in the activity called method pattern selection and & configuration (cf. Figure 6.13, page 161), namely a configured concept model as well as a set of derived horseshoe models. To guide their systematic instantiation, we refine the activity. The resulting process is illustrated in Figure 6.20.

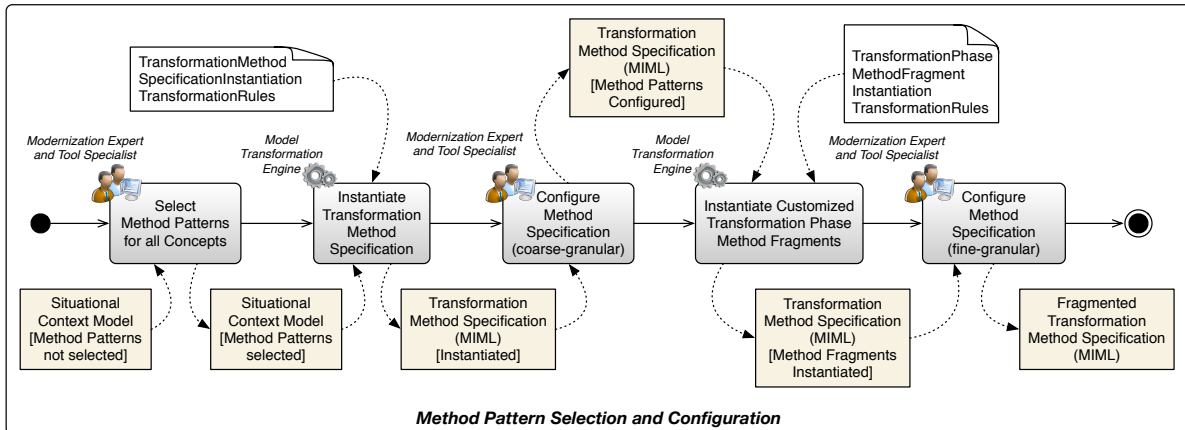


Figure 6.20 Method pattern selection and configuration process

The process begins with the selection of a method pattern for each concept identified. As the decisions on how to transform the concepts will essentially influence the overall efficiency of the resulting transformation method, they need to be performed against the background of the identified influence factors. If an anti-pattern arises, a resolution strategy needs to be applied as described in the beginning of this section.

Based on the resulting concept model, a transformation method specification gets automatically instantiated. At this point, the specification only contains instances from elements of the *Pattern* package in MIML (cf. Section 5.7.1), e.g., *Method Pattern* and *Method Pattern Configuration* instances.

After the transformation method specification has been instantiated, it gets configured by the third activity of the process. This comprises a coarse-granular configuration of the applied method patterns, i.e., a selection of optional parts and the provisioning of meaningful names.

As described in the beginning of this section, this configuration is essential to automatically derive customized method fragments, which is the purpose of the fourth activity. These fragments are instances of the *Fragments* package in MIML (cf. Section 5.7.1), e.g., *Model Discovery* or *LPSM* instances.

As a last activity, a fine-granular customization of the instantiated Method fragments takes place. Among other things, generated fragments can be added, renamed or removed.

6.4.2 Method Pattern Integration

So far, method patterns have been selected and configured for all identified concepts, based on which a fragmented transformation method specification has been derived (cf. Figure 6.14, page 162). As we aim to develop a coherent specification, we need to systematically integrate distinct parts. This is the purpose of this activity, its core idea is visualized in Figure 6.21.

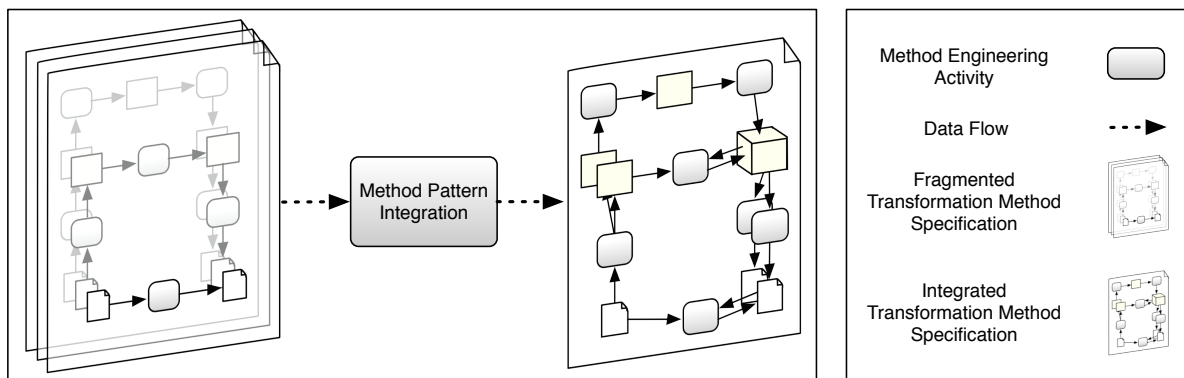


Figure 6.21 Integrating a fragmented transformation method specification

The starting point of the *Method Pattern Integration* activity is a fragmented specification, i.e., a specification that consists of customized method fragments for each pattern applied. The fragments that originate from one pattern form a horseshoe model. They are related to other fragments within the same horseshoe model by control or data flow relations, but no relations between horseshoe models exist. Intuitively, the specification consists of a set of separated specifications, as indicated in the left side of Figure 6.21.

By performing the current activity, we aim to integrate the separated specifications by establishing relations between fragments belonging to different horseshoe models. In the end, an integrated specification shall result as shown in the right side of Figure 6.21. We define an integrated transformation method specification as follows:

Notation 19 (*Integrated Transformation Method Specification*)

An integrated transformation method specification is a specification that consists of a set of horseshoe models whose method fragments have been integrated.

The integration is performed based on the MIML model. We want to point out that an integration is not absolutely necessary at this point. As the subsequent activities transform the MIML-based specification to SPEM, the integration could also be performed in the resulting SPEM-based specification. More precisely, it could be performed during the *Method Completion* activity (cf. Section 6.4.5). However, as demonstrated in Section 5.7.3, a method specification in SPEM consists of more modeling elements than its counterpart in MIML. As a result, performing changes in MIML requires less effort than performing the same changes in SPEM. Therefore, we assume to reduce the overall specification effort by enabling an integration within the MIML model.

To systematize the integration, we sketch different types of operations that can be executed on a set of method fragments. An overview of the proposed types is illustrated in Table 6.2. We

want to point out that the visualizations shall only give an idea of the operations. In particular, we did not formalize them or provide tool-support but leave this open for future work.

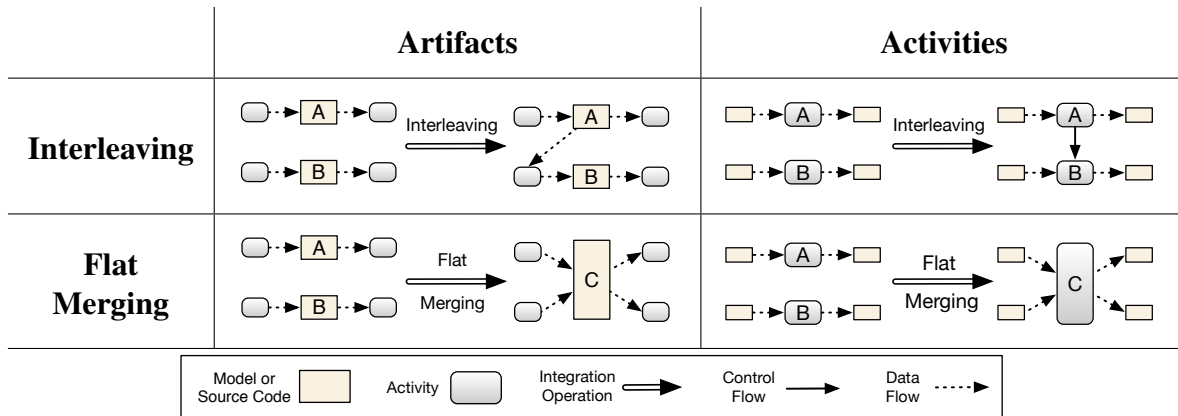


Table 6.2 Operations for the integration of method fragments that originate from different method pattern applications

The table shows integration operations applied on activities and artifacts. The most basic form of integration is called *interleaving*. This describes the introduction of control or data flow relations between fragments that originate from different pattern applications. While *interleaving*-based operations only introduce additional relations between existing method fragments, *merging*-based operations modify the fragments itself. In particular, applying a *flat merge* results in a fusion of two or more fragments. Thereby, the fragments need to originate from different pattern applications and need to have the same type, i.e., they need to be customizations of the same base fragment stored in the method base. Note that both operations can be applied on associated roles and tools, too.

One could ask oneself, why different types of operations are required, i.e., why it is not sufficient to perform the integration solely by applying *interleaving*-based operations. This is due to the fact that the different types of operations vary in their implications on the functional granularity of the resulting transformation method specification. When comparing *interleaving*-based operations to *flat merging*-based ones, this difference can be observed. While *interleaving*-based operations do not change the granularity, the fusion of fragments can be used to locally increase the granularity, i.e., decrease the level of preciseness.

Running example

We exemplify the integration of different method patterns based on the fragmented transformation method specification that has been introduced in the previous section (cf. Figure 6.15, page 164). We perform the integration by using the capability of our tooling to provide an

integrated horseshoe-based view on a selected set of applied method patterns. In Figure 6.22, an example for such a view is shown. More precisely, the integrated horseshoe model for the *Table-Based View*, *Attribute Calculation* and *View Relation* concept is depicted.

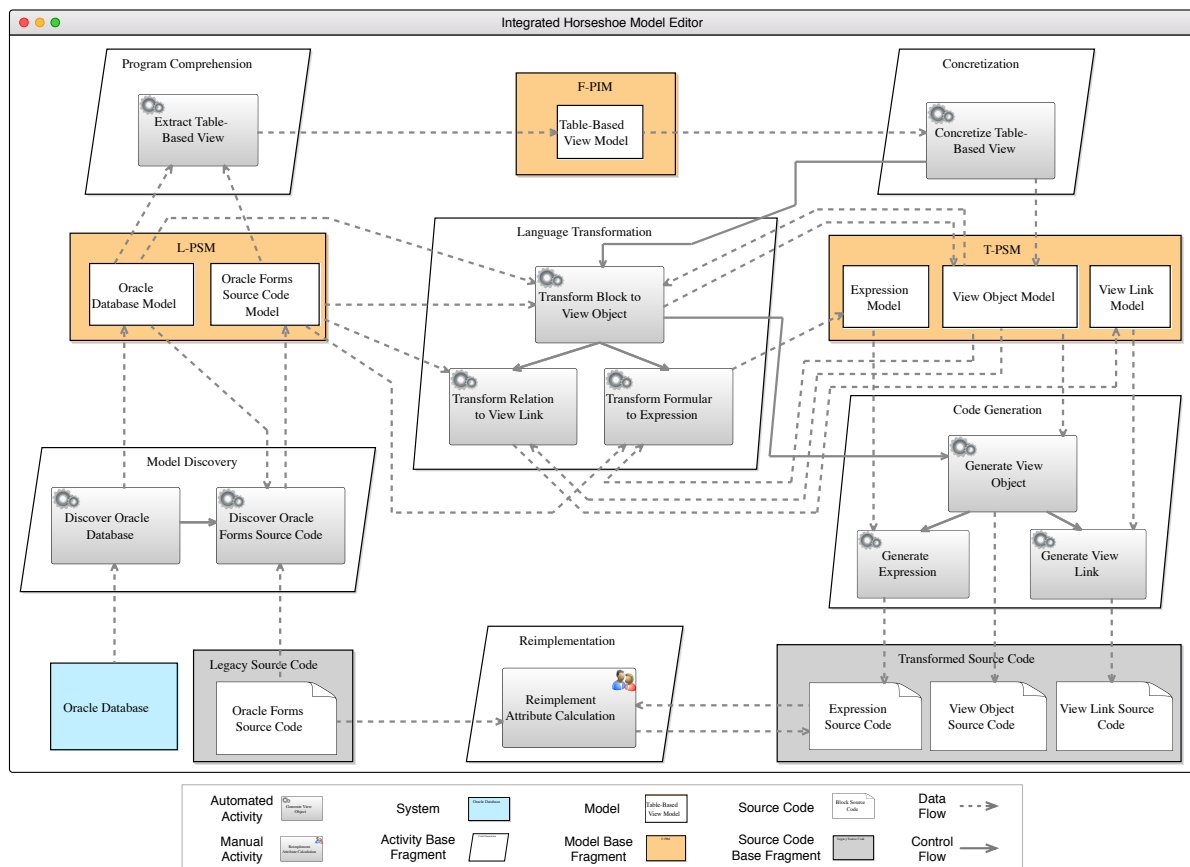


Figure 6.22 Integrated horseshoe model showing method fragments for the *Table-Based View*, *Attribute Calculation* and *View Relation* concept of the running example, specified in MIML

Note that the figure already shows the result of (manually) applying various integration operations to integrate the fragments of the three horseshoe models. This becomes clear when taking a look at Figure 6.16 on page 165 which shows a horseshoe-based view related to the *Table-Based View* concept before the integration. Subsequently, we highlight some of the integrations that have been performed.

In the example shown, the *Model Discovery* fragments that discover *Legacy Source Code* have been integrated by applying a *flat merge* operation. Before the integration, three customized *Model Discovery* fragments existed. Thereby, each fragment specified the syntactic and semantic analysis of language constructs that were used to realize the corresponding concept. For example, as the *Table-Based View* concept is realized by Blocks, a *Discover Block Source Code* activity existed (cf. Figure 6.16, page 165). However, we knew that a commercial

syntactic and semantic analyzer is available for the programming language of Oracle Forms. Therefore, we decided that it is not necessary to provide a fine-granular specification for the model discovery activity since the available analyzer shall be used. As a result, we merged all *Model Discovery* fragments that relate to the analysis of Oracle Forms source code into the *Discover Oracle Forms Source Code* activity.

The example demonstrates how we intentionally lower the granularity of the specification for those parts where a fine-granular specification is not necessary. In general, we assume that a fine-granular specification is beneficial whenever an in-house development needs to be performed. For example, if no analyzer for the programming language of Oracle Forms would have been available, then a fine-granular specification of the model discovery activity would be beneficial to guide the development of such an analyzer.

The horseshoe model shown in Figure 6.16 on page 165 shows a detailed view on inner and outer fragments that originate from three different method patterns. In contrast, Figure 6.23 focuses only on outer activities. In particular, it shows all processes that are part of the developed transformation method, based on the concept model shown in Figure 6.15 on page 164.

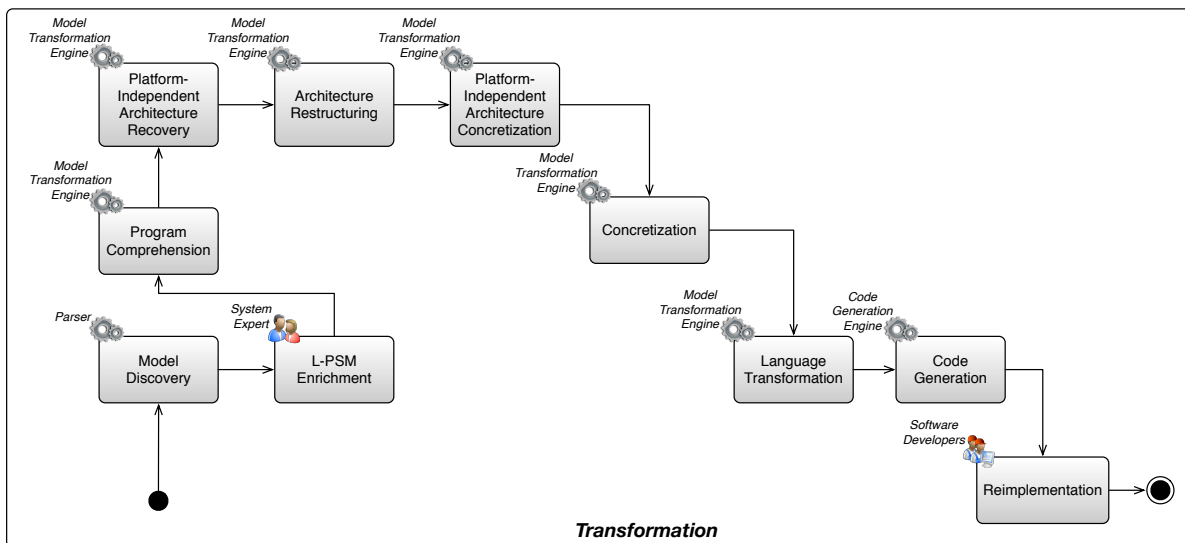


Figure 6.23 Transformation process

The model shows a view on the outer activities, control flow dependencies between them as well as associated roles or tools. The control flow dependencies have been manually derived from specified data flow dependencies. Note that they have been sequentialized in a graduated order, related to the abstraction levels. This is possible as the concept models adhered to the corresponding constraint (cf. Section 6.4.1).

Characteristics of Integrated Horseshoe Models

In the previous Section 6.4.1, we already discussed desired characteristics for a horseshoe model, namely its *Completeness* and *Soundness*. Thereby, the horseshoe model resulted from applying a method pattern on a single concept. Here, we discuss to what extent these characteristics still apply in the case of an integrated horseshoe model.

Completeness We consider a horseshoe model to be complete if the customized method fragments that result when applying a method pattern on a concept conform to the pattern. When transferring this definition to the level of an integrated horseshoe model, we can distinguish two cases: whether the integrated horseshoe model forms an instance of a pattern again, or not.

Take for example the integrated horseshoe model shown in Figure 6.22. This model is a result of integrating the fragments of a *Language Transformation*, *Language Transformation-Based Reimplementation* and *Concept Recognition-Based Language Transformation Pattern*. Due to that, the integrated horseshoe model conforms to the *Concept Recognition And Language Transformation-Based Reimplementation Pattern* (cf. Section 5.6). In this case, means provided to validate the underlying pattern can be reused to validate the integrated patterns.

As long as patterns of the same type are integrated, i.e., patterns that either address *functionality preserving* or *architectural restructuring* (cf. Section 5.2), the resulting integrated horseshoe model will always conform to a method patterns contained in the method base. As soon as both types of patterns are integrated, this will not be the case. However, we can still validate fragment-related constraints that have been introduced in the previous section.

Soundness The characteristics called *Unambiguous Order* and *Dependency Specification* describe requirements on the level of method fragments, i.e., constraints for a specific constellation of fragments. They are also applicable on an integrated horseshoe model.

For the *Dependency Specification*, we assume that an additional possibility to validate the dependencies within an integrated horseshoe model exists. In particular, the characteristic claims to specify dependencies between activities. In the case of an integrated horseshoe model, we assume that these dependencies must be consistent to the *consists-of* relations specified in the concept model.

Take for example the integrated horseshoe model shown in Figure 6.22. In this model, three *Language Transformation* activities have been specified as well as control flow relations between them. These activities originate from three different concepts, as can be seen in Figure 6.15. When comparing the concept model with the resulting integrated horseshoe model, it can be seen that the control flow dependencies between the activities are consistent with the *consists-of* relations in the concept model.

6.4.3 Instantiation of Tool Implementation Phase Fragments

So far, an integrated transformation method specification has been developed. The specification consists of customized method fragments that describe, i.e., guide, the transformation phase (cf. Section 5.3). As the specification shall also cover the tool implementation phase, related customized fragments need to be added. This is the purpose of this activity, its core idea is visualized in Figure 6.24.

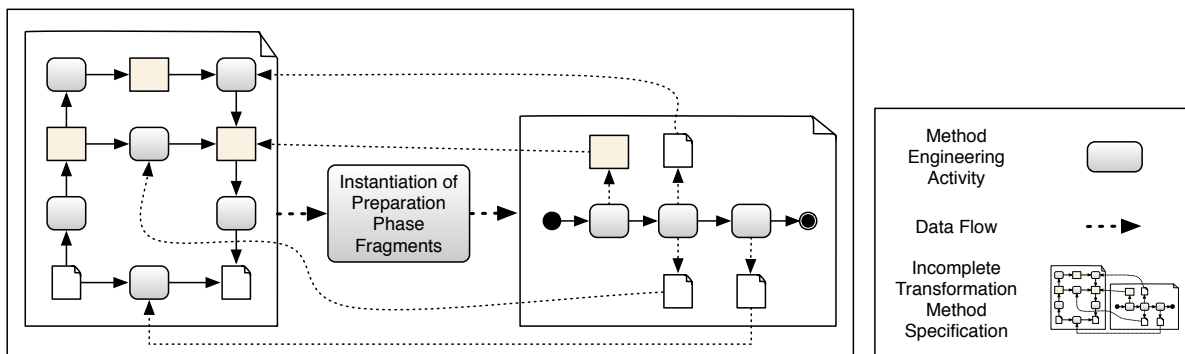


Figure 6.24 Instantiation of tool implementation phase fragments

The transformation method specification in its current state only contains customized method fragments that describe the transformation phase, shown in the left side of Figure 6.24. However, in order to enact this part of the specification, various tools need to be in place. For example, the instantiation of a model requires the presence of a metamodel, the execution of a model transformation requires model transformation rules, and manually performed reimplementations require guidance documents.

By enacting the activity called *Instantiation of Tool Implementation Phase Fragments*, the specification shall be enriched by additional fragments that guide the development of required tools. This part of the specification is shown in the right side of Figure 6.24. In particular, these fragments shall be derived from the existing specification. We assume that the derivation can be performed automatically by using a model transformation. However, we currently did not realize those transformations, wherefore it needs to be performed manually.

Note that the transformation method specification consists of all customized method fragments. However, as indicated in the figure, the specification can logically be separated into two specifications: One that contains all fragments that specify the actual transformation, and another one that contains all fragments that specify the preliminary development of required tools. We will call the latter part of the specification *tool implementation process*, while we call the former part *transformation process*.

Running example

We exemplify the activity based on the running example that has been introduced in Section 4.3. More precisely, Figure 6.25 shows the *tool implementation process* that needs to be performed to enable an enactment of the previously defined *transformation process*, shown in Figure 6.23.

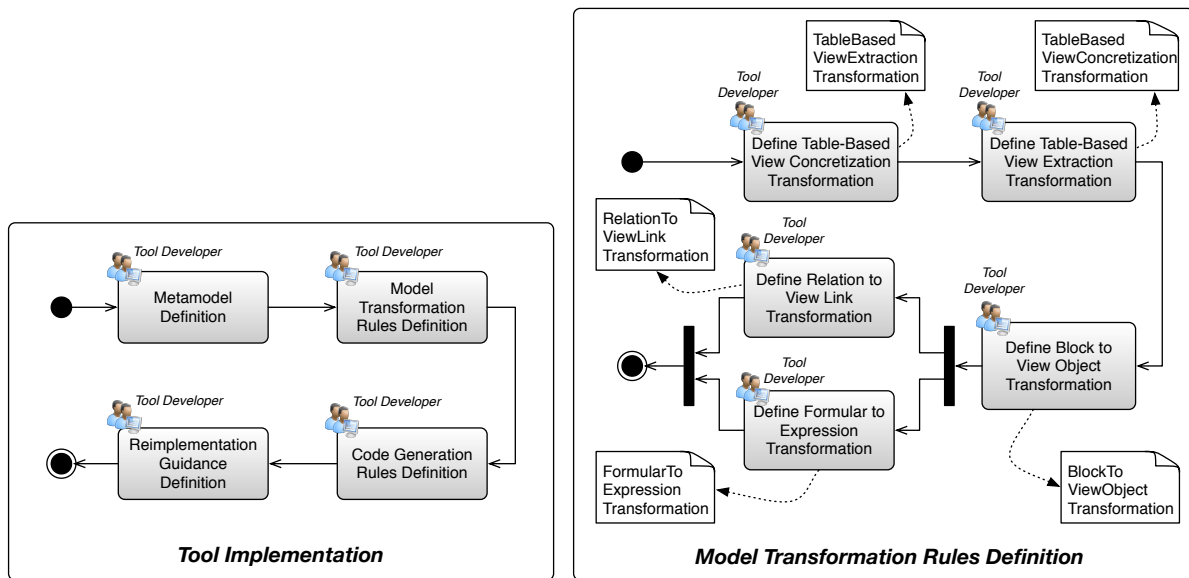


Figure 6.25 Overall *Tool Implementation* process (left), process of the *Model Transformation Rules Definition* activity (right)

In the left side of Figure 6.25, the outer activities of the tool implementation process are shown. Each of those activities contains inner activities that are customizations of the equally named fragments (cf. Section 5.4).

At first, the *Metamodel Definition* activity needs to be performed which outputs a set of metamodels for all models arising during the transformation. We assume that the metamodels always need to be defined first, as their manifestation can be relevant for all remaining activities. For example, the subsequent definition of model transformation rules and code generation rules directly depends on the metamodels used. As a last step, documents to guide reimplementation activities are defined. In principle, it is possible to enact all activities that follow the *Metamodel Definition* activity in parallel. However, we place them in a sequential order to enable integration testing during the development. For example, code generation rules can only be tested if corresponding models exist, which are generated by model transformations.

In the right side of Figure 6.25, the inner activities to develop required model transformation rules are shown. It can be seen that five rules are developed in total, they are an input of the model transformations specified in Figure 6.22 on page 176. Regarding the order, the same

argument as for the outer activities applies. In principle, all activities can be performed in parallel. However, to enable testing, we use the order implied by the transformation process.

6.4.4 MIML To SPEM Transformation

So far, the transformation method consists of customized method fragments for the tool implementation as well as for the transformation phase, specified in MIML. At this point, we foresee a transformation of the specification to SPEM [OMG08a]. Performing the transformation is the purpose of this activity.

The general idea of this transformation as well as an example has already been described in Section 5.7. We transform the MIML model to a SPEM model that extends the predefined *MEFiSto Method Base SPEM Model*. The predefined model formalizes the (generic) content of the method base.

We do not go into detail on the mapping applied but aim to give the general idea. Related to activities, we essentially need to distinguish inner and outer ones. As outer activities can be seen as a container for inner activities, i.e., as a representation of a process (cf. Section 6.4.1), they are mapped to *Activities* in SPEM. In contrast, inner activities become *Tasks* (cf. Figure 5.21, page 139). Related to the remaining fragments like artifacts, tools, and roles, this distinction is not necessary.

6.4.5 Method Completion

So far, we automatically derived an integrated transformation method specification in SPEM, based on a corresponding specification in MIML. At this point, the resulting SPEM-based specification is not complete in the sense that some information has not been specified yet. The completion of the specification is the purpose of this activity.

Most importantly, *descriptions* for all customized method fragments need to be provided. The transformation method specification will only provide actual guidance for people if its contained fragments are meaningfully described. SPEM provides various capabilities for this purpose. For example, for most elements of the *Method Content* package (cf. Section 5.7.2), textual attributes like *brief Description*, *main Description* or *purpose* can be specified [OMG08a, p.76]. In addition, tools to edit SPEM-based specifications, like the *Rational Method Composer (RMC)*, enable to include images or define graphical diagrams, like activity diagrams. However, we do not prescribe the extent of the descriptions, i.e., which attributes have to be used and which information to fill in, but leave this open for the experts to decide. We assume that the amount of descriptions required essentially depends on the people that shall be guided by the specification.

Besides providing descriptions, we also foresee a *refinement* of the functional granularity of the developed specification. SPEM provides capabilities for this purpose. For example, Steps can be specified for Tasks [OMG08a, p.88-90]. A Step refines a Task by partitioning the overall goal of the Task into smaller, meaningful parts. While Steps can be used to refine an existing Task without changing the overall structure of the specification, refinements could also be achieved by adding additional customized fragments. For example, one could add an additional customization of the *Model Discovery* fragment in order to refine the corresponding *Model Discovery*-process. However, such arbitrary refinements change the structure of the specification and therefore require its validation afterwards, which we currently do not address.

Running example

We exemplify the activity based on the running example that has been introduced in Section 4.3. More precisely, Figure 6.26 shows an excerpt of a SPEM-based method specification to transform the running example. It has been derived from the MIML-based specification described in the previous sections and been manually refined using the RMC.

The excerpt of the specification shows customized method fragments that are related to the transformation phase. In the left side of the figure, the phase itself is explicitly described by an activity diagram. The diagram contains the outer activities, i.e., Activities in SPEM-terminology, as well as control flow relations between them. These relations have been manually derived from existing data flow relations between inner activities (cf. Figure 6.22, page 176).

Each outer activity can be clicked to navigate to a detailed view on its inner activities. For the *Model Discovery* activity, such a view is shown in the upper right of Figure 6.26. As can be seen, the activity consists of three inner activities, i.e., three Tasks in SPEM-terminology. As for the phase, these activities have been described by an activity diagram.

Also, each Task, i.e., each inner activity, can be clicked to navigate to a detailed description of the Task itself. In the lower right of Figure 6.26, a description for the *Discover Oracle Forms Source Code* Task is shown. Among other things, the *Purpose* of the Task is described, its *Relationships* to artifacts as well as contained Steps.

This concludes a description of the activities that are concerned with the development of the transformation method. By enacting the next two activities of the method engineering process, the developed method gets enacted as specified. As providing guidance for the enactment is not a focus of this thesis (cf. Requirement 1, page 142), we do not go into details when describing the activities but focus on the general idea.

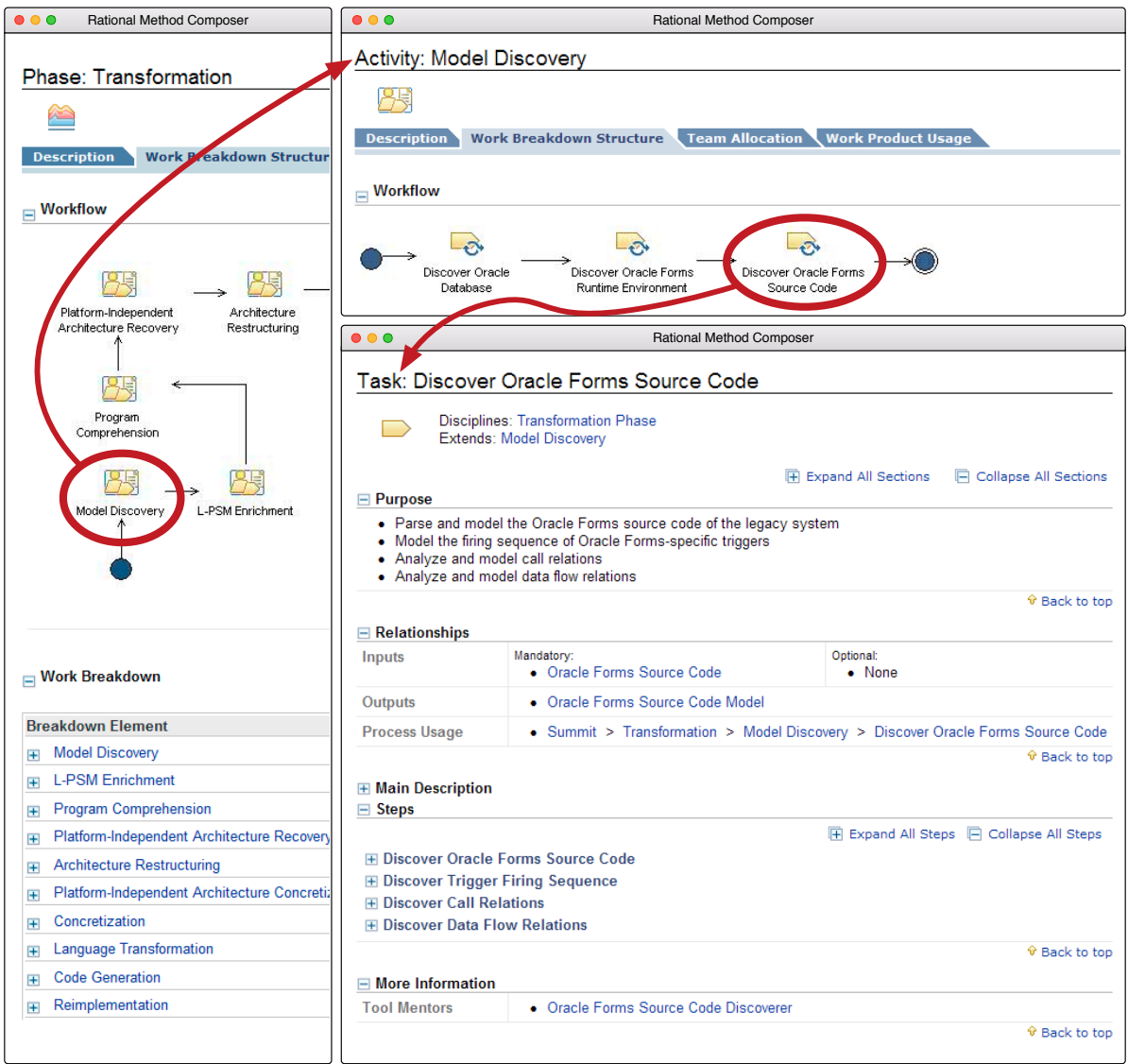


Figure 6.26 Excerpt of SPEM-based transformation method specification that defines a method to transform the running example

6.5 Tool Implementation

So far, a situation-specific transformation method has been developed and specified in SPEM. In this section, we discuss the enactment of the method as specified. In particular, we discuss the *Tool Implementation* activity, whose purpose it is to develop required tools as specified.

When performing the tool implementation activity, only those parts of the transformation method specification are enacted that specify the development of tools. Therefore, the concrete manifestation of the activity is defined by the specification. We assume that associated tool

developers use the specification as some kind of guidance, whereby we foresee at least two kinds of uses. On the one hand, the developers can *browse* the specification to get an overview of the method. By browsing the part that describes the transformation phase, they can get an understanding on how the actual transformation is supposed to take place, i.e., they can understand what the developed tools are supposed to accomplish. In contrast, the part that specifies the tool implementation phase can be used to understand what needs to be developed. On the other hand, the developers can *stepwise enact* the specification by reading the descriptions of the tool implementation activities in the specified order. These descriptions specify what to do in order to develop the required tools.

As future work, we envision the use of a process engine that automatically assigns activities to associated developers and therefore ensures that the specification is exactly enacted as specified. However, in the current state we assume a looser but more flexible form of guidance.

During and after the implementation of required tools, the quality of the implemented tool chain needs to be validated. While this could refer to characteristics like performance, we currently only assume that the software developers take care of their correctness, according to the specification. For this purpose, we assume that the developers use a small extent of the legacy system to test the developed tools.

As examples for the specification have already been given (cf. Section 6.4), we do not go into detail on the actual manifestation of the activity. For detailed examples concerning the development and use of real tools, we refer to the feasibility studies described in Chapters 7.2 and 7.3. Instead, we focus on the capabilities that a generic, project-independent tool infrastructure needs to provide in order to ease the development of project-specific tools. We assume that such tools always need to be developed when applying the MEFiSTo framework. Subsequently, we discuss the required capabilities.

Tool Infrastructure Capabilities

We derive a set of capabilities based on the method fragments proposed in Sections 5.3 and 5.4. We assume that a tool infrastructure shall provide these capabilities in an integrated manner. On the one hand, we describe capabilities that are required to support conversion-based, i.e., automated, transformation strategies. This comprises, for example, the application of a *Language Transformation Pattern*. On the other hand, we describe capabilities that support manual reimplementations activities. Subsequently, we first describe the required capabilities to support automated transformation strategies.

Metamodeling As automated conversions in MEFiSTo rely on the use of models that are transformed into each other, a tool infrastructure shall support the definition of corresponding metamodels. This comprises the definition of their syntax and associated constraints.

Model Repository Since legacy systems can become large, models that describe them can become large, too. Also, different models on potentially different levels of abstraction may arise that are related to each other. Therefore, a tool infrastructure shall support their integrated management by providing a model repository. Besides persisting models, the repository can support additional features like access control, model versioning or concurrent access.

Model Transformation Being able to transform models is essential when defining an automated transformation method using the MEFiSTo framework. Therefore, it shall be supported by the tool infrastructure.

Code Generation Generating code out of models is essential when defining an automated transformation method using the MEFiSTo framework. Therefore, it shall be supported by the tool infrastructure.

Parser Generation Parsing source code and representing the result in the form of an Abstract Syntax Tree (AST) is the first activity when defining an automated transformation method using the MEFiSTo framework. The development of a parser can be supported by providing a parser generator, i.e., a tool that generates a parser from a grammar [Aho+06, p.234]. However, it might not be necessary to develop a parser, as parsers for various programming languages already exist. Therefore, the generation of a parser can be supported by the tool infrastructure.

Besides the capabilities required to support conversions, we identified a set of capabilities to support manual reimplementations activities. These capabilities are described subsequently.

Model Views If a human is included into an automated conversion, e.g., if a system expert shall annotate certain models, then the interaction of the human with the repository can be supported by providing adequate views on the model repository. Thereby, such a view only shows the parts of the model that are relevant for the activity the human is supposed to do, probably in a concrete syntax. Therefore, a tool infrastructure shall support the definition of model views.

Model Querying The capability to query models is closely related to the capability to define model views. If a human derives information from the models stored in the repository during the transformation, it is often required to preprocess the model adequately. This can be performed by queries on the models stored in the repository, wherefore a corresponding tool infrastructure can support it.

6.6 Transformation

So far, a situation-specific transformation method has been developed and specified in SPEM. Also, tools required for the transformation have been implemented. In this section, we discuss the enactment of the method as specified. In particular, we discuss the *Transformation* activity, whose purpose it is to perform the actual transformation of a legacy system as specified.

When performing the transformation activity, only those parts of the specification are enacted that specify the transformation itself. Therefore, the concrete manifestation of the activity is defined by the specification. We assume that associated software developers and system experts use the specification as some kind of guidance. Thereby, we foresee the same kinds of uses as during the tool implementation activity. On the one hand, the people involved can *browse* the method to get an overview of the actual transformation that shall take place. On the other hand, they can *stepwise enact* the specification, as it indicates which activities they need to perform as well as their order. When performing an activity, the people can read the provided descriptions. However, we also foresee the use of separate guidance documents (cf. Section 5.4). If such documents are used, then they should be read instead.

While all activities belonging to the *Tool Implementation* activity are performed manually by tool developers, some activities that belong to the *Transformation* activity are performed automatically. Therefore, the interaction between peoples and tools during the transformation needs to be addressed. In particular, people must get notified whenever a tool has created an output, based on which they need to perform an activity. Also, whenever a person has performed an activity and a tool shall use that output, the person must know which tool to invoke and how. However, we do not address this challenge but assume that it is considered.

Also, we do not go into detail on the manifestation of the transformation activity, since the same arguments apply as for the tool implementation activity. Examples for the specification have already been given (cf. Section 6.4). For examples concerning the transformation of real-world legacy systems, we refer to the feasibility studies described in Chapters 7.2 and 7.3.

Instead, we focus on the incremental enactment of a transformation method. This shall enable an incremental transformation of a legacy system rather than transforming the whole system at once. Resulting implications are discussed subsequently.

Incremental Enactment

A main motivation for transforming a legacy system incrementally is to reduce the risk for a modernization project to fail [BS95, pp.13-14]. Transforming large legacy systems at once can make the project hard to manage due to the underlying complexity and take a particularly long

time before first results are observed. These are just some reasons, why it is generally accepted to perform the transformation incrementally.

An incremental transformation can be achieved by dividing the transformation of a legacy system into the transformation of distinct parts of the system, i.e., *increments* [SWH10, p.107]. We call such increments transformation packages. Different strategies have been proposed to identify such packages. For example, a package can correspond to an architectural layer of the legacy system or to a business functionality it provides.

Whenever transformation packages are identified, they need to be aligned with the developed transformation method. The challenge arises from the fact that the method has been designed by considering the legacy system as a whole. Intuitively, we require that the outcome of transforming a legacy system at once equals the outcome of transforming the identified packages. However, we do not specifically address this challenge but assume that it is considered.

We assume that using transformation packages enables improving the developed method or tools systematically, as the outcome of each increment can be separately validated. Thereby, knowledge gained can be used to improve the transformation of subsequent packages. For example, assume that a package has been transformed. Then, the result can be tested, e.g., by performing regression tests [SWH10, pp.164-166]. If the test indicates an error, software developers need to identify its origin. If it was introduced during a reimplementation activity, it can directly be fixed. However, it might be the case that a tool or even the method itself has flaws. Then, this knowledge can be used to systematically improve them, before the next increment is transformed. Note that this requires communication between the different roles involved, e.g., software developers, tool developers and modernization experts. In [Gri+14], we described how to realize this communication by introducing explicit artifacts in the form of feedback sheets. Details and results of applying this approach are described as part of the first feasibility study in Section 7.2.

6.7 Summary

The MEFiSTo framework enables the modular construction of situation-specific transformation methods by reusing methodological knowledge stored in a method base. In this chapter, we introduced a process to develop and enact situation-specific transformation methods, using the content of the method base.

First, we discussed requirements related to the method engineering process in Section 6.1 and refined its structure in Section 6.2. We introduced the phases of the process and described how they relate to each other.

In Section 6.3, we introduced a process to systematically discover and model the situational context of a modernization project. First, a concept model gets developed that describes a decomposition of the legacy system to transform into distinct concepts, i.e., distinct functionalities. Second, for each identified concept a set of suitable method patterns is identified as well as a set of related influence factors that affect their efficiency or effectiveness. These characteristics of the modernization project are modeled by an influence factor model. We discussed quality characteristics for both models.

In Section 6.4, we introduced a process to systematically construct a transformation method based on the previously discovered situational context. First, the concept model gets configured by deciding how to transform each concept, i.e., which method pattern to apply. Based on the configured concept model a set of horseshoe models is derived which specifies a transformation method in MIML. Thereafter, the horseshoe models get integrated to form a coherent method. As the specification at this point only describes the actual transformation, it gets enriched by a part that guides the development of project-specific tools. In the end of the process, the specification gets transformed into the SPEM standard, before it gets completed by providing missing descriptions. Here again, we discussed quality characteristics for all arising models.

In Section 6.5, we discussed capabilities that a generic tool infrastructure needs to provide in order to support the development of project-specific tools. The challenges and chances related to an incremental enactment of the actual transformation are discussed in Section 6.6.

The last two sections are concerned with the enactment of the developed transformation method specification. We kept them rather short as an enactment of developed transformation methods in the real world was performed as part of the feasibility studies. In the next part of this thesis, we describe those studies in detail.

PART III

EVALUATION AND CONCLUSION

“The greatest thing is when you do put your heart and soul into something over an extended period of time, and it is worth it.”

– STEVE JOBS

Feasibility Studies

In the previous chapters, the MEFiSTo framework has been defined that enables the development and enactment of situation-specific software transformation methods. In this chapter, we describe two feasibility studies in which MEFiSTo had been used to demonstrate the applicability of the framework. First, we revisit the evaluation criteria we aim to discuss with the studies in Section 7.1. The feasibility studies are described in Sections 7.2 and 7.3. Thereafter, we discuss the evaluation criteria based on the experiences made when performing the studies in Section 7.4. The findings of this chapter are summarized in Section 7.5.

7.1 Evaluation Criteria Revisited

The purpose of the feasibility studies is to evaluate the solution concept of this thesis, i.e., the MEFiSTo framework. Therefore, we aim to answer the following evaluation criteria that we formulated in Section 4.2:

- EQ1: Does the content of the method base of MEFiSTo enable to develop situation-specific software transformation methods for the modernization of legacy systems?
- EQ2: Does the method engineering process of MEFiSTo guide the development of situation-specific transformation method specifications?
- EQ3: Can a transformation method that had been developed by applying the MEFiSTo framework be enacted to transform a real-world legacy system?
- EQ4: Does the MEFiSTo framework support different environmental changes?

7.2 Feasibility Study 1: Oracle Forms to Oracle ADF

In this feasibility study we transformed a complete legacy system from the domain of real estates into a new environment by using the MEFiSTo framework [GS13]. Among other things, the purpose of the system was to manage properties and buildings as well as related contracts. It had been originally implemented around 1996 using the platform of Oracle Forms and was still in use. Technically, the system consisted of 15 *Form Modules*. These modules contained 5.000 Lines of Code (LOC), written in the programming language PL/SQL, and 2.000 declarative elements, defined in the Fourth-Generation Programming Language (4GL) of the source environment. A screenshot of the running legacy system can be seen in the left side of Figure 7.1, while the result of the transformation is shown in the right side.

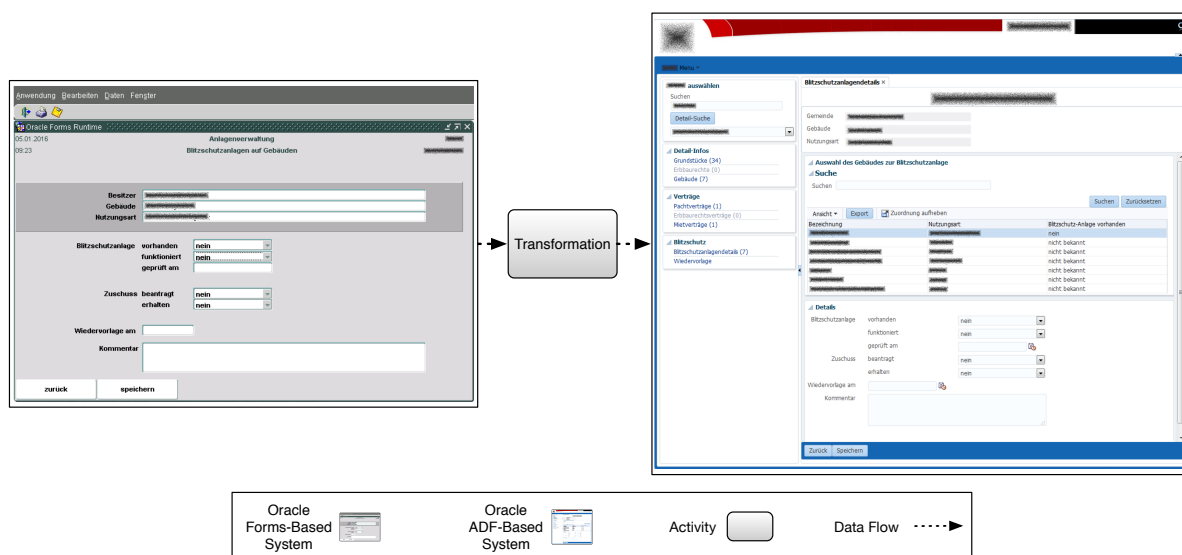


Figure 7.1 Real estate management legacy system in the source environment (left), the transformed system in the target environment (right)

Oracle ADF had been chosen as the target environment of the transformation. Note that this specific modernization scenario and its challenges are discussed in Section 3.1. It can be summarized that both environments are significantly different, for example, related to their architecture or how to realize certain functionalities. Therefore, when performing design decisions for the resulting target system of the transformation, one needs to decide whether to adapt the system to the new environment, or not. While the former case might require comprehensive changes, like an architectural restructuring, the latter case significantly decreases the non-functional properties of the resulting system. For this reason, all design decisions in this project had been driven by the intention to truly adapt the system to the new environment, i.e., to modernize it. Enabling the development of a transformation method for

this particular modernization scenario was the original motivation for the definition of the MEFiSTo framework [GGS12].

From an organizational perspective, the project took place in an industrial context. It had been performed together with an industrial partner, i.e., a mid-sized enterprise that was specialized on developing Oracle-based software systems. In total, five people had been assigned to the project, whereby some people had been associated with multiple roles. Two persons were responsible for the development of the transformation method. Thereby, one person took the role of a *Modernization Expert*, as the person had profound knowledge of the source and the target environment. The other person took the role of a *Tool Specialist*, as his knowledge was focused on software reengineering and related tools. The same two persons additionally took the role of *Tool Developers*, as they were experienced in this area, too.

Three other people were included during the *Transformation* activity. One of them had been experienced with the legacy system, wherefore the person took the role of a *System Expert*. The other two persons took the role of *Software Developers*. One of them had been experienced in developing software for both environments. In contrast, the other one only had general experience in developing software, but did not develop any software in either of both environments. As a side effect of the modernization project, the developers should become familiar with the target environment.

Subsequently, we describe the application of the MEFiSTo framework to transform the real estate management legacy system. We briefly describe the enactment of all activities of the method engineering process and summarize the main artifacts and findings. In addition, we describe the transformation of a selected set of functionalities in detail. The transformation of these functionalities shall exemplify the modernization that took place.

7.2.1 Situational Context Identification

The emphasis of this activity is put on identifying the concepts within the legacy system to transform and discussing the influence factors for each one separately (cf. Section 6.3.1). The concept model of the real estate management legacy system that had been defined by the two experts can be seen in Figure 7.2. In total, 23 concepts had been identified. Note that we already visualize the configured concept model, whereby the configuration only took place during the next activity (cf. Section 6.4).

As can be seen, the concept model shares some similarities with the one of the running example that has been developed in Section 6.3.1. This is due to the fact that the running example covers the same modernization scenario, namely the transformation from Oracle Forms to Oracle ADF. However, related to the *Model* concern, some differences can be found. Some concepts had been refined, like the *View* concept. In this system, a distinction is made

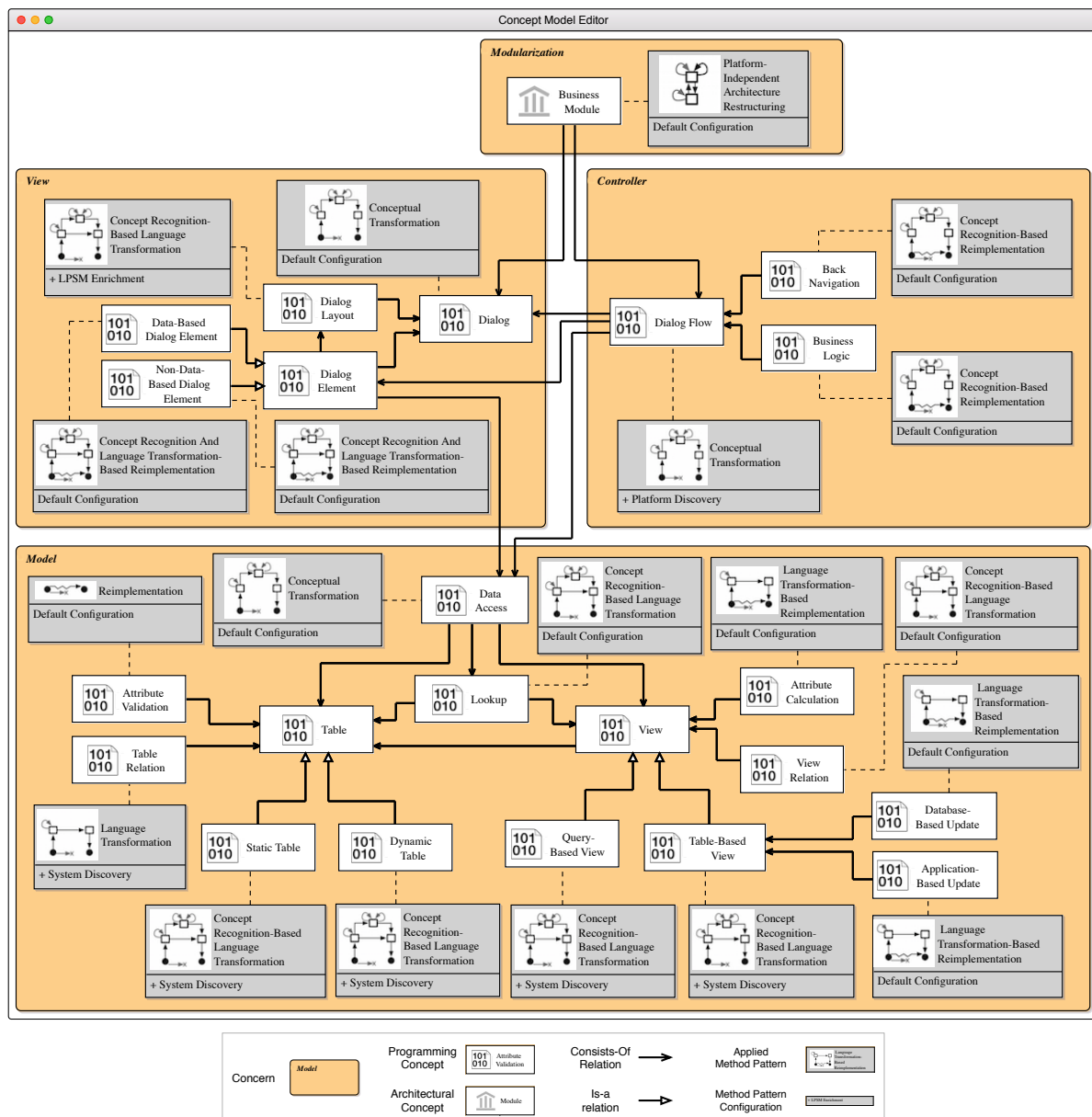


Figure 7.2 Configured concept model for the real estate management legacy system

whether the entries of a *Table* are changed during runtime (*Dynamic Table*), or not (*Static Table*). Also, additional concepts had been identified, like the *Lookup* concept. A *Lookup* is used to provide a meaning for an internal value. For example, if a person should select a customer, but the customer is internally only represented by an ID. Then, a *Lookup* is used to provide a meaning for that ID, like the name of the customer¹.

¹http://docs.oracle.com/cd/E28271_01/fusionapps.1111/e15524/bs_lookups.htm (accessed March 22th, 2016)

Besides additional concepts, additional concerns had been addressed, too. The *View* and *Controller* concerns comprise concepts that are related to the equally named layers in the target architecture (cf. Section 3.1). Concepts of the *View* concern describe the user interfaces of the system (*Dialog*), their layout (*Dialog Layout*), as well as their contained user interface elements (*Dialog Element*). Concepts of the *Controller* concern process user interaction which can result in a flow between user interfaces (*Dialog Flow*), specific ways of navigating (*Back Navigation*) or the execution of application logic (*Business Logic*).

The *Modularization* concern had been refined, i.e., its contained *Business Module* concept is a refinement of the *Module* concept, described in Section 4.3. The general idea is to modularize the legacy system by identifying parts of the system that belong together from a business perspective. In particular, we aimed to identify sets of *Form Modules* whereby the members of each set should be associated with a common business task.

For the remainder of this section, we focus on a subset of three concepts. As several concepts related to the *Model* concern were already discussed in detail (cf. Chapter 5), we use the *Dialog*, *Dialog Flow*, and *Business Module* concepts of the *View*, *Controller*, and *Modularization* concern, respectively. They can be seen in the upper center of Figure 7.2. Note that the *Business Module* concept consists of the other two concepts. This is due to the fact that we intended to modularize, i.e., cluster, the system based on the contained navigation flows. We assumed that a flow between two user interfaces which are part of different *Form Modules* is a hint that the modules are associated with a common business task.

Besides defining the concept model, an important activity of the situational context identification is the discussion of influence factors for the identified concepts (cf. Section 6.3.2). Here, we discuss factors of the *Dialog Flow* concept. To do so, we need to have a general understanding of its realization in the legacy and target system, which is shown in Figure 7.3.

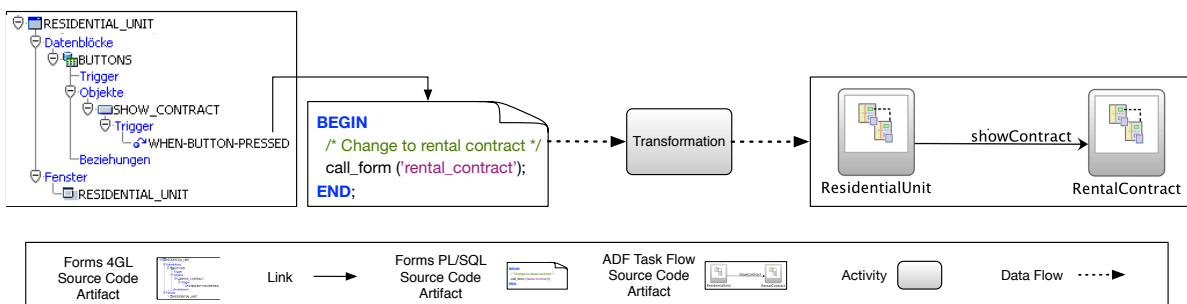


Figure 7.3 Legacy and target realization of the *Dialog Flow* concept

In the left side of the figure, the realization of the concept in the legacy system is shown. A *Dialog Flow* was realized imperatively in PL/SQL code by calling a platform-specific function, e.g., `call_form`. The code was part of a Trigger that got executed whenever an event

occurred, e.g., when a corresponding button was pressed. By invoking the platform-specific function, a new *Form Module* got executed [Ora00b, pp.25-26]. From a users' perspective, a change to another *Dialog*, i.e., user interface, was performed.

In the right side of Figure 7.3, the envisioned realization of the concept in the target system is shown. We intended to realize *Dialog Flows* by using the declarative language provided by the target environment, i.e., we intended to use so-called *Task Flows*². Among other things, *Task Flows* can be used to represent user interfaces (*ResidentialUnit* and *RentalContract*) and model the flow between them (*showContract*).

As the concepts would be realized significantly different in both systems, i.e., imperatively and declaratively, the experts assumed that the *Conceptual Transformation Pattern* would be suitable. This hypothesis was reinforced by the influence factors identified that relate to the method fragments of the pattern (cf. Section 5.5.2):

- For PL/SQL, a cost-effective commercial parser was available for purchase, while libraries provided by Oracle itself could be used to parse the declarative parts of the source code and to generate a model of the runtime environment.
- Platform-specific metamodels could be automatically reverse engineered based on artifacts provided by Oracle.
- Coding conventions for dialog flows existed, i.e., they had been realized homogeneously within the legacy system. Techniques were known [Nie+02] that address the tool-supported extraction of knowledge by exploiting such conventions.
- The *UI* package of the OMG standard KDM could be used as a metamodel to represent the *Dialog Flows* on a platform-independent layer.
- The complexity of the model transformation to concretize the platform-independent representation of the *Dialog Flows* was deemed to be low.
- The code generation rules could partly be automatically derived by reverse engineering existing applications in the target environment.

7.2.2 Transformation Method Construction

Based on the identified situational context, the actual transformation method was constructed. As a first step, a method pattern had been selected and coarse-granularly configured for each concept identified (cf. Section 6.4.1). The resulting concept model is shown in Figure 7.2.

It had been decided to convert 12 concepts, i.e., to transform them automatically. In contrast, 7 concepts were transformed semi-automatically and 1 concept was manually reimplemented.

²http://docs.oracle.com/cd/E23943_01/web.1111/b31974/taskflows.htm#ADFFD1631 (accessed March 22th, 2016)

Note that the concept model does not provide information on the size of a concept, i.e., the amount of its instances or its scope. It has been assumed that the size of reimplementation activities that are part of the *Model* concern are low. In contrast, reimplementation activities in the *View* and *Controller* concern are assumed to be large.

For each configured concept a horseshoe model had been automatically derived, before different horseshoe models had been integrated (cf. Section 6.4.2). The integrated horseshoe model for the *Dialog*, *Dialog Flow*, and *Business Module* concepts is shown in Figure 7.4. *Conceptual Transformation Patterns* were applied on the *Dialog* and *Dialog Flow* concepts, wherefore these concepts got represented on the platform-independent layer by a *Dialog & Dialog Flow Model*. It can be seen that a graduated enactment was performed (cf. Section 6.4.1), as the *Concretization* of that model depends on the transformation of the *Business Module* concept, i.e., the result of the architectural restructuring.

The intention of the architectural restructuring was to hierarchically cluster the legacy system, based on the *Business Module Model* [GSK14]. In this model, each *Form Module* was represented as a *component*, i.e., a unit of composition [Szy02, p.548]. Since each *Form Module* contains at least one *Dialog*, the flows between those *Dialogs* are used to model dependencies between the *components*. The idea was to find clusters in the resulting graph that have a high cohesion [YC79, pp.95-97] and a low coupling [YC79, pp.76-77] with other clusters. It was assumed that the *components* of each cluster that could be identified based on *Dialog Flow*-dependencies are closely related to a common business task.

The restructuring itself was performed by the *Restructure Business Modules* activity. It had been specified to perform the activity semi-automatically. As a first step, a set of clusters should be automatically calculated by an algorithm. It had been specified to apply the *Maximizing Cluster Approach* [PHY11] as a foundation for an automatic clustering algorithm, since the approach seemed to provide good solutions in practice. The algorithm addresses the problem of finding suitable clusters by formulating it as a multi-objective search problem. For example, the algorithm tries to maximize the sum of the edges within a cluster, while minimizing the sum of edges between clusters. For a more detailed description, we refer to [PHY11]. As a second step, an associated *System Expert* should validate the results. As multiple solutions to a multi-objective search problem exist, the expert may select a result and possibly refine it. Also, it was required that the expert provides a meaningful name for a cluster. The restructured *Business Module Model* formed the basis for restructuring the *Dialog & Dialog Flow Model*.

We want to point out that we did not create a SPEM-based representation of the transformation method as part of this modernization project. This is due to the fact that the people who were involved in the transformation were familiar with MIML. Therefore, the documentation of the method using MIML was deemed to be sufficient.

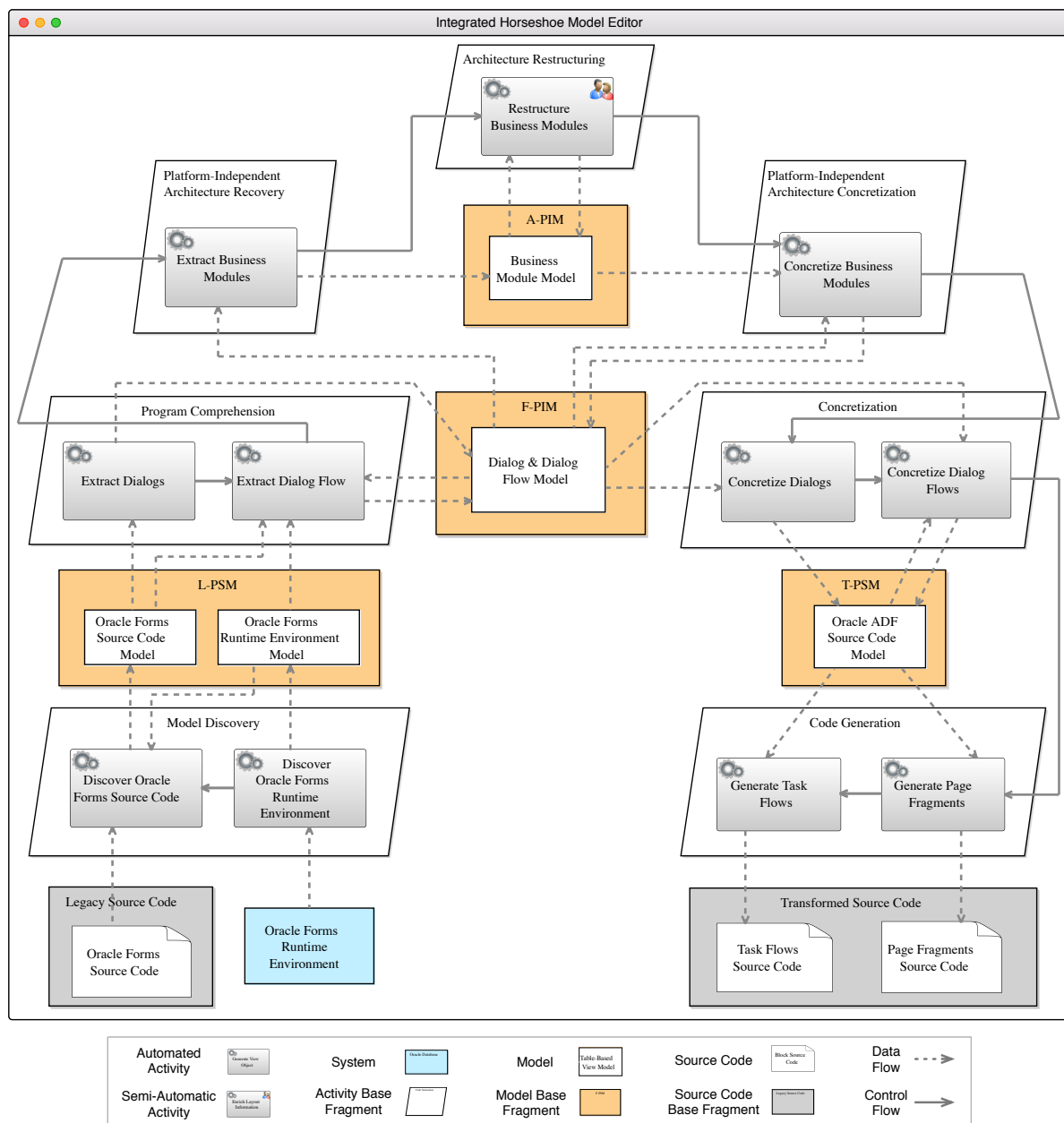


Figure 7.4 Horseshoe model for the *Dialog*, *Dialog Flow* and *Business Module* concepts

7.2.3 Tool Implementation

After the transformation method specification had been developed, required tools were implemented. Thereby, we first defined a project-independent tool infrastructure which consisted of a component-based architecture. Within this architecture, some components are project-independent and should be reused by subsequent projects, while others need to be exchanged by project-specific ones. The architecture can be seen in Figure 7.5.

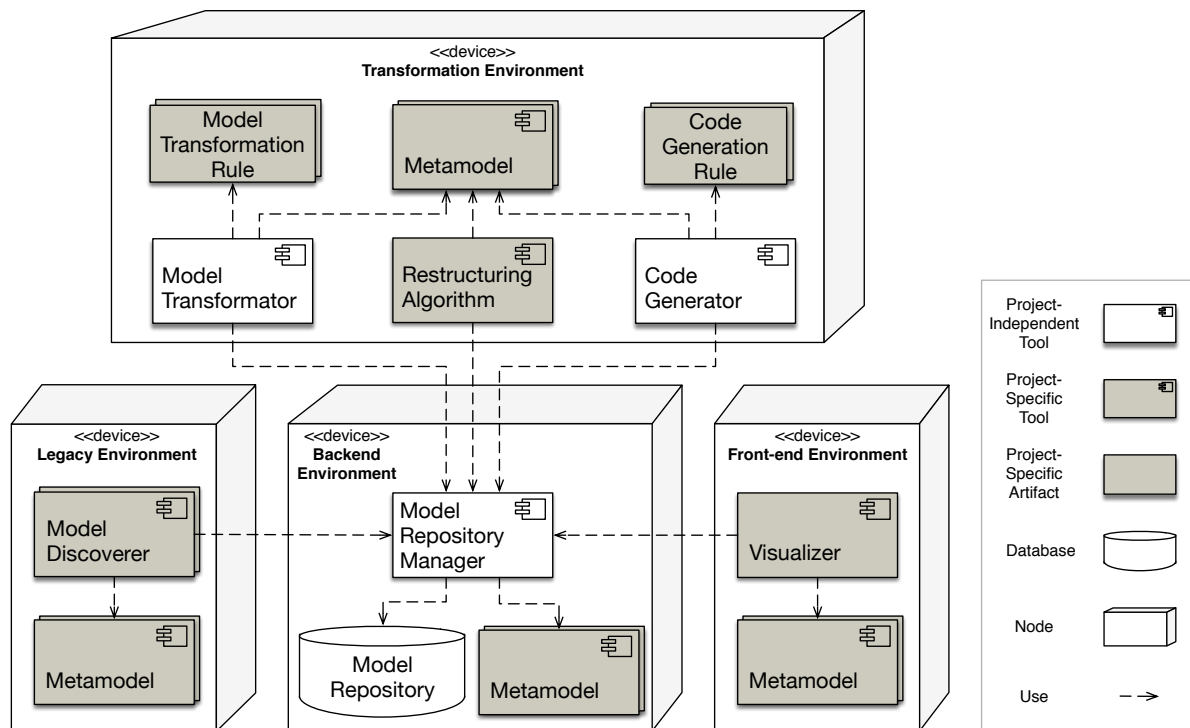


Figure 7.5 Architecture of the generic tool infrastructure

Project-independent components were only foreseen as part of two physical devices, namely the *Backend Environment* and the *Transformation Environment*. The component of the *Backend Environment*, i.e., the *Model Repository Manager*, provided capabilities to access models. It enabled persisting or retrieving them from a *Model Repository* via defined interfaces. In contrast, components of the *Transformation Environment* were responsible for actually transforming the models. The *Model Transformer* was capable of performing model-to-model transformations, while the *Code Generator* enabled performing model-to-text transformations.

From a technological perspective, we decided to use the established *Eclipse Modeling Framework (EMF)*³ to implement models and metamodels. The *Model Repository Manager* was based on the *Connected Data Objects (CDO) Model Repository*⁴ which provides capabilities to manage EMF-based models, e.g., transactional manipulations, views or queries. The *Model Transformer* was based on *Henshin*⁵. It enabled executing model transformations defined in the equally named transformation language. The *Code Generator* had been based on *Acceleo*⁶, which itself is based on the *MOF Model to Text Transformation Language (MOFM2T)* [OMG08b] standard of the OMG.

³<https://eclipse.org/modeling/emf/> (accessed March 22th, 2016)

⁴<http://projects.eclipse.org/projects/modeling.emf.cdo> (accessed March 22th, 2016)

⁵<http://www.eclipse.org/henshin/> (accessed March 22th, 2016)

⁶<http://www.eclipse.org/acceleo/> (accessed March 22th, 2016)

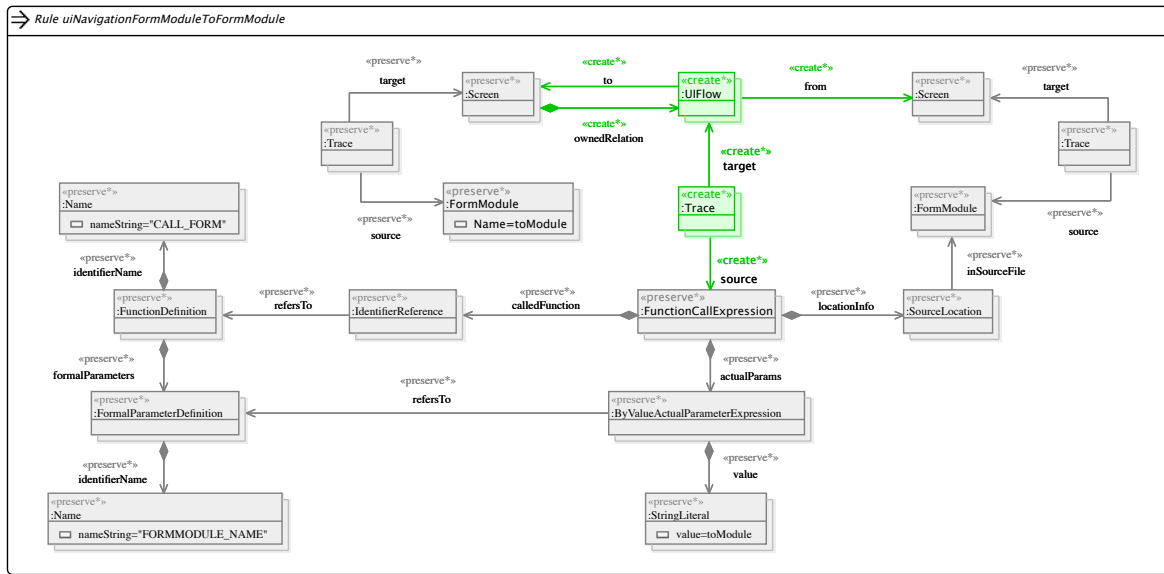


Figure 7.6 Model transformation rule to extract flows between dialogs, implemented in Henshin

A *Model Transformation Rule* is shown in Figure 7.6. It had been specified in *Henshin* to extract the *Dialog Flow* concept by formalizing an underlying implementation variant as a pattern on the ASG [Nie+02]. In this project, we developed 432 rules in total. The model transformation rule is an example for a project-specific artifact that was required by a project-independent component, i.e., the *Model Transformator*. The same is true for *Code Generation Rules* and *Metamodels*, these artifacts and components were required by project-independent components, too. Related to *Metamodels*, we reused the ADM-based metamodels that had been implemented as part of the MoDisco framework [Bru+14], namely KDM and ASTM (cf. Section 2.1.3). Note that we decided to deploy the used metamodels on all devices on which components were executed that had dependencies on them. This was due to the fact that this eased the implementation of those components.

The remaining environments contain components that had been specifically developed for the project. The *Model Discoverer* of the *Legacy Environment* was responsible for injecting platform-specific models of the artifacts and systems. The component was executed within an already existing device, i.e., a system on which the legacy system was still being developed and executed. This ensured that all prerequisites in terms of runtime libraries and components were in place. Technically, the Oracle Forms source code artifacts were parsed by using the *Java Development API (JDAPI)* [Ora09a, p.7] provided by Oracle. The same library could also be used to derive a model of the runtime environment of Oracle Forms. Source Code parts that

were written in PL/SQL were processed by applying a commercial parser⁷. The schema of the associated database had been extracted by parsing the *Data Dictionary*⁸.

The *Visualizer* of the *Front-end Environment* provided capabilities to visualize and manipulate models stored in the repository. It had been used primarily by the system expert during the architectural restructuring. The restructuring itself, i.e., the automatic calculation of a clustering solution was performed by a dedicated *Restructuring Algorithm* component. Technically, we used the *Opt4J Framework*⁹. This framework enables computing solutions to multi-objective optimization problems by applying genetic algorithms [Luk+].

7.2.4 Transformation

After required tools were implemented, the actual transformation of the legacy system into the new environment took place. The automatic conversion was performed by using the developed tools on the legacy source code. In total, 27.000 LOC (about 50% of the code) were generated by the tool chain. This resulted in a model repository consisting of nearly 37K entities. The L-PSM accounted for the largest part (70%), followed by the T-PSM (13%) and the PIMs (8%). The remaining entities belonged to models that represented infrastructural information, consisting of traceability links (7%) or language extensions, like stereotypes (2%).

An excerpt of the arising models and entities when converting the *Dialog*, *Dialog Flow* and *Business Module* concepts can be seen in Figure 7.7. Note that the figure shows the enactment of the transformation method that has been specified in Figure 7.4 on page 198.

First, the source code of the legacy system as well as the Oracle Forms runtime environment were parsed and a semantic analysis was performed. The L-PSM that resulted can be seen in the lower left of Figure 7.7. In this example, the ASG of the WHEN-BUTTON-PRESSED Trigger that was contained in the Form Module named Residential Unit is shown. It contains the imperative realization of a *Dialog Flow* to the Form Module named Rental Contract by invoking the platform-specific function named `call_form` (cf. Figure 7.3, page 195).

The L-PSM was used to extract a model of the system's *Dialogs* and the navigation flow between them. This had been realized by executing model transformations that formalize implementation variants as patterns on the ASG (cf. Figure 7.6). The F-PIM that resulted is shown in the upper left of Figure 7.7. As can be seen, the *Dialog Flow* is now explicitly modeled, e.g., the flow from the Screen named Residential Unit to the Screen named Rental Contract. Note that we used the notation of a Screen instead of Dialog. This is due to the fact that we used the KDM to represent this platform-independent model.

⁷<http://www.sqlparser.com> (accessed March 22th, 2016)

⁸http://docs.oracle.com/cd/B28359_01/server.111/b28318/datadict.htm (accessed March 22th, 2016)

⁹<http://opt4j.sourceforge.net> (accessed March 22th, 2016)

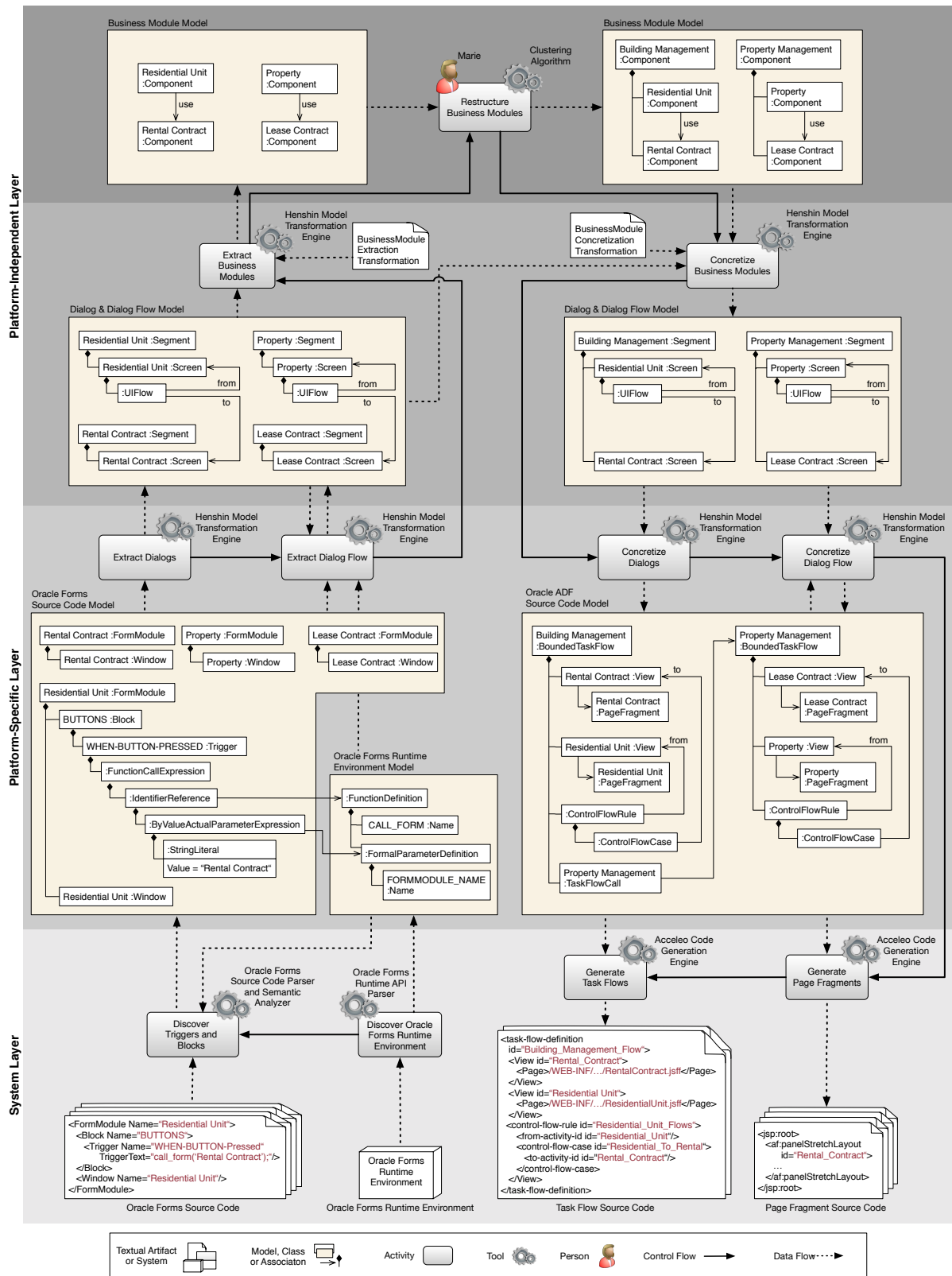


Figure 7.7 Enacting a transformation method to transform the *Dialog*, *Dialog Flow* and *Business Module* concepts of the real estate management legacy system

After the F-PIM had been created, it was used to extract an architectural view on the system. As discussed in Section 7.2.2, we interpreted each Form Module as a component while flows between contained *Dialogs* became dependencies between them. An excerpt of the A-PIM that resulted is shown in the upper part of Figure 7.7. To give an idea of the subsequent restructuring that was performed, the complete A-PIM is shown in Figure 7.8.

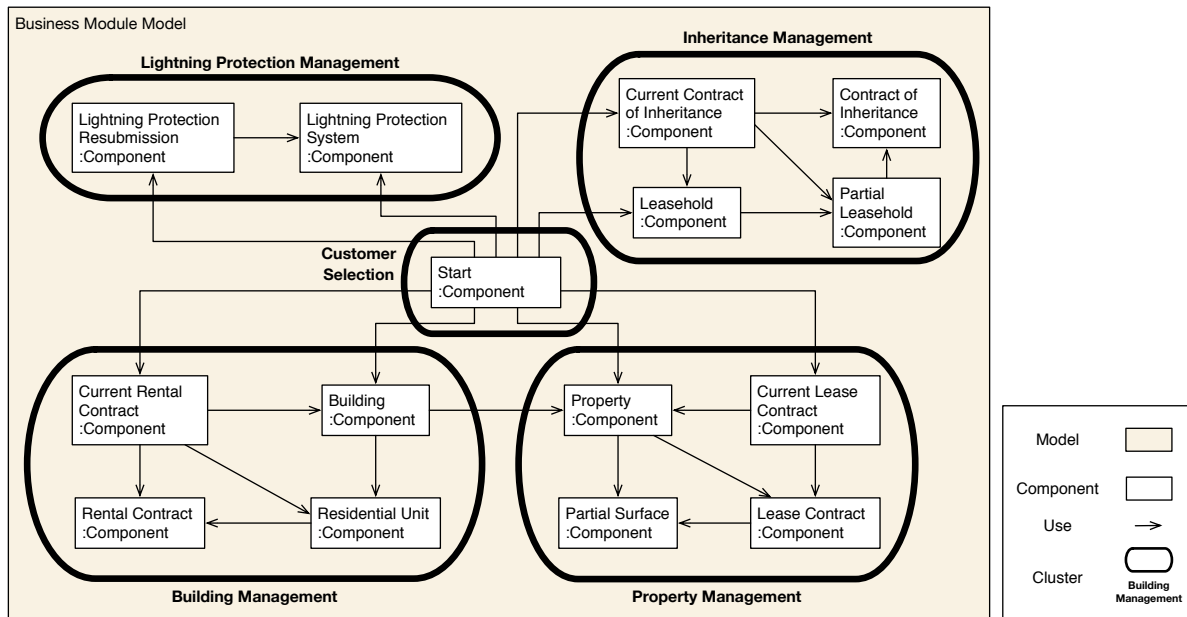


Figure 7.8 Result of a semi-automatic architectural restructuring of the real estate management legacy system by clustering related *Business Modules*

The model shows all Components that were part of the legacy system as well as the dependencies between them. The clusters indicated are the final ones the system expert had identified. The expert chose these clusters as the components of each cluster were closely related from a business perspective. For example, viewing the lease contract of a property first required to select the property itself via the *Dialog* contained in the Property Component. Thereafter, the lease contract could be shown by changing to the *Dialog* contained in the Lease Contract Component. In this example, both Form Modules are related to a common set of business task, namely tasks related to the *Property Management*.

Note that the cluster named *Customer Selection* had a special role as it served as an entry point to the application. In this *Dialog*, a customer could be (globally) selected for whom the different concerns could be managed, e.g., its properties or buildings.

After the restructuring of the A-PIM had been performed, the changes were reflected to the F-PIM. This is shown in the upper right of Figure 7.7. As can be seen, Form Modules (represented as Segments) had been merged according to the clustering. For example, the

Segment named Residential Unit and the Segment named Rental Contract had been merged into a Segment named Building Management.

The restructured *Dialog & Dialog Flow Model* was thereafter concretized in the target environment, i.e., a model representing the *Task Flows* of the system was derived (cf. Figure 7.3, page 195). The resulting T-PSM is shown in the lower right of Figure 7.7. In this excerpt, the Bounded Task Flow named Building Management is shown. It models the *Dialog Flows* inside the equally named cluster as well as flows to other clusters (cf. Figure 7.8). Finally, the model was used to generate source code in the target environment.

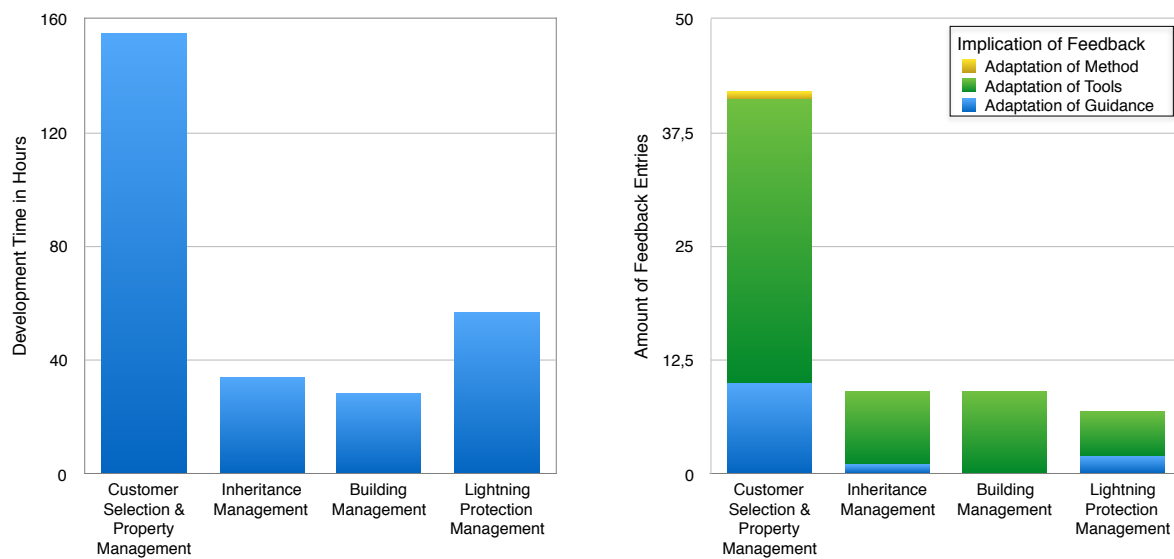


Figure 7.9 Effort for transforming the real estate management legacy system (left), amount of feedback exchanged between involved roles (right)

In Figure 7.9 some key figures about the overall transformation are shown. In the left side of the figure, the effort for transforming the legacy system in terms of hours spent is visualized. We separated the transformation into four transformation packages according to the clusters identified, whereby the first package consisted of two clusters.

As can be seen, most effort was spent in transforming the first package. The reason for this is twofold. First, the associated software developers needed to get familiar with the generated source code and the reimplementation instructions given. Also, one developer was not familiar with developing in the target environment at all, which led to slower development speed in the beginning. Second, we changed the target environment during the first transformation package, i.e., we switched the version of the ADF framework. Initially, we used the newest version which turned out to be too unstable at that moment. While we assumed that the adaptation of developed tools would not be too extensive when changing the version, some adaptations were still required which lead to increased development times.

The second and third transformation packages required notably less effort than the first one. This is due to the fact that the first three packages were quite similar and the developers became used to the reimplementation. The reason for the increased effort in the last transformation package is twofold. First, it has been only transformed by one developer, while the previous packages were transformed by two of them. The developer responsible for the last package was less experienced and therefore slower. Second, the modules contained in the last package were notably different compared to the ones of the previous packages. In particular, more instances of the *Attribute Validation* concept were present (cf. Figure 7.2, page 194). As this concept needed to be completely reimplemented, i.e., no code was generated, additional effort in the transformation of this package was required.

Issues with the generated source code or the transformation method itself were primarily found by the software developers when they performed reimplementation tasks. To use this knowledge, we exchanged structured feedback between the different persons involved in the transformation [Gri+14]. In this project, we generated source code for the software developers when they started working on a transformation package. Thereafter, we periodically gathered feedback from them in order to improve the method and/or the developed tools for the next transformation package.

In the right side of the figure, the amount and type of feedback exchanged as part of each transformation package is shown. We classified the feedback based on the implication it had. Only the feedback that arose during the first transformation packages resulted in an adaptation of the transformation method. In particular, the *Back Navigation* concept had not been discovered during the development of the method, but was added due to the feedback.

Feedback that resulted in the adaptation of tools or reimplementation guidance documents was mainly related to bugs that had been discovered. For example, generated code did not comply with coding conventions, or cases had been missing in guidance documents. As bugs were fixed, the feedback decreased over time

This concludes a description of the application of the MEFiSTo framework to transform the real estate management legacy system. Subsequently, we describe the second feasibility study of this thesis.

7.3 Feasibility Study 2: Oracle Reports to Jasper Reports

In this feasibility study we transformed a *Report* from the logistics domain into a new environment. A report can be seen as a program that enables extracting data from a data source and outputs that data in a human-readable and printable format. The purpose of the report we transformed was to output a list of transport units that either were delivered to or returned from

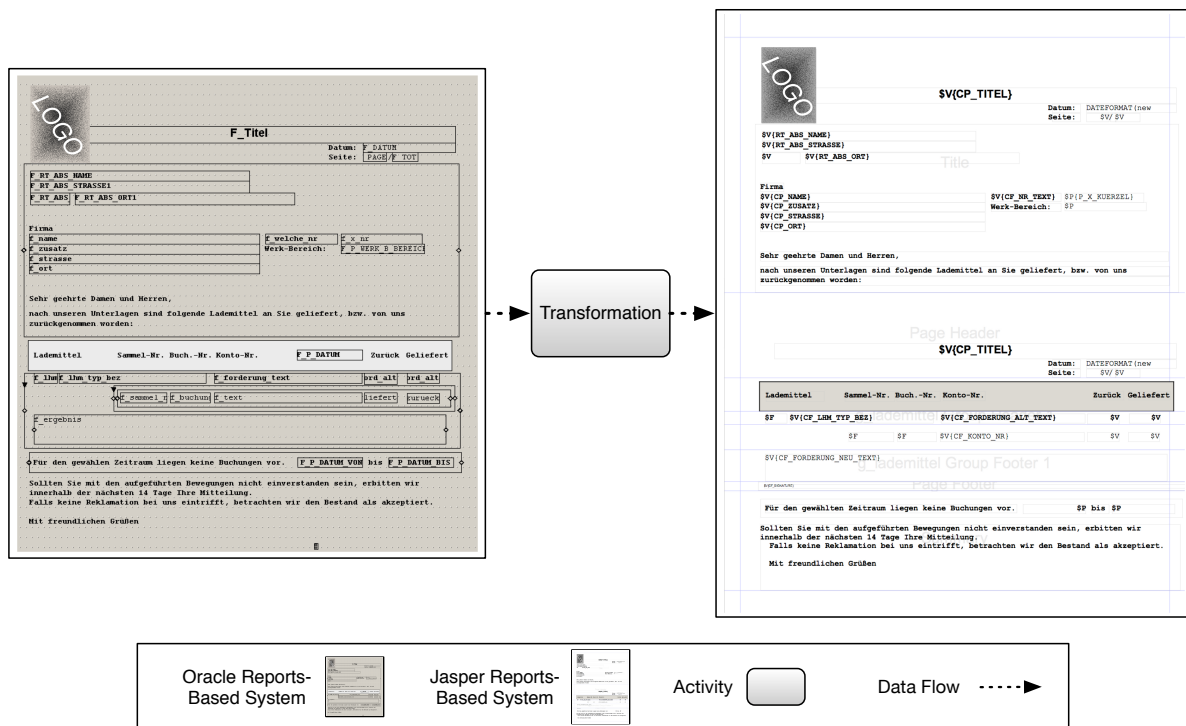


Figure 7.10 Legacy report in the development environment of the source environment (left), transformed report in the development environment of the target environment (right)

a customer. The list could be generated in the form of a letter which would have been sent to the customer. The report had been originally implemented around 1998 using the platform of Oracle Reports and was still in use. Technically, the report contained about 1.000 LOC, written in the programming language PL/SQL, and 150 declarative elements defined in the Fourth-Generation Programming Language (4GL) of the source environment. As can be seen, the legacy system is notably smaller than the one of the first feasibility study. However, we chose a smaller system on purpose as the main intention of this study was to demonstrate that the MEFiSTo framework is not limited to a specific environmental change.

Oracle Reports is a reporting tool to develop and execute reports that gather data from an associated database. The most important part when developing a report is to design its graphical layout, i.e., to design the content of the pages that are output when the report gets executed. Usually, reporting tools provide Integrated Development Environments (IDEs) for this purpose. These IDEs enable to design the pages of a report by placing elements on them. Examples for such elements are static texts or placeholders for data. A screenshot of the legacy system, i.e., the legacy report, can be seen in the left side of Figure 7.1. It is shown in the IDE of the source environment, while the corresponding transformed report is shown in the right side.

Jasper Reports had been chosen as the target environment of the transformation. We concluded that it would be possible to transform the report into that environment since it provides similar capabilities as Oracle Reports. However, like in the first feasibility study, we noticed that the way in which certain functionalities should be realized, differed significantly. Nevertheless, we intended to adapt the report to the new environment, i.e., to modernize it.

From an organizational perspective the project took place in an industrial context, too, as it had been performed together with an industrial partner. In total, two people had been assigned to the project. One person took the role of a *Modernization Expert*, as the person had profound knowledge of the source and the target environment. The other person took the role of a *Tool Specialist*, as its knowledge was focused on software reengineering and related tools. In addition, both persons took the roles of *System Experts* and *Tool Developers*.

Note that no *Software Developer* was part of the project. This is due to the fact that we did not perform manual reimplementations activities. Although enacting these activities would have been necessary to completely transform the report, we concluded that their enactment was not necessary to demonstrate the feasibility of the transformation. Solely applying the tools to generate parts of the report automatically resulted in a report in the target environment which could be executed. It only contained flaws related to the layout and the data fetching which would have required some manual rework.

Subsequently, we describe the application of the MEFiSTo framework to transform the legacy report. We briefly describe the enactment of all activities of the method engineering process and summarize the main artifacts and findings. In addition, we describe the transformation of a selected set of functionalities in detail. The transformation of these functionalities shall exemplify the modernization that took place.

7.3.1 Situational Context Identification

The emphasis of this activity is put on identifying the concepts within the legacy system to transform and discuss influence factors for each one separately (cf. Section 6.3.1). The concept model of the legacy report that had been defined by the two experts can be seen in Figure 7.11. In total, 7 concepts had been identified. Note that we already visualize the configured concept model, whereby the configuration only took place during the next activity (cf. Section 6.4).

We distinguished three elementary concerns, namely *Data*, *Program* and *User Interface*. Concepts of the *Data* concern are related to the data structures defined and used by the report. The *Data Model* concept describes such an internal data structure which is usually defined explicitly by a dedicated language. It contains definitions of datasets that are populated when the report gets executed (comparable to Blocks and Items in Oracle Forms).

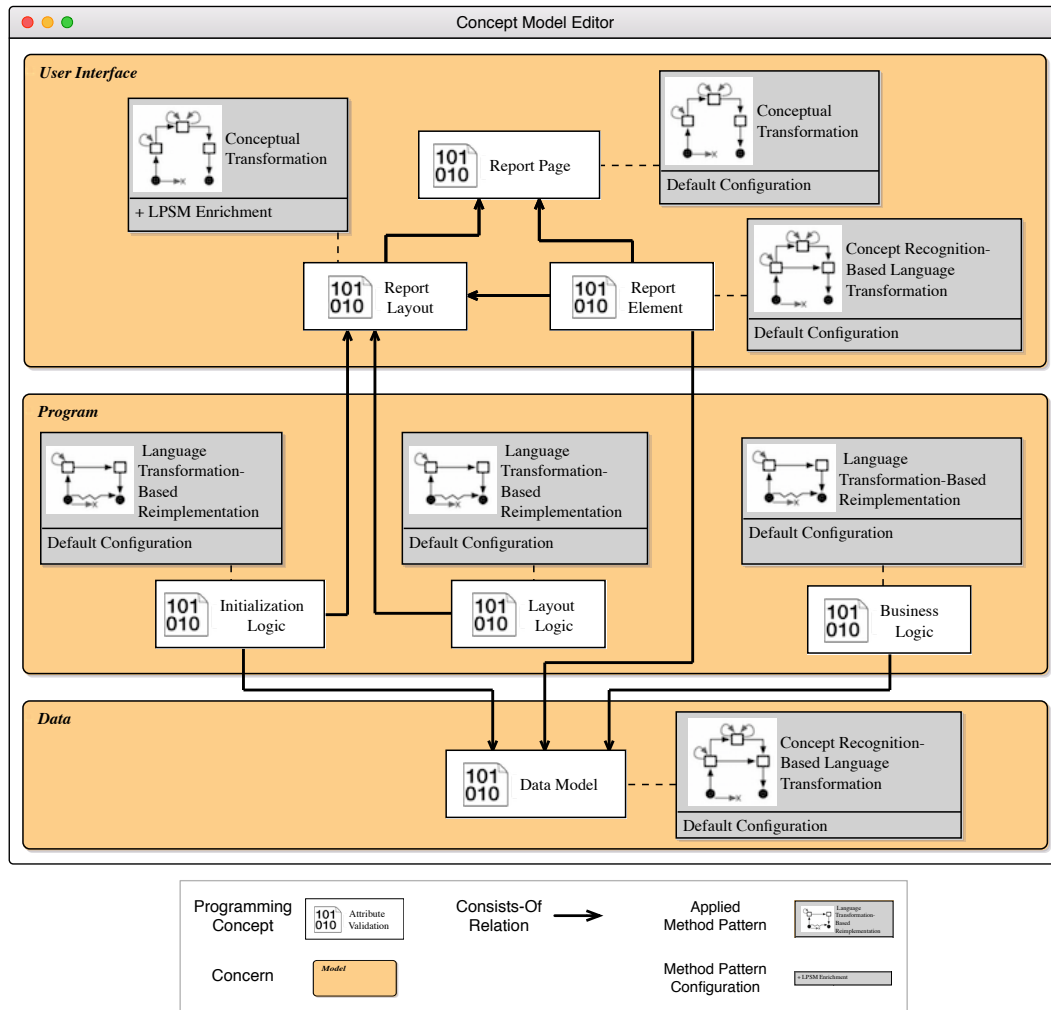


Figure 7.11 Configured concept model for the reports legacy system

The *Program* concern is related to the dynamic, i.e., programmatic parts of the report. The *Business Logic* concept describes application logic that resides within the report which interprets the data of the data model. *Initialization Logic* describes the functionality which initially sets the context of the report, like the current date. The *Layout Logic* describes the realization of dynamics within the structure of the user interface.

The *User Interface* concern is related to the graphical representation of the report. It consists of the *Report Page* itself, its layout and the elements which are placed on the page. In the remainder of this section, we focus on the last two concepts, namely the *Report Layout* and the *Report Element*. To discuss the influence factors of these two concepts (cf. Section 6.3.2), we need to have a general understanding of their realization in the legacy and target system. This is illustrated in Figure 7.12.

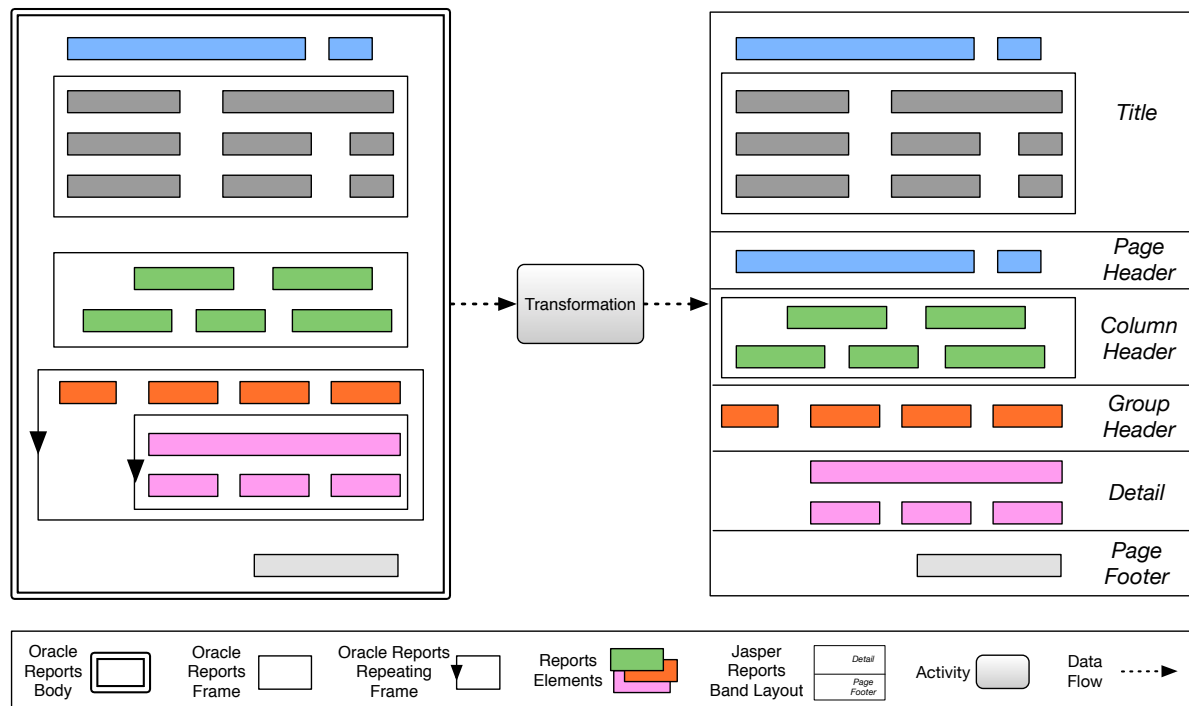


Figure 7.12 Legacy and target realization of the *Report Layout* and *Report Element* concepts

In the left side of the figure, the realization of the concepts in the legacy system is shown. Examples for *Report Elements* in Oracle Reports are Texts or Fields [Ora09b, pp.55-57]. The latter element type can be associated to a data source, e.g., a column in a database. The *Report Layout* is defined by the Body, i.e., the page itself, as well as by the contained, absolutely positioned Frames and Repeating Frames [Ora09b, pp.54-55]. Intuitively, those language constructs represent containers in which *Report Elements* can be placed. Frames provide means to configure how often its contained elements get printed. For example, a Frame can be configured to be solely printed on the first page or on every page. Repeating Frames are special in the sense that they are printed depending on the datasets of contained fields. Such a frame gets printed for each dataset associated with the fields. For example, if the fields are associated to database columns, then the frame gets printed for each row. If Repeating Frames are nested, they form a master-detail relationship (cf. Section 4.3).

In the right side of Figure 7.12, the envisioned realization of the concepts in the target system is shown. Examples for *Report Elements* in Jasper Reports are Static Texts or Text Fields [Tib15, pp.54-56]. In general, the elements provided by both environments are comparable. The *Report Layout* is defined by Bands which are arranged below each other. Each band has a type and associated characteristics [Tib15, pp.21-22]. For example, the Title Band is only printed on the first page, while the Page Header gets printed on every page.

The *Report Element* concept would be realized comparably in both environments, i.e., for each legacy report element a counterpart in the new environment could be found. Therefore, the experts assumed that the *Language Transformation Pattern* would be suitable, efficient and effective. In contrast, the *Report Layout* concept would be realized significantly different. A transformation requires identifying different *parts* of the legacy page. For example, the *Title* part will be an area at the top of the legacy page that only contains *Report Elements* which are printed once. Therefore, the experts assumed that the *Conceptual Transformation Pattern* (cf. Section 5.5.2) or the *Reimplementation Pattern* (cf. Section 5.5.3) would be suitable. Subsequently, the identified influence factors for the latter pattern are shown:

- For a human, it is straightforward to map the frames of the framework to the different types of bands. Corresponding guidance can be provided.
- Transforming the layout requires a pixel-precise alignment of Bands. This would be particularly error-prone when performed manually.
- A manual transformation of the layout of the report would prevent an automatic transformation of the elements of the report.

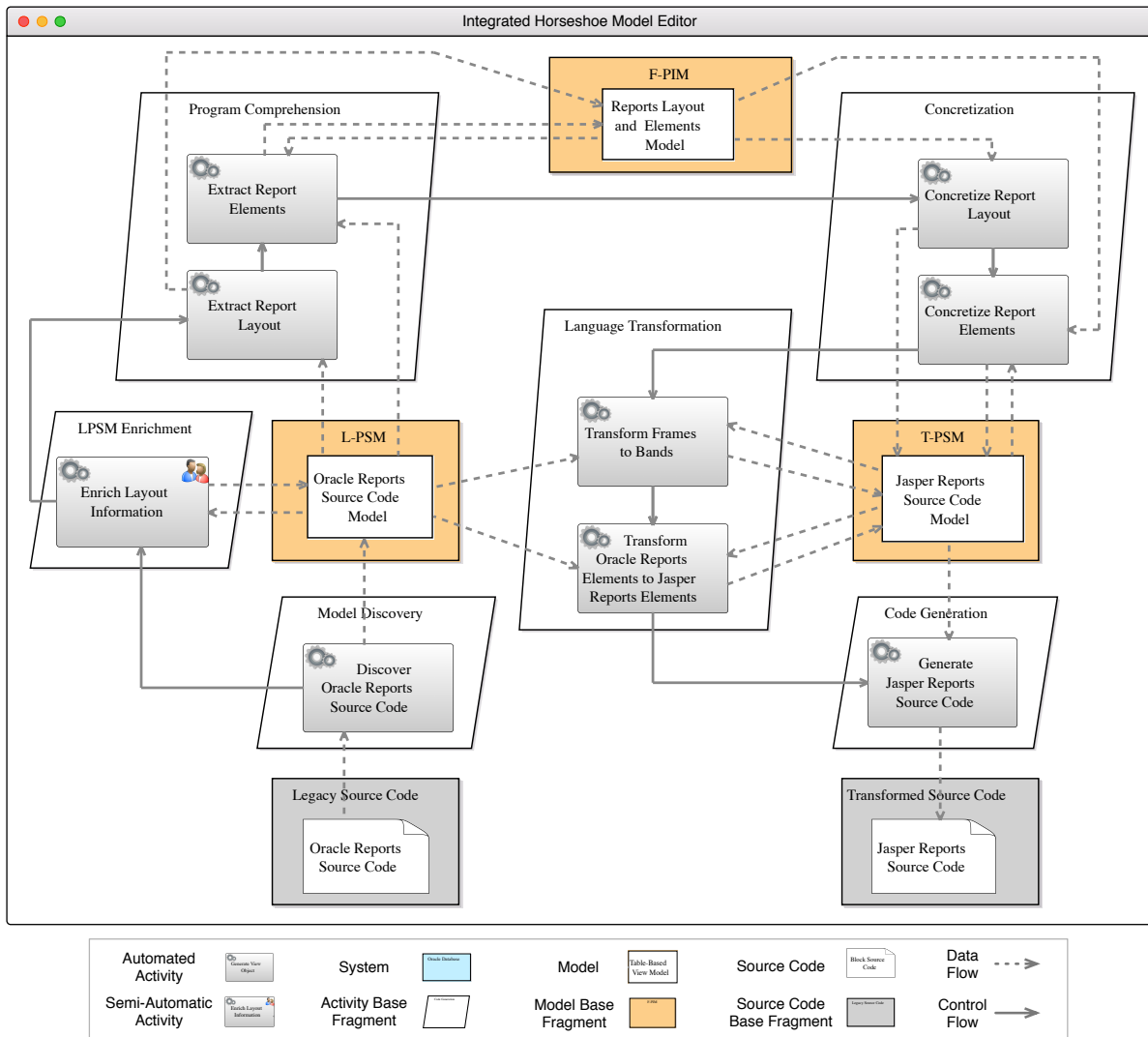
7.3.2 Transformation Method Construction

Based on the identified situational context, the actual transformation method was constructed. As a first step, a method pattern had been selected and coarse-granularly configured for each concept identified (cf. Section 6.4.1). The resulting concept model is shown in Figure 7.11.

It had been decided to convert four concepts, i.e., transform them automatically. The remaining three concepts were transformed semi-automatically. For each configured concept a horseshoe model had been automatically derived, before different horseshoe models had been integrated (cf. Section 6.4.2). The integrated horseshoe model for the *Report Layout* and *Report Elements* concepts is shown in Figure 7.13.

We chose to transform the *Report Layout* concept by applying the *Conceptual Transformation Pattern* instead of the *Reimplementation Pattern*. This was mainly for the reason that a manual transformation would prevent an automatic transformation of the *Report Elements*. As a result, the layout of the report should be explicitly represented by modeling its different parts, e.g., its title, page header or footer on the platform-independent level. This is the purpose of the *Reports Layout and Elements Model*.

The *Concept Recognition-Based Language Transformation Pattern* had been chosen for the *Report Element* concept. As discussed in the previous Section 7.3.1, it would have been suitable to transform the latter concept using the *Language Transformation Pattern*. However, as the *Report Layout* was transformed on a higher level of abstraction, we decided to represent the

Figure 7.13 Integrated horseshoe model for the *Report Layout* and *Report Element* concepts

Report Elements by placeholder on this level, too. Due to this, the design of the complete page could be seen on the platform-independent level. We assumed that this would be beneficial to evaluate the correctness of the transformation.

Related to the horseshoe model, i.e., the method specification, we want to point out two things: First, we included a semi-automatic *Enrichment* activity which can be seen in the middle left of Figure 7.13. The activity should be performed after the report was discovered but before the *Report Layout* got extracted. The associated tool should provide a view for associated *System Experts* which shows the different Frames and elements placed on the Body. The experts should annotate the purpose of them, e.g., whether the Frame is used as a title, a page header or a page footer. While it would be possible to automate this step, we assumed that

it would cost too much effort due to the various cases that needed to be considered. In this way, we included the benefit that was also associated with the *Reimplementation Pattern*, i.e., the fact that a human can easily identify the purpose of frames.

Second, several tools that automate activities were not realized by model transformation rules, i.e., mappings between language elements. As they required computation-intensive manipulations, they were realized programmatically by dedicated tools. Examples are the reverse engineering tools that perform the *Extract Report Layout* and *Extract Report Element* activity, respectively. Representing the different parts of the layout required to re-calculate the positioning of objects. This was necessary as the position of objects in Oracle Forms are specified in centimeters, while pixels are used in Jasper Reports. Also, the relation between different parts needs to be calculated. As this can be easier realized in an imperative programming language, we decided to develop dedicated components for this purpose.

We want to point out that we did not create a SPEM-based representation of the transformation method as part of this modernization project. This is due to the fact that the people who were involved in the transformation were familiar with MIML. Therefore, the documentation of the method using MIML was deemed to be sufficient.

7.3.3 Tool Implementation

After the transformation method specification had been developed, required tools were implemented. For the second feasibility study, we used the same architecture as for the first one, shown in Figure 7.5 on page 199. The project-independent components were directly reused as well as the KDM metamodel. Except for the *Restructuring Algorithm*, all other components had been exchanged with project-specific ones. We developed a *Model Discoverer* for Oracle Reports, required *Metamodels*, a *Visualizer* component to annotate layout information as well as *Code Generation* and *Model Transformation Rules*.

While we did not use a *Restructuring Algorithm*, we used various other components that also performed computation-intensive manipulations of the models stored in the model repository. As described in the previous section, the activities named *Extract Report Layout* and *Extract Report Element* required the recalculation of positions. For both activities, we developed project-specific components to automate them.

7.3.4 Transformation

After required tools were implemented, the actual transformation of the legacy system into the new environment took place. The automatic conversion was performed by using the developed tools on the legacy source code. This resulted in a model repository consisting of nearly

3K entities. The L-PSM accounted for the largest part (70%), followed by the T-PSM (13%) and the F-PIM (5%). The remaining entities belonged to models that represented infrastructural information, consisting of traceability links (8%) or language extensions like stereotypes (4%).

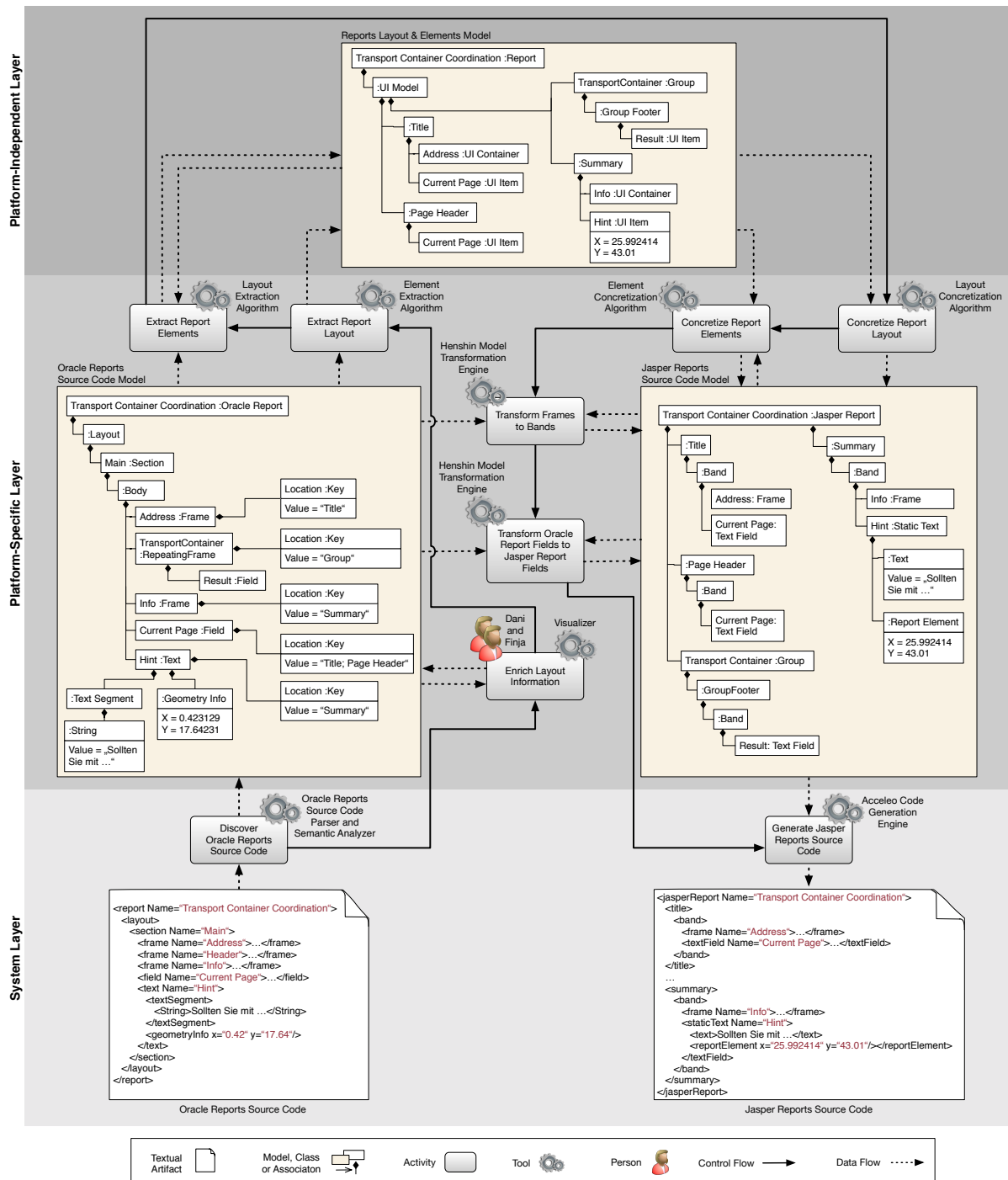


Figure 7.14 Enacting a transformation method to transform the *Report Layout* and *Report Element* concepts of the legacy report

An excerpt of the arising models and entities when converting the *Report Layout* and *Report Element* concepts can be seen in Figure 7.14. Note that the figure shows the enactment of the transformation method that has been specified in Figure 7.13.

First, the source code of the legacy report was parsed and a semantic analysis was performed. The resulting L-PSM can be seen in the middle left of Figure 7.14. In this example, the ASG of the report is shown. *Report Elements* are represented by *Fields* and *Text* objects, while the *Body*, *Frame* and *Repeating Frame* objects represent the *Report Layout*. To ease the extraction of the *Report Layout* on a platform-independent level, system experts added annotations to the objects contained in the *Body* that indicate to which part they belong to. The tool associated with the activity provided a corresponding view on the model repository for the experts. That view showed the objects to annotate and enabled adding the annotations.

The resulting ASG was used to extract the platform-independent representation of the report. The F-PIM that resulted is shown in the upper part of Figure 7.14. By the first activity named *Extract Report Layout*, the different parts of the page were created. Some *Frames* were placed as equally named *UI Containers* inside a part, like the *Address :Frame* which became a *UI Container* inside the *Title* part. Among other things, this was necessary whenever the *Frame* was not only used to group *Report Elements* but was also visualized, i.e., its border was drawn. Other *Frames* became a part itself, like the *Transport Container :Repeating Frame*, which became a *Group*. By the second activity named *Extract Report Elements*, the actual elements were placed inside the parts and containers. During both activities, the position of the extracted objects was recalculated to be relative to each other.

After the extraction, the report was concretized in the target environment. Thereby, the *Reports Layout & Elements Model* was used to create the overall structure of the report, while details were added by dependent language transformations on the platform-specific layer. For example, the *Hint :Static Text* object was created by performing the *Concretize Report Elements* activity, based on the platform-independent model. Thereafter, the contained *Text* object was created by performing the *Transform Oracle Report Fields to Jasper Report Fields* activity. The resulting T-PSM is shown in the right side of Figure 7.14. Finally, this model was used to generate source code in the target environment.

The automated conversion resulted in a source file consisting of nearly 650 LOC. The file could be opened in the IDE of the target environment and be executed. As required reimplementations activities were not performed (cf. Section 7.3), some functionalities were missing. For example, as the functionality which initializes the context of a report was not transformed (cf. Section 7.2.2), a field which visualizes the current date was empty. Nevertheless, it could be seen that the overall structure of the report had been successfully adapted to the target environment.

7.4 Discussion

In this section, we discuss and answer the evaluation criteria introduced in Section 7.1 based on our experiences made as part of the feasibility studies.

EQ1: Does the content of the MEFiSTo method base enable to develop situation-specific software transformation methods for the modernization of legacy systems?

We consider a transformation method to be situation-specific if it is efficient and effective (cf. Section 6.3.2). Effectiveness relates to properties of the resulting system, while efficiency relates to properties of the enacted process.

To determine the effectiveness of the method, we need to evaluate whether we were able to transform the legacy system as intended. In both feasibility studies, we intended to modernize the systems, i.e., to adapt them to the new environment. To assess whether we were able to modernize the systems, we (hypothetically) compared them to systems that were natively developed in the target environment and searched for differences.

We performed this examination together with the modernization expert of the project and concluded that both systems were, for the most parts, adapted to the new environment. This was mainly enabled by performing the transformation on a higher level of abstraction, i.e., using models on a platform-independent layer. In fact, it was also acknowledged by the Oracle community that a model-driven approach seems to be particularly suitable for the transformation of Oracle Forms-based systems to Oracle ADF [Ora10] [KGB14].

Nevertheless, the adaptation was still limited. For example, related to the Forms/ADF feasibility study, the transformed system enables a user-based navigation between different *Dialogs*, like this was the case in the legacy system. When performing a business task, a user needs to choose which *Dialogs* to use in order to fulfill the task. Here, the underlying business process is not represented in the system. When developing a native ADF-based system, one would identify such processes and model the corresponding navigation flows. This would require performing an abstraction on a *Computation-Independent Layer* (cf. Section 2.1.2), which is currently not supported by the method base. Extending the method base to that abstraction level would increase the degree of modernization possible.

To determine the efficiency of the method in terms of effort required, we would need to quantify and compare it with other methods. To determine whether it is the most efficient one, we would need to compare it with all possible transformation methods that could have been used to transform the legacy systems of the studies into the same resulting systems. A meaningful quantification to this extent was not performed as part of this thesis. Also, the legacy systems we transformed were not that large. As automated conversions can only be

efficient when the legacy system is sufficiently large [Fle+07], it probably would have required less effort in terms of development time to completely reimplement both systems.

Nevertheless, based on discussions with the modernization expert of the studies, we still conclude that the transformation methods developed were efficient. More specifically, they would have been efficient if the legacy system would have been larger, i.e., if the systems would have contained more modules or more reports, respectively. This assessment is primarily based on the observation that the method base provided a high degree of freedom when choosing the way of how to transform a concept. Since different transformation strategies were supported, one was not limited to a specific strategy, e.g., to solely performing an automated conversion. Instead, manual or semi-automatic activities could be included whenever deemed appropriate by the experts that were responsible for developing the method. Of course, this shifts the responsibility to the experts. If they perform wrong estimations, it will have a direct influence on the efficiency of the resulting method.

In case that wrong estimations led to an inefficient method, the MEFiSTo framework at least enables recognizing the mistake made. This is due to the fact that the rationale of the decision points that led to the developed method are made explicit. This enables assessing whether the estimations performed were correct after the transformation was performed. We assume that this is valuable knowledge which can be reused in subsequent projects during the development of a method to increase the probability of developing an efficient method.

EQ2: Does the method engineering process of MEFiSTo guide the development of situation-specific transformation method specifications?

Based on our experiences with constructing the transformation method for both feasibility studies, we conclude that the method engineering process provides useful guidance in the development of a transformation method. This conclusion is based on two observations.

First, we observed that the notation of *Concepts* and *Method Patterns* has proved to be useful in order to systematically develop a transformation method. After having intensive discussions during the development of the transformation methods, we came to the conclusion that these notations provide a helpful terminology, i.e., vocabulary, to discuss the most relevant aspects of a transformation method: they enable to discuss *what to transform* (concept) and *how* (method pattern). Thereby, the notations provide a sufficient degree of abstraction. In a discussion, method patterns can be used to refer to an idea of how to perform the transformation without discussion technical details, i.e., single activities. In contrast, concepts can be used to refer to some functionality of the system to transform without discussing single lines of code.

Second, the defined method engineering process has proved to cover the relevant activities to develop a transformation method. By enacting the activities as defined by the MEFiSTo

framework, we were able to systematically develop the methods for both feasibility studies. Also, the descriptions given of the artifacts to generate provided useful guidance.

Nevertheless, getting familiar with the framework took some time. For example, representing a software system by a set of concepts is a challenging task, especially, if it is performed for the first time. This was particularly noticeable during the first study, where we revised the concept model multiple times. In contrast, when performing the second study, we were already familiar with the MEFiSTo framework. As a result, performing the activities and creating prescribed artifacts required significantly less effort.

EQ3: Can a transformation method that had been developed by applying the MEFiSTo framework be enacted to transform a real-world legacy system?

The legacy systems of both feasibility studies were real-world systems and still in use, i.e., we did not use artificial systems. Based on our experiences with transforming these systems, we conclude that the MEFiSTo framework is suitable to transform real-world legacy systems.

Transforming the real-estate legacy system resulted in a modernized system that could be used productively. Problems that arose during the transformation were made explicit by the feedback exchanged (cf. Section 7.2.4). It could be seen that most problems were related to bugs in the tools or the reimplementation guidance specifications. While the overall transformation was successful, it is an indication that the *Tool Implementation* activity can still be improved, for example, by providing better guidance. In the current state of the framework, the required tools are specified informally in the transformation method specification. A formal specification may enable to derive parts of the tools automatically and can also enable their validation. However, formally specifying the required tools is left open for future work.

For the transformation of the report, we did not perform any reimplementation activities wherefore we did not gather any feedback. However, based on the results of the automatic conversion, we concluded that the transformation was successful (cf. Section 7.3.4).

As a limitation, we are aware that the systems itself were not that large. Therefore, the question arises whether the framework can be applied on systems that consist of millions of lines of code. Such instances can lead to new challenges, e.g., concept models becoming quite large and complex. Evaluating the scalability of MEFiSTo is left open for future work.

EQ4: Does the MEFiSTo framework support different environmental changes?

An environmental change is the change that a legacy system experiences during its transformation. Based on our experiences in applying the MEFiSTo framework in both feasibility studies, we conclude that the framework supports different environmental changes.

The changes that took place in the studies need to be described from different perspectives. On the one hand, one needs to consider *which parts* of the legacy system change, e.g., the user

interface or the data part [SWH10, pp.38-44]. In both feasibility studies, all parts of the legacy systems were affected, i.e., we did not restrict the change to a specific part.

On the other hand, one needs to consider *in which way* the legacy system is affected, i.e., on which abstraction level the change occurs. For example, the programming language can be changed and/or the software architecture. In both feasibility studies, we observed changes on various levels of abstraction, e.g., according to the layers introduced in Section 5.3. On the *System Layer*, the runtime environment and the associated programming language changed in both feasibility studies. In addition, the environments differ in terms of the technologies used. On the *Platform-Specific Layer*, the realization of various concepts differed significantly. In the first feasibility study, an architectural change on the *Platform-Independent Layer* was necessary. However, all those changes could be addressed by the MEFiSTo framework.

Nevertheless, the supported environmental change is still limited. For example, as discussed for first evaluation criteria, the *Computation-Independent Layer* is currently not supported by the method base. Therefore, changes on that layer could not be addressed.

7.5 Summary

In this chapter, we described two feasibility studies in which we applied the MEFiSTo framework. We transformed two legacy systems that have been developed in different technologies.

First, we revisited the evaluation criteria whose fulfillment discussed by the feasibility studies in Section 7.1. We addressed different parts of the framework by the evaluation criteria, namely the method base, the method development process and the method enactment process. In addition, we aimed to investigate whether different environmental changes can be addressed.

We described the two feasibility studies in Sections 7.2 and 7.3. For both studies, the actual application of the MEFiSTo framework was described. We introduced the main artifacts and findings of each activity of the method engineering process. In addition, the transformation of a selected set of functionalities was described in detail.

The evaluation criteria have been discussed in Section 7.4, based on the experiences made. In summary, we concluded that the MEFiSTo framework enables the systematic development and enactment of situation-specific transformation methods to modernize software systems. Thereby, its application is not limited to a specific environmental change.

Conclusion and Future Work

In the previous chapters, the MEFiSTo framework has been defined and its application in practice has been demonstrated. In this chapter, we conclude the main findings of this thesis and discuss future work. In Section 8.1, we describe the main contributions of this thesis. In Section 8.2, we discuss in which way the MEFiSTo framework fulfills the requirements that have been identified in Section 3.2. Finally, we discuss future work in Section 8.3.

8.1 Contributions

The development of situation-specific transformation methods is a critical but essential part of each software modernization project. Although various approaches to support this task were proposed over time, most do either not provide a sufficient degree of flexibility when developing a method or fall short in controlling, i.e., guiding, the endeavor. We observed situations in practice in which the lack of support for the development of transformation methods hindered a tool-supported transformation of legacy systems but led to a redevelopment instead.

In this thesis, we addressed this problem by defining a situational method engineering framework called MEFiSTo that supports the development of model-driven transformation methods. The framework provides a high degree of flexibility as methods are developed by assembling predefined method building blocks. This enables a precise adaptation of a method to the situation at hand. The associated method engineering process provides comprehensive guidance for this endeavor. Subsequently, we discuss the main contributions of this thesis.

C1: A method base for model-driven transformation methods

We introduced a method base that represents a repository for reusable building blocks of transformation methods. The method base has two main constituents, namely a set of method fragments and method patterns.

Method fragments constitute atomic building blocks of methods, i.e., single activities, artifacts, tools or roles. The fragments that we proposed enable to specify the actual transformation of a legacy system as well as preliminary measures like the development of required tools. They are based on principles from the field of model-driven engineering and aligned with standards that have been proposed by the Architecture-Driven Modernization (ADM) initiative of the Object Management Group (OMG). The fragments enable to express various transformation strategies, like an automated conversion using tools, or a manual reimplementation.

To guide the use of fragments, we additionally proposed a set of method patterns. Each pattern represents a transformation strategy and encodes construction guidelines for a method that follows the strategy, i.e., it describes which method fragments to use. Thereby, configuration points are foreseen to enable a fine-granular adaptation of each pattern to the situation at hand. To guide the use of patterns, we discussed their characteristics, like the strategy followed by the pattern, or the situation in which it is applicable. In addition, we gave a detailed example for each pattern, based on situations that we observed in practice

C2: A method engineering process for situation-specific transformation methods

We introduced a method engineering process that provides guidance for the modular construction of situation-specific transformation methods. We described the purpose and emphases of each activity within the construction process and provided detailed examples. We discussed quality characteristics of artifacts that arise during the construction process and defined an intermediate modeling language to formally describe them. We introduced a mapping of the intermediate language to the Software and Systems Process Engineering Metamodel (SPEM), which is a standard proposed by the OMG to formally describe methods.

The method engineering process has been designed to support software modernization scenarios. For this purpose, we centered the process around the technique of concept modeling which we transferred to the domain of software transformation. The general idea is to represent the software system to transform as a set of concepts. A concept does not conform to a syntactical entity within the system but represents (parts of) its functionality on a higher level of abstraction. Therefore, the process enables focusing on transforming the actual functionality of a legacy system but not necessarily preserving the way in which that functionality is realized technically. Instead, it enables technically adapting the functionality to the new environment, i.e., it enables modernization.

The construction of a situation-specific transformation method is essentially based on decisions performed by experts. For example, experts decide on the method pattern to apply in order to transform some functionality, based on an assessment of the situation. To make informed and traceable decisions, the method engineering process includes principles from the

domain of architectural design decisions. In particular, it prescribes to perform a systematic exploration of the situational context and enables documenting influence factors on the decision points as well as the rationale for a decision. We assume that preserving such information is essential in the long term in order to establish a knowledge base that provides additional guidance in the development of a transformation method.

C3: Application of the solution concept in practice

We applied the MEFiSTo framework in an industrial context in order to evaluate its feasibility in practice. In particular, we performed two feasibility studies and discussed their results.

In the first feasibility study, we transformed a legacy system from the domain of real estates to a new environment. The system had been developed in the platform Oracle Forms and was still in use. We transformed it to the more recent platform Oracle ADF. By this study, we were able to demonstrate that the MEFiSTo framework can be used to modernize real-world systems in an actual project context. The transformation lasted several weeks whereby multiple project team members were involved.

In the second feasibility study, we transformed a legacy system from the logistics domain into a new environment. The system had been developed in the platform Oracle Reports and was transformed to the platform Jasper Reports. By this study, we were able to demonstrate that the MEFiSTo framework supports different environmental changes.

8.2 Requirements Revisited

In Section 3.2 we stated a set of requirements that a method engineering approach needs to fulfill in order to enable the definition of transformation methods in the context of a modernization scenario (cf. Section 3.1). Subsequently, we describe how the MEFiSTo framework fulfills these requirements.

Flexibility The requirement for *flexibility* claims that a method engineering approach shall provide a high degree of freedom in the development of a method. The more flexibility is provided, the better a method can be adapted to the situation at hand.

The MEFiSTo framework provides a high degree of flexibility since transformation methods are developed in a modular way by assembling method fragments. The fragments are stored in the associated method base and enable to express various transformation strategies as they can be flexibly combined.

Control The requirement for *control* claims that a method engineering approach shall provide a high degree of guidance in the development of a method. The more control is provided, the better the result of the development can be assured, e.g., the quality of the method.

The MEFiSTo framework provides a high degree of control for two reasons. On the one hand, the method patterns stored in the method base provide guidance in the development of a transformation method. They encode construction guidelines for a method and are associated with characteristics to determine their suitability in a given situation.

On the other hand, the proposed method engineering process defines and exemplifies all essential activities to develop a transformation method. Arising artifacts can be specified formally using a proposed intermediate modeling language. In addition, quality characteristics of the artifacts have been discussed to ensure quality characteristics of the resulting method.

Generality The requirement for *generality* claims that a method engineering approach shall not be limited to a specific environmental change. To fulfill this requirement, a method engineering approach cannot assume that the legacy system was developed in a specific technology or employs a specific software architecture.

The MEFiSTo framework is not limited to a specific environmental change as generic method fragments are stored in the method base. The fragments are only based on principles from the domain of model-driven engineering but are not related to a specific technology. The customization of these fragments to specifics of an environmental change is considered as part of the method engineering process.

Granularity The requirement for *granularity* claims that a method engineering approach shall enable to adapt the granularity of the resulting transformation method specification as deemed appropriate. To fulfill this requirement, a method engineering approach needs to enable the development of a fine-granular method specification as well as a coarse-granular one.

The MEFiSTo framework enables adapting the granularity of a specification during its development due to the use of concept modeling. In particular, the general idea to develop methods is to apply method patterns on self-defined concepts. Therefore, the granularity of the resulting specification directly depends on the amount of concepts defined. The more concepts are used, the more fine-granular the specification will become. In addition, methods are formalized by using SPEM which supports refinement, retrospectively.

Versatility The requirement for *versatility* claims that a method engineering approach shall support the definition of different transformation strategies. The more transformation strategies are provided, the better a method can be adapted to the situation at hand.

The MEFiSTo framework enables specifying different transformation strategies due to the method fragments stored in the method base. The fragments enable to express conversion- and reimplementation-based strategies as well as combinations of them. Each strategy is encoded by a method pattern.

Continuity The requirement for *continuity* claims that the use of tools during the enactment of a developed transformation method shall be guided. To fulfill this requirement, a method engineering approach can either provide guidance in the development of custom tools or provide predefined ones and guide their usage.

The MEFiSTo framework guides the use of tools in two ways. On the one hand, a generic set of required capabilities that a tool infrastructure needs to provide is described. These capabilities are required to enable the enactment of transformation methods that have been developed by using the MEFiSTo framework. On the other hand, it is foreseen to specify the development of custom tools as part of the transformation method specification.

Formalization The requirement for *formalization* claims that a transformation method shall be specified formally. A formal specification enables a precise description of the method which is essential, e.g., to reuse the method in subsequent modernization projects.

Each transformation method specification that has been developed by the MEFiSTo framework is specified formally. In particular, the specification is a model that conforms to a metamodel. First the MEFiSTo Intermediate Modeling Language (MIML) is used during the development of the method. When the development of the method is completed, the specification is transformed into the Software and Systems Process Engineering Metamodel (SPEM) standard that was proposed by the OMG.

As can be seen based on the descriptions given, the MEFiSTo framework fulfills the requirements stated in Section 3.2. Due to this, as well as due to the feasibility studies performed, we conclude that the framework enables the development and enactment of transformation methods in the context of a modernization scenario.

8.3 Future Work

In this section, we describe future work that could be performed in the area of developing situation-specific transformation methods for software modernizations scenarios. First, we describe possible enhancements of the MEFiSTo framework, before sketching an idea for the automated learning of transformation methods.

8.3.1 Enhancing the MEFiSTo framework

In this section, we describe future work that could be performed to further enhance the MEFiSTo framework. We structure the possible enhancements of the MEFiSTo framework according to the focus areas of the framework, namely the *method base*, the *method development* and *method enactment processes* as well as the *evaluation* of the framework.

Method Base The content of the method base is essential for the flexibility of the MEFiSTo framework in the development of transformation methods. We were able to show that the method base provided sufficient flexibility to modernize the legacy systems of both feasibility studies. However, in its current state, the highest level of abstraction used is the *Platform-Independent Layer*. The feasibility studies revealed that an abstraction on a *Computation-Independent Layer* can enable to even better adapt a software system to a target environment (cf. Section 7.4). Therefore, including method fragments and patterns for this level of abstraction could enhance the framework.

Method Development During the method development phase of the framework, the transformation method specification gets systematically defined. In this context, the MEFiSTo framework could be enhanced in three areas.

First, the intermediate language called MIML could be enhanced to enable the specification of more detailed transformation methods. In its current state, MIML does not support the specification of an artifact's lifecycle (cf. Section 6.4.1). As a result, the lifecycle is only described implicitly by specifying the order between activities that manipulate an artifact. Another shortcoming of MIML is related to the specification of concept models. In its current state, it is only possible to specify concepts but not their instances (cf. Section 6.3.1). For example, it could be possible to specify the amount of concept instances that reside within a legacy system, or even the amount of lines of code related to a concept instance. We assume that this would be a valuable information for a modernization expert in the selection of a method pattern to develop a situation-specific method.

Second, the introduced quality characteristics and integration operations could be formalized to enable tool support. In its current state, we introduced various quality characteristics related to the artifacts arising during the development of a transformation method (cf. Section 6.4). For example, concept models shall be free of cycles. Formalizing such characteristics would enable to provide tool support in order to evaluate them automatically. The same is true for the introduced operations to integrate different artifacts (cf. Section 6.4.2). A formalization of these operations would enable to provide tool support, too.

Third, tool support in general could be enhanced to ease the development of transformation methods. On the one hand, the tooling itself could be made more mature, i.e., it could be optimized for the use in an industrial context. For example, the editor we provided to manipulate transformation method specifications does not perform well when dealing with large specifications. On the other hand, additional capabilities could be provided, like the already mentioned evaluation of quality characteristics.

Method Enactment During the method enactment phase of the framework, tools are developed that are required for automating (parts of) the transformation. Thereafter, the method is performed as prescribed by the transformation method specification. In this context, the MEFiSTo framework could be enhanced in both areas.

First, the actual implementation of tools could be enhanced by defining a closer integration between the transformation method specification and the tool infrastructure. In its current state, the specification prescribes which tools need to be developed by tool developers, based on a tool infrastructure. The infrastructure provides some generic capabilities, like means to develop metamodels or model transformations. This infrastructure could benefit from ongoing research which aims to identify generic reengineering services (cf. Section 3.3.2). Intuitively, the idea is to derive project-specific tools by parameterizing those services. When including such services in the tool infrastructure of MEFiSTo, we assume that the transformation method specification could be used to parameterize the services. However, this may require specifying some parts of the transformation method more technically.

Second, the actual transformation could be enhanced by providing tool support that guides software developers in reimplementation activities. In particular, a process engine could be included that assigns activities to a developer, based on the developed method specification. Thereby, the engine should also be integrated with the model repository that originates during the transformation. In this case, it could provide adequate views on the repository if deemed necessary for a reimplementation task.

Evaluation By the evaluation performed in this thesis, we were able to demonstrate that it is feasible to apply the MEFiSTo framework in practice to modernize legacy systems (cf. Section 7.4). However, evaluating other characteristics of the framework that are relevant for its use in practice is an open task. For example, it needs to be determined whether the framework could be used to transform large-scale legacy systems that consist of millions of lines of code.

8.3.2 Example-Based Method Learning

The MEFiSTo framework enables the development of situation-specific transformation methods by following a construction-based approach. Thereby, a method is developed in advance of the actual transformation by experts, who decide on the transformation strategies to apply. Therefore, mistakes of the experts involved, e.g., in terms of wrong assumptions, directly have a negative influence on the resulting method.

To exclude mistakes during the development which can result due to humans involved, another approach could follow the direction to retrospectively learn the transformation method by using examples. In particular, we could envision that a set of pairs is provided consisting of legacy source code and the transformed equivalent in the new environment. Based on these sets, an approach could automatically learn the underlying transformation method and apply it on additional legacy source code.

We assume that the feasibility of such an approach is dependent on advancements in current research areas. For example, techniques from the area of *model transformations by example* [Var06] could enable an automatic derivation of model transformations between exemplary models on the same level of abstraction. In addition, techniques from the area of *program comprehension* could enable to automatically perform abstractions, which is essential in the area of software modernization.

References

- [Aho+06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.
- [AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A pattern language: towns, buildings, construction*. Oxford University Press, 1977.
- [AK01] Colin Atkinson and Thomas Kühne. “Processes and products in a multi-level metamodeling architecture”. In: *International Journal of Software Engineering and Knowledge Engineering* 11.06 (2001), pp. 761–783.
- [AKB02] Mehmet Aksit, Ivan Kurtev, and Jean Bézivin. “Technological Spaces: an initial appraisal”. In: *Proceedings of the 4th International Symposium on Distributed Objects and Applications (DOA)* (2002). <http://doc.utwente.nl/55814/> (accessed March 22th, 2016).
- [Bar+10] Franck Barbier, Parastoo Mohagheghi, A. J. Berre, Andrey Sadovykh, and Gorka Benguria. “Reuse and Migration of Legacy Systems to Interoperable Cloud Services - The REMICS project”. In: *4th Workshop on Modeling, Design, and Analysis for the Service Cloud*. 2010.
- [BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.
- [BG01] Jean Bezivin and Olivier Gerbe. “Towards a precise definition of the OMG/MDA framework”. In: *Proceedings of the 16th International Conference on Automated Software Engineering (ASE)*. 2001, pp. 273–280.
- [Bis+99] Jesus Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. “Legacy Information Systems: Issues and Directions.” In: *IEEE Software* 16.5 (1999), pp. 103–111.
- [BR15] Franck Barbier and Jean-Luc Reoussine. *COBOL Software Modernization*. John Wiley & Sons, 2015.
- [Bri96] Sjaak Brinkkemper. “Method engineering: engineering of information systems development methods and tools.” In: *Information and Software Technology* 38.4 (1996), pp. 275–280.

- [Bro83] Ruven E. Brooks. “Towards a Theory of the Comprehension of Computer Programs”. In: *International Journal of Man-machine Studies* 18 (6 1983), pp. 543–554.
- [Bru+14] Hugo Brunelière, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. “MoDisco: a Model Driven Reverse Engineering Framework”. In: *Information and Software Technology* 56.8 (2014), pp. 1012–1032.
- [BS95] Michael L. Brodie and Michael Stonebraker. *Migrating legacy systems: gateways, interfaces & the incremental approach*. Morgan Kaufmann Publishers Inc., Aug. 1995.
- [Bus+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., 1996.
- [Byr92] Eric J. Byrne. “A conceptual foundation for software re-engineering”. In: *Proceedings of the 5th International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 1992, pp. 226–235.
- [Cab+15] Jordi Cabot, Javier Luis Canovas Izquierdo, Leire Orue-Echevarria Arrieta, Olivier Strauss, Manuel Wimmer, and Hugo Bruneliere. “Software Modernization Revisited: Challenges and Prospects”. In: *Computer* 48.8 (Aug. 2015), pp. 76–80.
- [CC90] Elliot J. Chikofsky and James H. II Cross. “Reverse engineering and design recovery: a taxonomy”. In: *IEEE Software* 7.1 (1990), pp. 13–17.
- [Cos+12] Valerio Cosentino, Jordi Cabot, Patrick Albert, Philippe Bauquel, and Jacques Perronnet. “A Model Driven Reverse Engineering Framework for Extracting Business Rules Out of a Java Application”. In: *Rules on the Web: Research and Applications*. Vol. 7438. Lecture Notes in Computer Science. Springer, 2012, pp. 17–31.
- [Dat03] Christopher J. Date. *An Introduction to Database Systems (8th Edition)*. Addison-Wesley, 2003, p. 1024.
- [Der+14] Mahdi Derakhshanmanesh, Jürgen Ebert, Thomas Iguchi, and Gregor Engels. “Model-Integrating Software Components”. In: *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Valencia, Spain: Springer, 2014.

- [Der+16] Mahdi Derakhshanmanesh, Marvin Grieger, Jürgen Ebert, and Gregor Engels. “Thoughts on the Evolution Towards Model-Integrating Software (to appear)”. In: *Softwaretechnik-Trends* (2016). Proceedings of the 3rd Workshop Model-Based and Model-Driven Software Modernization (MMSM).
- [Des00] Philippe Desfray. “UML Profiles versus Metamodel extensions: An ongoing debate”. In: *OMG’s UML Workshops: UML in the .com Enterprise: Modeling CORBA, Components, XML/XMI and Metadata Workshop* (2000), pp. 6–9.
- [DF94] Mark Dowson and Christer Fernström. “Towards requirements for enactment mechanisms”. In: *Software Process Technology*. Springer, 1994, pp. 90–106.
- [DG15] Mahdi Derakhshanmanesh and Marvin Grieger. “On Enabling Technologies for Longevity in Software”. In: *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015*. Vol. 1337. CEUR Workshop Proceedings, 2015, pp. 112–114.
- [DLGB12] Gaëtan Deltombe, Olivier Le Goaer, and Franck Barbier. “Bridging KDM and ASTM for Model-Driven Software Modernization.” In: *Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering (SEKE)* (2012), pp. 517–524.
- [Ebe+02] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. “Gupro - Generic Understanding of Programs. An Overview”. In: *Electronic Notes in Theoretical Computer Science* 72.2 (2002), pp. 47–56.
- [ES10] Gregor Engels and Stefan Sauer. “A Meta-Method for Defining Software Engineering Methods”. In: *Graph Transformations and Model-Driven Engineering*. Vol. 5765. Lecture Notes in Computer Science. 2010, pp. 411–440.
- [FBGS15] Masud Fazal-Baqaie, Marvin Grieger, and Stefan Sauer. “Tickets without Fine - Artifact-based Synchronization of Globally Distributed Software Development in Practice”. In: *Proceedings of the 16th International Conference of Product Focused Software Development and Process Improvement (PROFES)*. Vol. 9459. Lecture Notes in Computer Science. Springer, 2015, pp. 167–181.
- [FBLE13] Masud Fazal-Baqaie, Markus Luckey, and Gregor Engels. “Assembly-Based Method Engineering with Method Patterns.” In: *Software Engineering Workshopband* (2013), pp. 435–444.
- [FCL09] Charles N. Fischer, Ron K. Cytron, and Richard J. LeBlanc. *Crafting a compiler*. Addison-Wesley, 2009.

- [Fle+07] Franck. Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jeam-Marc Jézéquel. “Model-Driven Engineering for Software Migration in a Large Industrial Context”. In: *Proceedings of the 10th international conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 2007, pp. 482–497.
- [Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.
- [FRE14] Andreas Fuhr, Volker Riediger, and Jürgen Ebert. “Towards Generic Services for Software Reengineering”. In: *Softwaretechnik-Trends* 34.2 (2014). Proceedings of the 16th Workshop Software-Reengineering and Evolution (WSRE).
- [Fuh+12] Andreas Fuhr, Tassilo Horn, Volker Riediger, and Andreas Winter. “Model-Driven Software-Migration - Process Model, Tool Support and Application”. In: *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. IGI Global, 2012, pp. 153–184.
- [Gam+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.
- [GFB15] Marvin Grieger and Masud Fazal-Baqaie. “Towards a Framework for the Modular Construction of Situation-Specific Software Transformation Methods”. In: *Softwaretechnik-Trends* 35.2 (2015). Proceedings of the 17th Workshop Software-Reengineering and Evolution (WSRE), pp. 41–42.
- [GFBS16] Marvin Grieger, Masud Fazal-Baqaie, and Stefan Sauer. “A Method-Base for the Situation-Specific Development of Model-Driven Transformation Methods (to appear)”. In: *Softwaretechnik-Trends* (2016). Proceedings of the 3rd Workshop Model-Based and Model-Driven Software Modernization (MMSM).
- [GGS12] Marvin Grieger, Baris Güldali, and Stefan Sauer. “Sichern der Zukunftsfähigkeit bei der Migration von Legacy-Systemen durch modellgetriebene Softwareentwicklung”. In: *Softwaretechnik-Trends* 32.2 (2012). Proceedings of the 14th Workshop Software-Reengineering (WSR), pp. 37–38.
- [Gri+14] Marvin Grieger, Masud Fazal-Baqaie, Stefan Sauer, and Markus Klenke. “A Method to Systematically Improve the Effectiveness and Efficiency of the Semi-Automatic Migration of Legacy Systems”. In: *Softwaretechnik-Trends* 34.2 (2014). Proceedings of the 16th Workshop Software-Reengineering and Evolution (WSRE), pp. 77–78.

- [Gri+16] Marvin Grieger, Masud Fazal-Baqaie, Gregor Engels, and Markus Klenke. “Concept-Based Engineering of Situation-Specific Migration Methods”. In: *Proceedings of the 15th International Conference on Software Reuse (ICSR)*. Vol. 9679. Lecture Notes in Computer Science. Springer, 2016, pp. 199–214.
- [GS13] Marvin Grieger and Stefan Sauer. “Wiederverwendbarkeit von Migrationswissen durch Techniken der modellgetriebenen Softwareentwicklung”. In: *Software Engineering 2013 Workshopband*. Köllen Druck+Verlag GmbH, 2013, pp. 189–200.
- [GS94] David Garlan and Mary Shaw. *An Introduction to Software Architecture*. Technical Report. Carnegie Mellon University. 1994.
- [GSK14] Marvin Grieger, Stefan Sauer, and Markus Klenke. “Architectural Restructuring by Semi-Automatic Clustering to Facilitate Migration towards a Service-oriented Architecture”. In: *Softwaretechnik-Trends* 34.2 (2014). Proceedings of the 2nd Workshop Model-Based and Model-Driven Software Modernization (MMSM), pp. 44–45.
- [GW05] Rainer Gimnich and Andreas Winter. “Workflows der Software-Migration”. In: *Softwaretechnik-Trends* 2.25 (2005). Proceedings of the 16th Workshop Software-Reengineering (WSR), pp. 22–24.
- [Har97] Anton Frank Harmsen. “Situational Method Engineering”. PhD thesis. University of Twente, 1997.
- [HBO94] Frank Harmsen, Sjaak Brinkkemper, and Han Oei. “Situational method engineering for informational system project approaches.” In: *Methods and Associated Tools for the Information Systems Life Cycle* (1994), pp. 169–194.
- [Hec+08] Reiko Heckel, Rui Correia, Carlos Matos, Mohammad El-Ramly, Georgios Koutsoukos, and Luis Andrade. “Architectural Transformations: From Legacy to Three-tier and Services”. In: *Software Evolution*. Springer, 2008, pp. 139–170.
- [Her03] Jack Herrington. *Code Generation in Action*. 2003.
- [HS+14] Brian Henderson-Sellers, Jolita Ralyté, Pär J Ågerfalk, and Matti Rossi. *Situational Method Engineering*. Springer, 2014.
- [HSGP11] Brian Henderson-Sellers and Cesar Gonzalez-Perez. “Towards the Use of Granularity Theory for Determining the Size of Atomic Method Fragments for Use in Situational Method Engineering”. In: *Engineering Methods in the Service-Oriented Context*. Vol. 351. IFIP Advances in Information and Communication Technology. Springer, 2011, pp. 49–63.

- [HSGPR08] Brian Henderson-Sellers, Cesar Gonzalez-Perez, and Jolita Ralyté. “Comparison of method chunks and method fragments for situational method engineering”. In: *Proceedings of the 19th Australian Software Engineering Conference (ASWEC)* (2008), pp. 479–488.
- [HV97] Arthur H.M. ter Hofstede and T.F. Verhoef. “On the feasibility of situational method engineering”. In: *Information Systems* 22.6-7 (1997), pp. 401–422.
- [IEEE00] Institute of Electrical and Electronics Engineers. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. <https://standards.ieee.org/findstds/standard/1471-2000.html> (accessed March 22th, 2016). 2000.
- [Jel15] Jan Jelschen. “Service-oriented toolchains for software evolution”. In: *Proceedings of the 9th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA)*. IEEE. 2015, pp. 51–58.
- [JGY16] Ivan Jovanovikj, Marvin Grieger, and Enes Yigitbas. “Reusing Test Cases in Software Migration Projects: Challenges & Opportunities (to appear)”. In: *Softwaretechnik-Trends* (2016). Proceedings of the 18th Workshop Software-Reengineering and Evolution (WSRE).
- [Jov+16] Ivan Jovanovikj, Marvin Grieger, Baris Güldali, and Alexander Teetz. “Reengineering of Legacy Test Cases: Problem Domain & Scenario (to appear)”. In: *Softwaretechnik-Trends* (2016). Proceedings of the 3rd Workshop Model-Based and Model-Driven Software Modernization (MMSM).
- [KGB14] Markus Klenke, Marvin Grieger, and Wolf Gerhard Beckmann. “Forms2ADF mal anders: Wie aus einer Oracle-Vision Praxis wird”. In: *DOAG News* 3 (June 2014), pp. 38–42.
- [KGW98] Rainer Koschke, Jean-François Girard, and Martin Wirthner. “An intermediate representation for integrating reverse engineering analyses”. In: *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE)* (1998), pp. 241–250.
- [Kha+11] Ravi Khadka, Gijs Reijnders, Amir Saeidi, Slinger Jansen, and Jurriaan Hage. “A method engineering based legacy to SOA migration method”. In: *Proceedings of the 27th International Conference on Software Maintenance (ICSM)*. IEEE. 2011, pp. 163–172.

- [KLV06] Philippe Kruchten, Patricia Lago, and Hans van Vliet. “Building Up and Reasoning About Architectural Knowledge”. In: *Proceedings of the 2nd International Conference on Quality of Software Architectures (QoSA)*. Springer, 2006, pp. 43–58.
- [KNE92] Wojtek Kozaczynski, Jim Ning, and Andre Engberts. “Program Concept Recognition and Transformation”. In: *IEEE Transactions on Software Engineering*. 1992, pp. 1065–1075.
- [KR91] Jürgen Koenemann and Scott P. Robertson. “Expert Problem Solving Strategies for Program Comprehension”. In: *Proceedings of the 9th Conference on Human Factors in Computing Systems (CHI)*. ACM, 1991, pp. 125–130.
- [Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction*. 3rd ed. Addison-Wesley, 2003.
- [KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.
- [KWC98] Rick Kazman, Steven G. Woods, and S. Jeromy Carrière. “Requirements for Integrating Software Architecture and Reengineering Models: CORUM II”. In: *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 1998, pp. 154–163.
- [Luk+] Martin Lukasiewicz, Michael Glaß, Felix Reimann, and Jürgen Teich. “Opt4J - A Modular Framework for Meta-heuristic Optimization”. In: *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO)*, pp. 1723–1730.
- [MD08] Tom Mens and Serge Demeyer. *Software Evolution*. Springer, 2008.
- [Men+14] Andreas Menychtas, Kleopatra Konstanteli, Juncal Alonso, Leire Orue-Echevarria, Jesus Gorronogoitia, George Kousiouris, Christina Santzaridou, Hugo Bruneliere, Bram Pellens, Peter Stuer, et al. “Software modernization and cloudification using the ARTIST migration methodology and framework”. In: *Scalable Computing: Practice and Experience* 15.2 (2014).
- [MH11] Carlos Matos and Reiko Heckel. “Legacy transformations for extracting service components”. In: *Rigorous software engineering for service-oriented systems*. Springer, 2011, pp. 604–621.
- [MV93] A. von Mayrhauser and A.M. Vans. “From program comprehension to tool requirements for an industrial environment”. In: *Proceedings of the 2nd Workshop on Program Comprehension*. July 1993, pp. 78–86.

- [Nie+02] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, and J. Welsh. “Towards Pattern-Based Design Recovery”. In: *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. 2002, pp. 338–348.
- [OMG08a] Object Management Group. *Software & Systems Process Engineering Meta-model Specification (SPEM), Version 2.0*. <http://www.omg.org/spec/SPEM/2.0/> (accessed March 22th, 2016). 2008.
- [OMG08b] Object Management Group. *MOF Model to Text Transformation Language, Version 1.0*. <http://www.omg.org/spec/MOFM2T/1.0/> (accessed March 22th, 2016). 2008.
- [OMG11a] Object Management Group. *Architecture-Driven Modernization: Abstract Syntax Tree Metamodel (ASTM)*. <http://www.omg.org/spec/ASTM/1.0/> (accessed March 22th, 2016). 2011.
- [OMG11b] Object Management Group. *Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM)*. <http://www.omg.org/spec/KDM/1.3> (accessed March 22th, 2016). 2011.
- [OMG14] Object Management Group. *Model Driven Architecture (MDA): MDA Guide rev. 2.0*. <http://www.omg.org/cgi-bin/doc?ormsc/14-06-01> (accessed March 22th, 2016). 2014.
- [OMG15a] Object Management Group. *Meta Object Facility (MOF), Version 2.5*. <http://www.omg.org/spec/MOF/2.5/> (accessed March 22th, 2016). 2015.
- [OMG15b] Object Management Group. *UML Infrastructure Specification, v2.5*. <http://www.omg.org/spec/UML/2.5/> (accessed March 22th, 2016). 2015.
- [Ora00a] Oracle. *Oracle Forms Developer and Oracle Reports Developer: Guidelines for Building Applications*. http://download.oracle.com/otn_hosted_doc/forms/forms/A73073_01.pdf (accessed March 22th, 2016). 2000.
- [Ora00b] Oracle. *Oracle Forms Developer: Form Builder Reference, Volume 1*. http://download.oracle.com/otn_hosted_doc/forms/forms/A73074_01.pdf (accessed March 22th, 2016). 2000.
- [Ora00c] Oracle. *Oracle Forms Developer: Form Builder Reference, Volume 2*. http://download.oracle.com/otn_hosted_doc/forms/forms/A73074_01.pdf (accessed March 22th, 2016). 2000.
- [Ora09a] Oracle. *Oracle Forms Services & Oracle Forms Developer 11g Technical Overview*. <http://www.oracle.com/technetwork/developer-tools/forms/overview/technical-overview-130127.pdf> (accessed March 22th, 2016). 2009.

- [Ora09b] Oracle. *Oracle Fusion Middleware: Oracle Reports User's Guide to Building Reports*. https://docs.oracle.com/cd/E12839_01/bi.11111/b32122.pdf (accessed March 22th, 2016). 2009.
- [Ora10] Oracle. *Oracle Forms Migration Framework (OFMF)*. <http://www.oracle.com/technetwork/de/community/forms/wp-oracle-forms-migration-framework-2193628-de.pdf> (accessed March 22th, 2016). 2010.
- [Ora12] Oracle. *Oracle application development tools statement of direction: Oracle Forms, Oracle Reports and Oracle Designer*. <http://www.oracle.com/technetwork/issue-archive/2010/toolssod-3-129969.pdf> (accessed October 7th, 2015). 2012.
- [Ora13] Oracle. *Oracle Fusion Middleware 12c 12.1.3*. <http://docs.oracle.com/middleware/1213/adf/docs.htm> (accessed March 22th, 2016). 2013.
- [Pan+14] Gaurav Pandey, Jan Jelschen, Dilshodbek Kuryazov, and Andreas Winter. "Quality Measurement Scenarios in Software Migration". In: *Softwaretechnik Trends*. Vol. 34. 2. Proceedings of the 16th Workshop Software-Reengineering and Evolution (WSRE). 2014, pp. 54–55.
- [PCDGP11] Ricardo Pérez-Castillo, Ignacio Garcia-Rodriguez De Guzman, and Mario Piatini. "Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems". In: *Computer Standards & Interfaces* 33.6 (2011), pp. 519–532.
- [PHY11] Kata Praditwong, Mark Harman, and Xin Yao. "Software module clustering as a multi-objective search problem". In: *IEEE Transactions on Software Engineering* 37.2 (2011), pp. 264–282.
- [Ral04] Jolita Ralyté. "Towards situational methods for information systems development: engineering reusable method chunks". In: *Proceedings of the 13th International Conference on Information Systems Development (ISD)*. 2004, pp. 271–282.
- [RB00] Václav T. Rajlich and Keith H. Bennett. "A Staged Model for the Software Life Cycle". In: *Computer* 33.7 (2000), pp. 66–71.
- [RDR03] Jolita Ralyté, Rébecca Deneckère, and Colette Rolland. "Towards a generic model for situational method engineering". In: *Advanced Information Systems Engineering*. Springer, 2003, pp. 95–110.

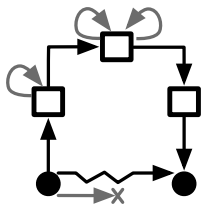
- [RE+12] Roberto Rodríguez-Echeverría, José María Conejero, Pedro J. Clemente, Juan C. Preciado, and Fernando Sánchez-Figueroa. “Modernization of Legacy Web Applications into Rich Internet Applications”. In: *Current Trends in Web Engineering*. Vol. 7059. Lecture Notes in Computer Science. Springer, 2012, pp. 236–250.
- [RL15] Maryam Razavian and Patricia Lago. “A systematic literature review on SOA migration”. In: *Journal of Software: Evolution and Process* 27.5 (2015), pp. 337–372.
- [RM11] Grant Ronald and Lynn Munsinger. *A Case Study in an Oracle Forms Redevelopment Project to Oracle ADF*. Technical Report. http://download.oracle.com/otn_hosted_doc/jdeveloper/11gdemos/SummitADF/SummitADF_Redevelopment.pdf (accessed March 22th, 2016). 2011.
- [RP96] Colette Rolland and Véronique Plihon. “Using generic method chunks to generate process models fragments”. In: *Proceedings of the 2nd International Conference on Requirements Engineering (RE)* (1996), pp. 173–180.
- [Rug95] Spencer Rugaber. *Program Comprehension*. Tech. rep. College of Computing, 1995.
- [RVP06] Aoun Raza, Gunther Vogel, and Erhard Plödereder. “Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering”. In: *Proceedings of the 11th International Conference on Reliable Software Technologies (Ada-Europe)*. Lecture Notes in Computer Science. Springer, 2006, pp. 71–82.
- [RW02] Václav Rajlich and Norman Wilde. “The Role of Concepts in Program Comprehension”. In: *Proceedings of the 10th International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society, 2002, pp. 271–278.
- [San14] Óscar Sánchez Ramón. “Model-driven modernisation of legacy graphical user interfaces”. <http://tdcat.cesca.es/handle/10803/286504> (accessed March 22th, 2016). Ph.D. thesis. 2014.
- [Sne05] H. M. Sneed. “Estimating the Costs of a Reengineering Project”. In: *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2005, pp. 111–119.
- [Som10] Ian Sommerville. *Software Engineering*. 9th. Addison-Wesley, 2010.

- [SSG14] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. “Model-driven reverse engineering of legacy graphical user interfaces”. In: *Proceedings of the 28th International Conference on Automated Software Engineering (ASE)* 21 (2 2014), pp. 147–186.
- [Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.
- [SWH10] H. M. Sneed, Ellen Wolf, and Heidi Heilmann. *Software-Migration in der Praxis: Übertragung alter Softwaresysteme in eine moderne Umgebung*. dpunkt Verlag, 2010.
- [Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. 2nd. Addison-Wesley, 2002.
- [Tib15] Tibco Analytics. *TIBCO Jaspersoft Studio User Guide, Version 6.2*. <http://community.jaspersoft.com/documentation?version=29351> (accessed March 22th, 2016). 2015.
- [TV00] Andrey A. Terekhov and Chris Verhoef. “The Realities of Language Conversions”. In: *IEEE Software* 17.6 (2000), pp. 111–124.
- [UN10] William M. Ulrich and Philip Newcomb. *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. 2010, p. 429.
- [Var06] Dániel Varró. “Model Transformation by Example”. In: *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 4199. Lecture Notes in Computer Science. Springer, 2006, pp. 410–424.
- [VMXe15] Verein zur Weiterentwicklung des V-Modell XT e.V. *V-Modell XT: Das deutsche Referenzmodell zur für Systementwicklungsprojekte, Version 2.0*. <http://www.v-modell-xt.de/> (accessed March 22th, 2016). 2015.
- [VR09] Peter Van Roy. “Programming paradigms for dummies: What every programmer should know”. In: *New computational paradigms for computer music* 104 (2009).
- [YC79] Edward Yourdon and Larry L Constantine. *Structured design: Fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc., 1979.
- [Zim09] Olaf Zimmermann. “An Architectural Decision Modeling Framework for Service-Oriented Architecture Design”. http://elib.uni-stuttgart.de/bitstream/11682/2682/1/SOAD_ArchitecturalDecisionModeling_PhDThesisOlafZimmermann.pdf (accessed March 22th, 2016). Ph.D. thesis. 2009.

APPENDIX A

Characterization of Method Patterns

Concept Recognition-Based Reimplementation (F₆)

Intent	Perform a semi-automatic transformation of the legacy system's functionality into a new environment by combining a conversion- and a reimplementation-based transformation strategy
Strategy	Use an intermediate representation of parts of the functionality to transform on a platform-independent layer to guide a manual reimplementation. The intermediate representation is reverse engineered from an ASG ¹ of the legacy system on a platform-specific layer and concretized into an ASG ¹ of the target environment. Subsequently, code is generated in the target environment that forms the basis for manual reimplementation activities performed by software developers
Structure	

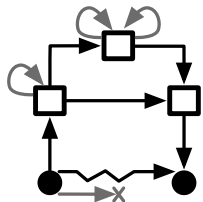
Continued on next page

Applicability	<p>Use when the functionality to transform is realized significantly different in the legacy and target environment and when software developers can be pointedly guided if parts of the functionality are made explicit. The use of an intermediate representation can reduce the complexity of a direct transformation by separating the concerns of reverse engineering, restructuring and mapping the functionality. The manifestation of these concerns essentially influences the efficiency and effectiveness of the pattern. Those parts of the functionality, which have not been converted, shall be reimplemented manually. Thereby, the amount of available developers and their experience form influence factors to consider</p>
Preparation	<p>Applying this pattern essentially requires realizing a parser, model-to-model transformations, code generation rules and guidance documents to systematize the reimplementation. In addition, it can be necessary to realize a semantic analyzer, dedicated reverse engineering algorithms or model views</p>
Example	<p>In the first feasibility study (cf. Section 7.2), we applied the pattern on the <i>Back Navigation</i> and <i>Business Logic</i> concepts of the <i>Controller</i> concern. The method described in [Fle+07] conforms to the pattern</p>
Known Uses	<p>The strategy realized by the pattern is comparable to the strategy followed by an MDA-based development process that starts on the platform-independent layer. While parts of the functionality to transform are automatically generated using a model-driven tool chain, remaining parts are completed manually</p>
Related Patterns	<p>Combines F₂, F₃ and F₄; Similar to F₅</p>

Table A.1 Characterization of the *Concept Recognition-Based Reimplementation Pattern*

¹ Depending on the situation, using an AST can be sufficient

Concept Recognition and Language Transformation-Based Reimplementation (F₈)

Intent	Perform a semi-automatic transformation of the legacy system's functionality into a new environment by combining a conversion- and a reimplementation-based transformation strategy
Strategy	Use an intermediate representation of parts of the functionality to transform on a platform-independent layer to enhance a direct mapping between the programming languages of the environments involved and to guide a manual reimplementation. The intermediate representation is reverse engineered from an ASG ² of the legacy system on a platform-specific layer and concretized into an ASG ² of the target environment. When the ASG ² of the legacy system is transformed into an ASG ² of the target environment, the model transformation used is dependent on the transformation of the intermediate representation. Subsequently, code is generated in the target environment that forms the basis for reimplementation activities performed by software developers
Structure	

Continued on next page

Applicability	Use when the functionality to transform is realized significantly different in the legacy and target environment. In addition, the complexity of a direct transformation should become low if parts of the functionality are made explicit while it should also enable to pointedly guide software developers. The use of an intermediate representation can enhance a direct transformation but requires addressing the concerns of reverse engineering, restructuring and mapping the functionality explicitly. The manifestation of these concerns as well as the remaining complexity of the direct transformation influences the efficiency and effectiveness of the pattern. Those parts of the functionality, which have not been converted, shall be reimplemented. Thereby, the amount of available developers and their experience form influence factors to consider
Preparation	Applying this pattern essentially requires realizing a parser, model-to-model transformations, code generation rules and guidance documents to systematize the reimplementation. In addition, it can be necessary to realize a semantic analyzer, dedicated reverse engineering algorithms or model views
Example	In the first feasibility study (cf. Section 7.2), we applied the pattern on the <i>Non-Data-Based Dialog Element</i> concept of the <i>View</i> concern
Known Uses	The strategy realized by the pattern is comparable to the strategy followed by an MDA-based development process that starts on the platform-independent layer. While parts of the functionality to transform are automatically generated using a model-driven tool chain, remaining parts are completed manually
Related Patterns	Combines F ₁ , F ₂ , F ₃ and F ₄

Table A.2 Characterization of the *Concept Recognition And Language Transformation-Based Reimplementation Pattern*

² Depending on the situation, using an AST can be sufficient

Platform-Independent Architecture Restructuring (A_2)


Intent	Perform an automated restructuring of a legacy system's architecture during its transformation into a new environment
Strategy	An intermediate representation of the system's architecture is used on a platform-independent layer. The representation is reverse engineered from a model of the system's functionality on a platform-independent layer. After the architectural model has been restructured, the changes are reflected back to the model that represents the functionality
Structure	
Applicability	Use when the architecture of the system to transform differs in the source and in the target environment and if the difference is too significant to perform the restructuring implicitly when converting the functionality. In addition, the structures that imply the architecture of the system and those that are affected by the restructuring need to reside in models on the platform-independent layer
Preparation	Applying this pattern requires realizing model transformations rules. In addition, it can be necessary to realize dedicated architecture recovery algorithms or model views
Example	In the first feasibility study (cf. Section 7.2), we applied the pattern on the <i>Business Module</i> concept of the <i>Modularization</i> concern
Known Uses	The strategy realized by the pattern is comparable to the strategy followed by architectural reengineering tools
Related Patterns	Similar to A_1

Table A.3 Characterization of the *Platform-Independent Architecture Restructuring Pattern*

Architecture Restructuring (A_3)

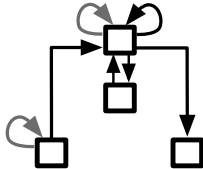
Intent	Perform an automated restructuring of a legacy system's architecture during its transformation into a new environment
Strategy	An intermediate representation of the system's architecture is used on a platform-independent layer. The representation is reverse engineered from models of the system's functionality on a platform-specific and platform-independent layer. After the architectural model has been restructured, the changes are reflected back to the models that represent the functionality
Structure	
Applicability	Use when the architecture of the system to transform differs in the source and in the target environment and if the difference is too significant to perform the restructuring implicitly when converting the functionality. Also, the structures that imply the architecture of the system and those that are affected by the restructuring need to reside in models on the platform-specific and platform-independent layer
Preparation	Applying this pattern requires realizing model transformations rules. In addition, it can be necessary to realize dedicated architecture recovery algorithms or model views
Example	—
Known Uses	The strategy realized by the pattern is comparable to the strategy followed by architectural reengineering tools
Related Patterns	Combines A_1 and A_2

Table A.4 Characterization of the *Architecture Restructuring Pattern*

Overview of the Method Engineering Process

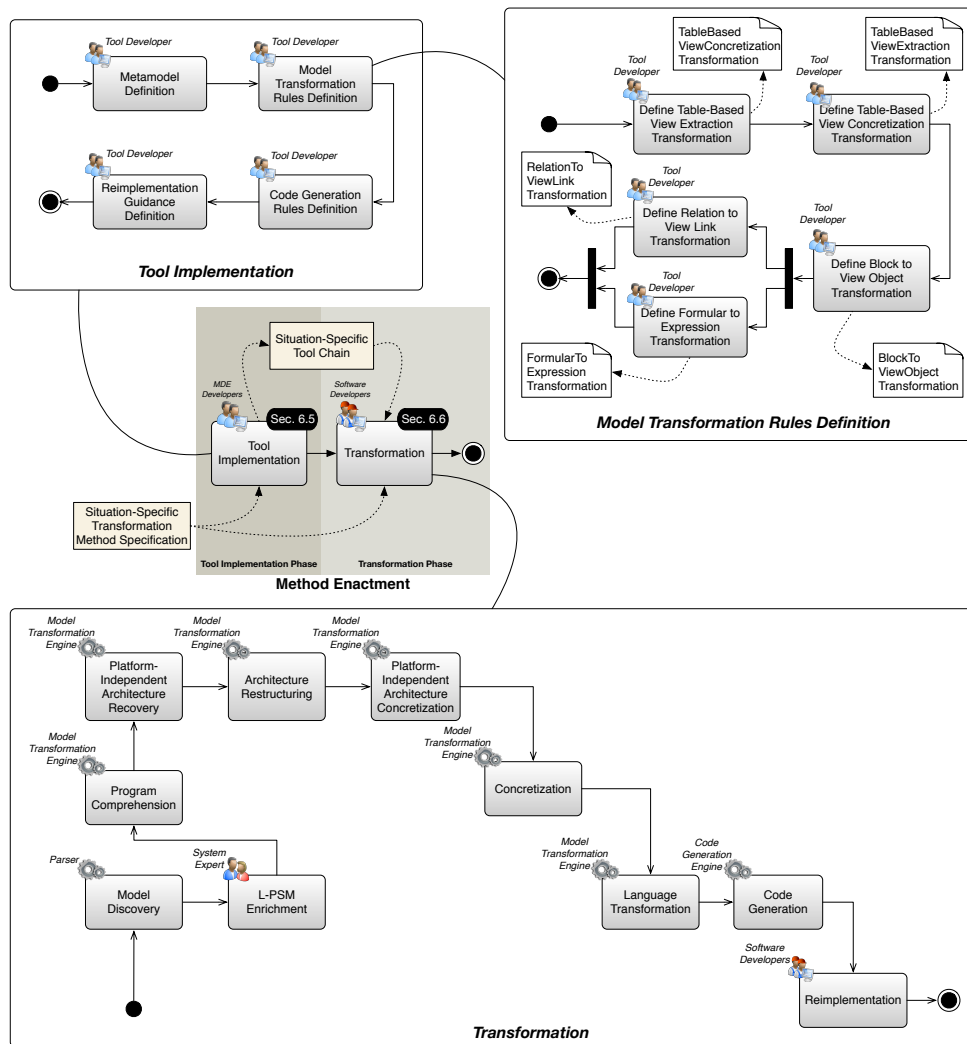


Figure B.1 Overview of the method enactment process of MEFiSto

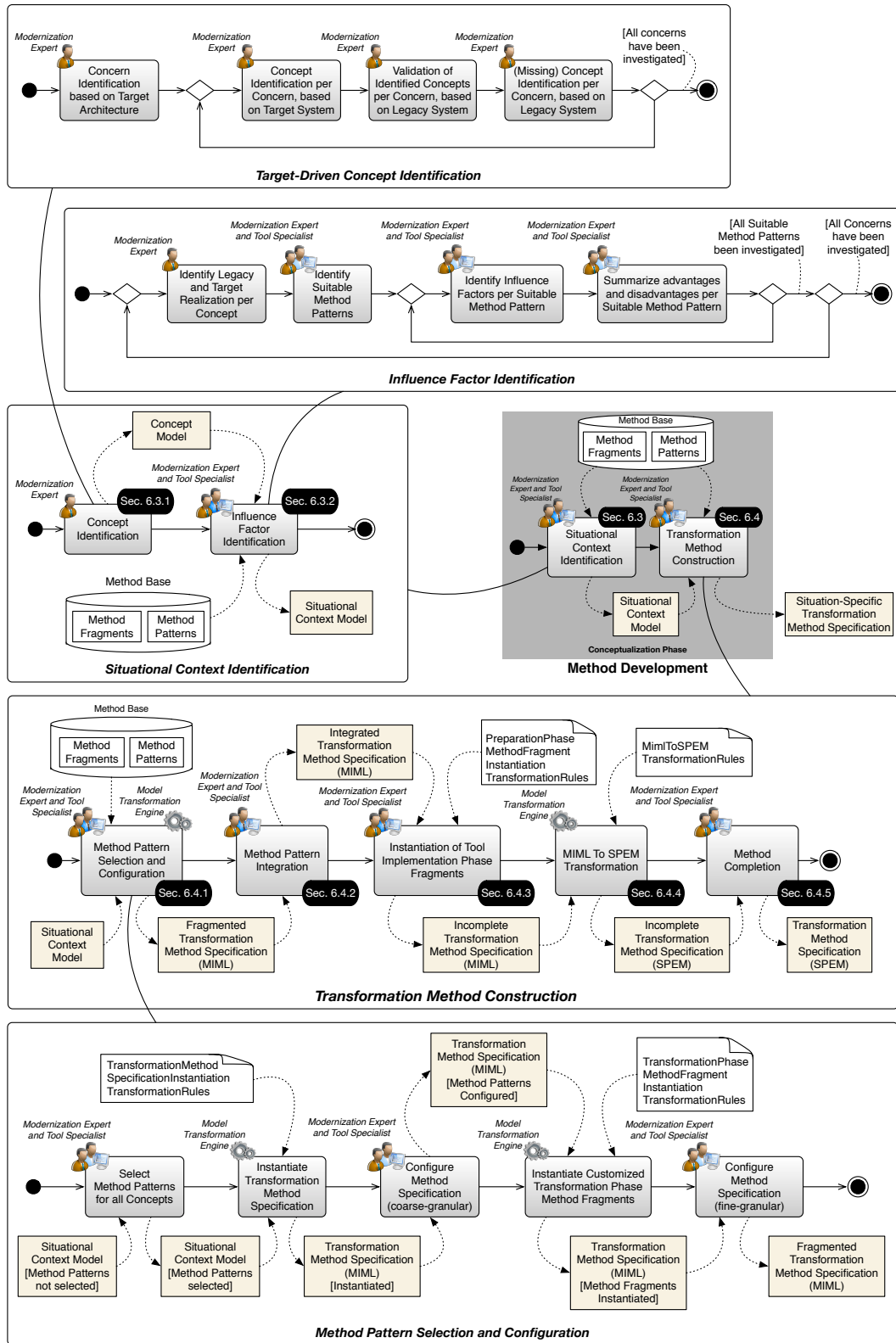


Figure B.2 Overview of the method development process of MEFiTo

abstract concept

An abstract concept represents a language-independent idea of a computation or problem solving principle [KNE92].

abstract syntax graph

An abstract syntax graph is an abstract syntax tree that has been extended by semantic edges, e.g., edges that link the use of an identifier to its declaration [KGW98].

abstract syntax tree

An abstract syntax tree is a tree whose nodes represent syntactic constructs of source code, while edges represent the hierarchical relation between them [Aho+06, p.41].

architectural concept

An architectural concept is an abstract concept and represents components or interfaces.

architectural platform-independent model

An architectural platform-independent model is a platform-independent model defined by the MEFiSTo framework that represents architectural structures of a software system to transform.

architectural style

An architectural style describes a family of software systems that share a common software architecture in terms of a pattern or structural organization [GS94, p.6].

architecture-driven modernization

The architecture-driven modernization is a software modernization paradigm, based on model-driven engineering concepts. It is defined by the OMG which promotes corresponding standards [BCW12, pp.45-47].

component

A component is a unit of composition with contractually specified interfaces and explicit context dependencies only [Szy02, p.548].

computation-independent model

A computation-independent model is an artifact defined by the MDA and ADM. It describes the context, requirements and purpose of the solution without any binding to computational implications [BCW12, p.40].

concept model

The concept model is a directed, acyclic and connected graph. Its nodes are shared concepts, while edges between them represent is-a or consists-of relations.

controlled flexibility

Controlled flexibility describes the ratio of a method engineering approach between the flexibility it provides to adapt a method to a given situation while controlling the result of the adaptation, i.e., while guiding the construction of a method.

conversion

Conversion is a software transformation strategy that aims to transform parts of a legacy system by executing a formal description, using the source code of the legacy system as an input [SWH10, pp.11-12].

database schema

An (external) database schema is a set of definitions that specifies the content of a database, e.g., its tables and views [Dat03, p.39].

database table

A database table is a storage format for data in which the data is stored as a set of rows or, more precisely, tuples [Dat03, p.15].

database view

A database view is a derived representation of data stored in a database which results by evaluating an expression on a database [Dat03, p.72].

discipline

A discipline is a categorization of work (i.e., activities), based upon the similarity of concerns and cooperation of work effort [OMG08a, p.161].

evolution stage

The evolution stage is a stage in the staged software lifecycle of a software system. In this stage, it is feasible to modify the information system in any way [RB00].

forward engineering

Forward engineering is the process of moving from high-level abstractions to the physical implementations of a system [CC90].

functional platform independent model

A functional platform independent model is a platform-independent model defined by the MEFiSTo framework that represents the functionality of a software system to transform by technology-independent concepts.

horseshoe model

The horseshoe model is a process that consists of the consecutive phases of reverse engineering, restructuring and forward engineering [KWC98]. In the context of MEFiSTo, it is a model which consists of a set of customized method fragments and conforms to a method pattern. The method fragments specify a method to transform a concept.

influence factor

An influence factor is a characteristic of a software modernization project that has an impact on the efficiency or effectiveness of a software transformation method.

influence factor model

An influence factor model is a model which consists of a set of influence factors.

Jasper Reports

Jasper Reports¹ is an open-source Java framework component to develop reports.

Java EE

Java EE² is a platform to develop business information systems based on Java.

language concept

A language concept is a syntactic entity of a programming language.

¹<http://community.jaspersoft.com/project/jasperreports-library> (accessed March 22th, 2016)

²<http://www.oracle.com/technetwork/java/javaee/index.html> (accessed March 22th, 2016)

layered architecture

A layered architecture is an architectural style in which a software system is organized hierarchically into layers, each layer providing service to the layer above it and serving as a client to the layer below [GS94, p.11].

legacy platform-specific model

A legacy platform-specific model is a platform-specific model defined by the MEFiSTo framework that represents the software system to transform by technology-specific concepts of the legacy environment.

legacy system

A legacy system is any software system that significantly resists modification and evolution to meet new and constantly changing business requirements [BS95, p.3].

metamodel

A metamodel is the explicit specification of a simplification by defining relevant concepts, relationships between these concepts and possibly logical assertions [BG01].

metamodel profile

A metamodel profile defines an extension of a metamodels by specifying additive adaptations, e.g., the addition of new classes, relations or constraints [OMG15b, pp.250-270].

method

A method is a description of how to systematically perform an endeavor. This comprises a process and its contained activities, artifacts, roles, tools and relationships between these elements on varying levels of granularity [ES10].

method base

A method base is a repository that contains reusable building blocks of methods, examples being method fragments or method patterns [Bri96].

method engineering

Method engineering is the engineering discipline to systematically develop or adapt methods [Bri96].

method fragment

A method fragment is a reusable, atomic building block of a method, i.e., a single activity, artifact, role or tool. [HS+14, pp.27-31].

method part

A method part is a reusable building block of a method at any level of granularity, examples are method fragments, method chunks or method components [HS+14, p.4].

method pattern

A method pattern is associated with a problem that shall be addressed by enacting a method. It encodes the solution in the form of construction guidelines for a method, i.e., it specifies which method fragments to use and how to assemble them.

model

A model is a reduced representation of something with an intended goal in mind [Sta73].

model transformation

A model transformation is the automatic generation of a target model from a source model, according to a set of transformation rules. Thereby, each rule specifies how one or more constructs in the source language can be transformed into one or more constructs in the target language [KWB03, p.24].

model view

A model view is a derived model that results when evaluating a query on another model or model repository.

model-driven architecture

The model-driven architecture is a model-driven development paradigm. It is defined by the OMG which promotes corresponding standards [BCW12, pp.9-10].

model-driven development

Model-driven development is a software development paradigm, based on model-driven engineering concepts [BCW12, pp.9-10].

model-driven engineering

Model-driven engineering is a paradigm that uses models as the primary artifacts when performing a software engineering task [BCW12, pp.9-10].

model-view controller architecture

A model-view controller architecture is an architectural style that separates a system into a model, a view and a controller part [Fow02, pp.330-332].

monolithic architecture

A monolithic architecture is an architectural style consisting of a single layer.

Oracle ADF

Oracle ADF³ is a Java EE framework of the vendor Oracle to develop information systems by providing infrastructure services and a visual and declarative development experience.

Oracle Forms

Oracle Forms⁴ is a component of Oracle Fusion Middleware of the vendor Oracle to design and build enterprise applications.

Oracle Reports

Oracle Reports⁵ is a component of Oracle Fusion Middleware of the vendor Oracle to develop reports.

phase

A phase represents a significant period in a project, ending with a milestone or a set of deliverables [OMG08a, p.156].

platform-independent model

A platform-independent model is an artifact defined by the MDA and ADM. It describes the behavior and structure of an information system, regardless of the implementation platform [BCW12, p.41].

platform-specific model

A platform-specific model is an artifact defined by the MDA and ADM. It describes the behavior and structure of an information system on a specific platform. Developers may use this artifact to implement the executable code [BCW12, p.41].

programming concept

A programming concept is an abstract concept and represents general programming strategies, data structures or algorithms [KNE92].

³<http://www.oracle.com/technetwork/developer-tools/adf/overview/index.html> (accessed March 22th, 2016)

⁴<http://www.oracle.com/technetwork/developer-tools/forms/overview/index.html> (accessed March 22th, 2016)

⁵<http://www.oracle.com/technetwork/middleware/reports/overview/index.html> (accessed March 22th, 2016)

rational method composer

The rational method composer⁶ is a method authoring tool developed by the vendor IBM. It enables to author SPEM-based methods.

redevelopment

Redevelopment is the endeavor to rewrite legacy system from scratch, using modern software techniques and the hardware of the target environment. The resulting system can extend beyond the functionality of the legacy system [BS95, p.8].

reimplementation

Reimplementation is a software transformation strategy that aims to transform a legacy system or parts of it by having them rewritten manually by developers. The resulting system does not extend beyond the functionality of the legacy system [SWH10, pp.11].

restructuring

Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior [CC90].

reverse engineering

Reverse engineering is the process of analyzing a subject system to create representations of the system in another form or at a higher level of abstraction [CC90].

semantic analysis

Semantic analysis is performed on an abstract syntax tree to check its semantic consistency and add additional information, like semantic edges [Aho+06, p.8-9].

service-oriented architecture

A service-oriented architecture is an architectural style which supports service-orientation, i.e., which supports a way of thinking in terms of services and service-based development and the outcomes of services⁷.

shared concept

A shared concept is an abstract concept of the legacy system that can be realized in the target environment.

⁶<http://www-03.ibm.com/software/products/en/rmc> (accessed March 22th, 2016)

⁷<http://www.opengroup.org/soa/source-book/soa/soa.htm> (accessed March 22th, 2016)

situation-specific

A software transformation method is situation-specific if it is efficient and effective against the background of the associated situation. Efficiency relates to properties of the enacted process, while effectiveness relates to properties of the resulting information system.

situational context

A situational context is associated to a software modernization project and consists of a set of influence factors. It needs to be considered in order to design a situation-specific transformation method.

situational method engineering

Situational method engineering is a kind of method engineering which encompasses all aspects of creating a method for a situation at hand [HS+14, p.5].

software and systems process engineering metamodel

The software and systems process engineering metamodel is a metamodel defined by the OMG that can be used to formally specify methods [OMG08a].

software architecture

The software architecture is the fundamental organization of a software system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution [IEEE00].

software migration

Software migration is a kind of software reengineering concerned with the transition of a legacy system to a new environment while retaining the system's data and functionality [Bis+99].

software modernization

Software modernization is a kind of software migration concerned with the transition of a legacy system to a new environment while adapting the system to the new environment.

software modernization method

A software modernization method is a method that is used guide a software modernization endeavor.

software modernization project

A software modernization project is established by a company to carry out a software modernization by defining and enacting a software modernization method.

software reengineering

Software reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [CC90].

software reengineering method

A software reengineering method is a method that is used to guide a software reengineering endeavor.

software transformation method

A software transformation method is an instance of the horseshoe model and used to guide the technical transition of a legacy system into a new environment during a software modernization endeavor.

software transformation package

A software transformation package represents a part, i.e., an increment, of a legacy system that is used during the incremental transformation of that legacy system [BS95, pp.13-14].

software transformation strategy

A software transformation strategy is a specific way of how to perform the technical transition of a legacy system during a software modernization, established examples being reimplementation, wrapping and conversion [SWH10, pp.10-13].

staged software lifecycle

The staged software lifecycle is a model that describes stages in the lifecycle of a software system after the initial development [RB00].

syntactic analysis

Syntactic analysis is performed by a parser on source code to create an abstract syntax tree which represents that source code [Aho+06, p.8].

technological space

A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities [AKB02].

transformed platform-specific model

A transformed platform-specific model is a platform-specific model defined by the MEFiSTo framework that represents the software system to transform by technology-specific concepts of the target environment.

wrapping

Wrapping is a software transformation strategy that aims to transform parts of a legacy system by encapsulating them using a wrapper. The wrapper acts as a connection point in the resulting information system [SWH10, pp.12-13].

Acronyms

4GL

Fourth-Generation Programming Language.

A-PIM

Architectural Platform-Independent Model.

ADM

Architecture-Driven Modernization.

ADMTF

Architecture-Driven Modernization Task Force.

API

Application Programming Interface.

ASG

Abstract Syntax Graph.

AST

Abstract Syntax Tree.

ASTM

Abstract Syntax Tree Metamodel.

CDO

Connected Data Objects.

CFG

Control Flow Graph.

CIM

Computation-Independent Model.

COTS

Commercial Off-The-Shelf.

CRUD

Create Read Update Delete.

DSL

Domain-Specific Language.

EMF

Eclipse Modeling Framework.

F-PIM

Functional Platform-Independent Model.

GASTM

Generic Abstract Syntax Tree Metamodel.

IDE

Integrated Development Environment.

JDAPI

Java Development API.

KDM

Knowledge Discovery Metamodel.

L-PSM

Legacy Platform-Specific Model.

LOC

Lines of Code.

MDA

Model-Driven Architecture.

MDD

Model-Driven Development.

MDE

Model-Driven Engineering.

MEFiSTo

Method Engineering Framework for Situation-Specific
Software Transformation Methods.

MIML

MEFiSTo Intermediate Modeling Language.

MoCo

Model-Integrating Component.

MOF

Meta-Object Facility.

MOFM2T

MOF Model to Text Transformation Language.

MVC

Model-View-Controller .

OCL

Object Constraint Language.

OMG

Object Management Group.

PIM

Platform-Independent Model.

PSM

Platform-Specific Model.

ReMiP

Reference Migration Process.

RMC

Rational Method Composer .

RUP

Rational Unified Process.

SASTM

Specialized Abstract Syntax Tree Metamodel.

SME

Situational Method Engineering.

SOA

Service Oriented Architecture.

SPEM

Software and Systems Process Engineering Metamodel.

T-PSM

Transformed Platform-Specific Model.

UML

Unified Modeling Language.