Model-Driven Software Development

Model-Driven Software Development

Technology, Engineering, Management

Thomas Stahl

and

Markus Völter

with

Jorn Bettin, Arno Haase and Simon Helsen

Foreword by Krzysztof Czarnecki

Translated by Bettina von Stockfleth

John Wiley & Sons, Ltd

Copyright © 2006 John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester,
West Sussex PO19 8SQ, England

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on www.wiley.com

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The Publisher is not associated with any product or vendor mentioned in this book.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the Publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 33 Park Road, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada M9W 1L1

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data

(to follow)

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN-13: 978-0-470-02570-3
ISBN-10: 0-470-02570-0

Typeset in insert
Printed and bound in Great Britain by insert Bell & Bain, Glasgow
This book is printed on acid-free paper responsibly manufactured from sustainable forestry
in which at least two trees are planted for each one used for paper production.

Contents

Foreword

by Krzysztof Czarnecki

Modeling is a key tool in engineering. Engineers routinely create models when analyzing and designing complex systems. Models are abstractions of a system and its environment. They allow engineers to address their concerns about the system effectively, such as answering particular questions or devising required design changes. Every model is created for a purpose. A particular model may be appropriate for answering a specific class of questions, where the answers to those questions will be the same for the model as for the actual system, but it may not be appropriate for answering another class of questions. Models are also cheaper to build than the real system. For example, civil engineers create static and dynamic structural models of bridges to check structural safety, since modeling is certainly cheaper and more effective than building real bridges to see under what scenarios they will collapse.

Models are not new in software development. Over the past few decades, the software industry has seen numerous analysis and design methods, each with its own modeling approaches and notations. More recently, we have witnessed the remarkable progress of Unified Modeling Language (UML), which now has a larger market penetration than any single previous modeling notation. Still, analysis and design models rarely enjoy the same status as code. The reality of most software projects is that models are not kept up-to-date with the code, and therefore they become obsolete and useless with time.

Model-Driven Software Development (MDSD) puts analysis and design models on par with code. Better integration of such models and code should significantly increase the opportunity to effect change through the models, rather than simply modifying the code directly. MDSD encompasses many different techniques across the entire spectrum of software development activities, including model-driven requirements engineering, model-driven design, code generation from models, model-driven testing, model-driven software evolution, and more.

The Model-Driven Architecture (MDA) initiative by the Object Management Group (OMG) has also certainly contributed a great deal to the recent surge of interest in software modeling and model-driven techniques. But the effects of that initiative have been both good and bad. On the positive side, I'm glad that modeling has been moved into the center of interest and that organizations are now trying to figure out how their current practices can be leveraged through model-driven techniques. At the same time, the marketing hype around MDA has tended to create some unrealistic expectations. Putting all this hype aside, I do think that MDSD has great ideas to offer, many of which can be put to work in practical situations today. Realizing these potentials requires a solid understanding of current MDSD technology, its applicability, and its limitations.

The authors of this book are at the forefront of MDSD research and practice. Markus and Jorn have organized and participated in a series of MDSD workshops at several OOPSLA conferences. Simon has participated in OMG's standardization efforts on model transformation. All the authors are pioneering this technology in practice in several domains, ranging from enterprise applications to embedded software in both small and large organizations, such as b+m, Siemens, and BMW.

I'm very pleased to introduce this book to you. In my view, this is one of the rare books in the model-driven space that talks not only about the vision, but also about what is possible today, and how to do it. After a minimal but necessary dose of basic concepts and terminology, the authors cover a wide range of MDSD technology topics such as metamodeling, component architectures and composition, code generation, model transformation, MDA standards, and MDSD tools. I particularly like the hands-on approach that the authors have taken. They illustrate available tools and techniques through concrete modeling examples and code snippets, and they give numerous practical tips and 'mind-the-gap' hints. In addition to the technology topics, the authors also present a comprehensive treatment of essential software engineering aspects such as testing, versioning, process management, and adoption strategies, as they apply to MDSD. The content is topped off with two case studies, that were inspired by realistic applications from the authors' collective experiences.

The authors present a broad perspective of MDSD that goes beyond MDA to cover a range of related approaches including software product lines, domain-specific languages, software factories, and aspect-oriented and generative programming. As in any young, dynamic, and still evolving field, the abundance of competing ideas, concepts, and parallel terminologies in today's model-driven space can be bewildering. As a result, the authors had to do a lot of 'sifting through the mud' to give us a clear and balanced picture of the entire model-driven field.In this book, they have done just that, tremendously well.

I invite you to explore this new and exciting field, and this book is a great place to start!

Krzysztof Czarnecki
Waterloo, January 2006

Part I
Introduction

1 Introduction

1.1 The Subject of the Book

This book is about *Model-Driven Software Development*, or ‘MDSD’. A less precise but common name for this discipline is *Model Driven Development* (MDD). Maybe you wonder why we decided to write such a book. We believe that Model-Driven Software Development is quite important, and will become even more so in the future. It is the natural continuation of programming as we know it today.

The application of models to software development is a long-standing tradition, and has become even more popular since the development of the Unified Modeling Language (UML). Yet we are faced with ‘mere’ documentation, because the relationship between model and software implementation is only intentional but not formal. We call this flavor of model usage *model-based* when it is part of a development process. However, it poses two serious disadvantages: on one hand, software systems are not static and are liable to significant changes, particularly during the first phases of their lifecycle. The documentation therefore needs to be meticulously adapted, which can be a complex task – depending on how detailed it is – or it will become inconsistent. On the other hand, such models only indirectly foster progress, since it is the software developer’s interpretation that eventually leads to implemented code. These are the reasons why – quite understandably – many programmers consider models to be an overhead and see them as intermediate results at best.

Model-Driven Software Development has an entirely different approach: Models do not constitute documentation, but are considered equal to code, as their implementation is automated. A comparison with sophisticated engineering fields, such as mechanical engineering, vividly illustrates this idea: imagine, for example, a computer-controlled mill that is fed CAD[1] data that enables it to transform a model into a physical workpiece automatically. Or consider an automotive production line: your order for a car that includes custom features is turned into reality. Here, the actual production process is mostly automated.

These examples demonstrate that the *domain* is essential for models, just as for automated production processes. Neither the customer-oriented ‘modeling language’ for car manufacture – in this case, an order form – nor the manufacturer’s production line are able to build prefabricated houses, for example.

[1] CAD = Computer Aided Design

MDSD therefore aims to find domain-specific abstractions and make them accessible through formal modeling. This procedure creates a great potential for automation of software production, which in turn leads to increased productivity. Moreover, both the quality and maintainability of software systems increase. Models can also be understood by domain experts. This evolutionary step is comparable to the introduction of the first high-level languages in the era of Assembler programming. The adjective 'driven' in 'Model-Driven Software Development' – in contrast to 'based' – emphasizes that this paradigm assigns models a central and active role: they are at least as important as source code.

To successfully apply the 'domain-specific model' concept, three requirements must be met:

- Domain-specific languages are required to allow the actual formulating of models.
- Languages that can express the necessary model-to-code transformations are needed.
- Compilers, generators or transformers are required that can run the transformations to generate code executable on available platforms.

In the context of MDSD, graphical models are often used, but this is neither mandatory nor always suitable. Textual models are an equally feasible option. Typically, these models are translated into programming language source code to enable their subsequent compilation and execution.

1.1.1 MDA

If you are familiar with the Object Management Group's (OMG) Model Driven Architecture (MDA), you might think that this sounds a lot like MDA. This is correct to a certain extent. In principle, MDA has a similar approach, but its details differ, partly due to different motivations. MDA tends to be more restrictive, focusing on UML-based modeling languages. In general, MDSD does not have these restrictions. The primary goal of MDA is interoperability between tools and the long-term standardization of models for popular application domains. In contrast, MDSD aims at the provision of modules for software development processes that are applicable in practice, and which can be used in the context of model-driven approaches, independently of the selected tool or the OMG–MDA standard's maturity.

At present (2005) the MDA standardization process is still in its fledgling stages, which means that, on one hand, some aspects of the original MDA vision must be omitted, while others need be interpreted pragmatically to get practicable results. On the other hand, a practical *methodological* support for MDA is not necessarily the OMG's main focus. This is in part also reflected by MDA's goals.

In this book we take a closer look at the relationship between MDA and MDSD. For now, it is safe to state that MDA is a standardization initiative of the OMG focusing on MDSD.

1.2 Target Audience

Specific concepts, terminology and basic ideas must be understood by all those involved in an MDSD project, otherwise it cannot be completed successfully. The introductory chapters of this book are therefore mainly dedicated to these aspects.

1.2.1 Software Architects

Three aspects concerning MDSD are relevant for the software architect:

- First, the approach requires a clear and concise definition of an application's architectural concepts.
- Furthermore, MDSD often takes place not only as part of developing an entire application, but in the context of creating entire product lines and software system families. These possess very specific architectural requirements of their own that the architects must address.
- In addition, a totally new development approach must be introduced to the project. This is due to the separation of models and modeling languages, of programs and generators, of respective tools and specific process-related aspects.

All these issues will be discussed in this book.

1.2.2 Software Developers

When dealing with a software development paradigm, it almost goes without saying that the role of the software developer is pivotal. To some extent, MDSD implies more precise and clearer views of aspects such as the meaning of models, the separation of domain-specific and technical code, the relationship between design and implementation, round-trip problems, architecture and generation, framework development, versioning and tests. When applied correctly, MDSD will make the software developer's work much easier, help to avoid redundant code, and enhance software quality through the use of formalized structures.

1.2.3 Managers and Project Leaders

Economic considerations such as the cost-value ratio or the break-even point underlie the decision to use Model-Driven Software Development. There is no 'free lunch': model-driven software comes with a price too. There are many project contexts for which a model-driven approach can be recommended, but there are some circumstances under which we would advise against it. Even though the focus of this book is technical, we will take into account organizational and economic aspects that are relevant from the project's or company's viewpoint.

A model-driven approach also impacts project organization and team structure as well as the software development process. We will address this as well.

1.3 The Goals of the Book

The goal of the book is to convince you, the reader, that MDSD is a practicable method today, and that it is superior to conventional development methods in many cases. We want to encourage you to apply it sooner rather than later, since MDSD is neither merely a vision nor dry theory. To this end, we want to equip you with everything you need. If you already practice MDSD, this book might offer you some advice or provide further insight into specific topics or fields.

More specifically, we pursue a number of ‘subgoals’ – independent of the book’s structure – that we want to elaborate briefly here.

First, we introduce the theoretical framework for MDSD, its basic concepts and terminology. We also touch on related topics and approaches, such as OMG’s Model Driven Architecture (MDA). We also want to provide hands-on help for specific MDSD-relevant issues. Among these are metamodeling (with UML and MOF[2]), code generation, versioning, testing, as well as recommendations for choosing the right tools. Organizational and process-related issues are also very important to us. Additionally, we want to argue for MDSD from an economical standpoint.

Although it is impossible to work without at least some theoretical basis, the book first and foremost aims to provide practical support, as well as taking a more detailed look at some of the relevant theoretical issues mentioned above. Best practice, as well as the dissemination of concrete experiences are important to us, as well as, in part, personal opinions. A number of case studies from various domains supplement the more detailed parts of the book.

We also wish to answer prevailing questions and address current discussions, so an outlook on trends and visions in the MDSD context completes the book.

1.4 The Scope of the Book

This is not an MDA book. We describe the basic concepts and terminology of the OMG MDA standard as well the underlying vision, and we also offer a synopsis of the current state of standardization (see Chapter 12). However, the book represents our own views and experiences. Secondary literature about MDA can be found in sources such as [Fra02], as well as in the OMG specification itself.

Our book does not intend to define a cohesive, heavyweight MDSD development process. Instead, we report on best practices that lend themselves to being used in agile processes, as described by Crystal [Coc01], and in the context of product line development for the construction of customized development processes (see Section 16.2).

1.5 The Structure of the Book and Reader Guidelines

This book describes the model-driven approaches which the authors have successfully applied in practice for many years. We look at the subject-matter from a technological as well as from an engineering and management perspective, as you will see from the book’s structure.

- *Part 1 – Introduction.* This part contains the introduction you are reading, plus an explanation of the most important basic ideas behind MDSD, and the basic terminology derived from the Model Driven Architecture. We then proceed to address the architecture-centric flavor of MDSD, which is ideally suited for a practice-oriented introduction. We convey the concrete techniques based on a comprehensive case study from the e-business/ Web application field, followed by a more comprehensive MDSD concept formation

[2] MOF = Meta Object Facility, an OMG standard

building on the points made. This chapter is extremely important, particularly because the rest of the book is based on the terminology defined here. This is also where you can find the conceptual, artifact-related definition of MDSD. The first part of the book is completed by a classification of and distinction between related topics such as agile software development.

- *Part II – Domain Architectures.* Domain architecture is the core concept of MDSD. Among other aspects, it contains the domain's modeling language and the generation rules that are supposed to map the models to a concrete platform. This part of the book conveys best practices for the construction of such domain architectures. The chapter on metamodeling forms its basis, followed by a detailed examination of the special role of the target architecture in the MDSD context. The three following chapters take a more detailed look at building transformations, including a description of code-generation techniques and QVT-based model-to-model transformations (QVT is the OMG's standard for model-to-model transformations). A short comparison with interpreter-based approaches is also included. For building domain architectures, generic tools are best used, so the chapter on tool architecture and selection provides some background for this. Finally, we take a deeper look at the MDA standard in the last chapter of Part II.
- *Part III – Processes and Engineering.* In the third part of the book we deal with process-related aspects of MDSD and engineering issues that assume specific characteristics through MDSD. Here at last it should become clear that MDSD is not just a technology. We present a number of best practices that can be combined into a practical and pragmatic development process, look at architecture development, and take a glimpse into product-line engineering. Following that, we tackle testing and versioning issues. We finally look at two case studies, one from the embedded domain, the other from the world of enterprise systems.
- *Part IV – Management.* The last part of the book is aimed primarily at IT managers and project leaders. It can largely be read independently from the rest of the book. We take a closer look at economic and organizational aspects and discuss adoption strategies. The first chapter of this part includes a FAQ[3] section of MDSD-related questions.

We have taken the utmost care in structuring this book so that its didactic effect is optimal when read sequentially in spite of its cyclic dependencies. However, since we address a divergent audience, some readers might initially wish to read the book selectively. In this context, please note that readers whose interest is primarily technical and who already possess some MDA knowledge can start directly with the case study in Part I, continue with Chapter 4, then immerse themselves in the technical issues addressed in Parts II or III before moving on to the rest of the book.

If you want to know what the economic advantages of MDSD are before learning about MDSD in more detail, please read Chapter 18 first. To gain a better understanding of it, we recommend that you also read Chapter 2.

[3] FAQ: Frequently asked questions

1.6 The Accompanying Web Site

Some topics dealt with in this book are undergoing rapid evolution, while others could only be touched upon due to space limitations. The book's accompanying Web site can be found at *http://www.mdsd-buch.org.* You can find up-to-date information there, as well as interesting links that we update on a regular basis.

1.7 About the Authors

Thomas Stahl works as chief software architect at b+m Informatik AG, where he is responsible for project-centric architecture management, including reusable software assets. He creates software architectures and frameworks and accompanies their use in both large and small projects. He also works as a consultant. His main focus is currently the field of Model-Driven Software Development, in which he has significant and practical long term experience. The creation of a good MDSD-generator framework was a pioneering effort. It is a popular Open Source project (*http://www.openArchitectureWare.org*) and is supported by a very active developer community. Besides his project-related work in several domains, Thomas writes articles for IT magazines and speaks at software conferences. He spends his spare time, among other things, as an active musician. He can be reached at *t.stahl@bmiag.de*

Markus Völter works as an independent consultant in software technology and engineering. His work focuses on software architecture and Model-Driven Software Development, as well as on middleware. Markus has extensive experience with these topics in many sectors, including the automotive industry, science, health care, telecommunications, and banking, as well as Web-based systems and telematics. He has worked and consulted for many leading enterprises, mostly but not exclusively in Germany, in projects ranging from three to 150 developers. Markus is also a regular speaker at international software conferences, as well as an active member of the international patterns community.

In addition to this book and its German predecessor, Markus has also co-authored two patterns books, *Server Component Patterns* and *Remoting Patterns*, both published in Wiley's Software Design Pattern series. He has also contributed to a German book on software architecture. As with Tom Stahl, Markus is also a contributor to the openArchitectureWare framework. When not working, Markus spends his time in his sailplane. He can be reached via *http://www.voelter.de* or at *voelter@acm.org*.

1.8 About the Cover

When we discussed the cover we thought that we wanted something that resembled the concept of 'model-driven' in some way. So, showing 'something', as well as a 'model of something' was the idea. We thought about buildings or machines. Then we thought about using an aircraft, since Markus' hobby is flying. After a couple of draft covers we agreed on the final one. This shows an Alexander-Schleicher ASW 27 – one of the highest-performance racing-class gliders – as well as a technical drawing of the same plane. The picture shows Markus' own plane with the German registration D-6642. If you are interested in more information about this aeroplane, you might want to visit the manufacturer's site at *http://www.alexander-schleicher.de*. A huge collection of photos of this and other such aircraft can be found at *http://www.voelter.de/flying/pictures.html* and *http://www.schogglad.de*. Have fun.

1.9 Acknowledgments

Writing a book like this is more challenging than you might think. Without the support of a large number of people its completion would have been even more of a challenge, which is why want to thank the following people.

First of all, there are our contributing authors, Jorn Bettin, Arno Haase, and Simon Helsen. Jorn contributed mainly to the book's section about management, helped us with terminology issues, and provided valuable input regarding the structuring of this work. Simon provided us with a unique insight into QVT and the processes and institutions behind it. We could not have written the QVT chapter without this vital input. Arno added to the book by looking into how interpreters fit into the MDSD world. Of course we also want to thank Krzysztof Czarnecki for writing a great foreword.

We wish to thank Bernd Oesterreich for discussions as well as for the material that found its way into the first case study in the book. We also thank Peter Roßbach for the interesting discussion about MDSD, particularly from the test perspective.

We – and especially Simon – also want to thank Sreedhar Reddy and Mariano Belaunde for their feedback on the QVT chapter. Gabor Karsai and Akos Ledeczi provided support in the context of MIC and GME – thanks for that! Thanks also to Juha-Pekka Tolvanen for providing us with screenshots for MetaEdit+.

Furthermore, we very much wish to thank our reviewers for fruitful discussions and useful feedback regarding many details, and also for input regarding the structuring of this book. The reviewers were, in alphabetical order, Frank Derichsweiler, Wolfgang Görigk, Michael Kircher, Michael Kunz, Wolfgang Neuhaus, Jürgen Rühle, Martin Schepe, Klaus-Dieter Schmatz, Eberhard Wolff, and Ghica van Emde Boas.

A very big thanks goes to our copy-editor Steve Rickaby of WordMongers. As with Markus' previous books, he has done a wonderful job of polishing the manuscript with regards to language, as well as other (small and not-so-small) issues.

We also want to thank Rene Schoenfeldt, who was our editor for the original German edition of the book. He is a great to work with. The same is true for the people at Wiley for this English edition: specifically we want to thank our editor Sally Tickner.

Thomas wishes to expresses his gratitude to Markus for the extensive amount of work and time he has spent on the updates that went into the English edition. He also most profoundly wants to thank his wife Anja and his children, who gave him the support he needed and greatly helped him with their considerateness.

2 MDSD – Basic Ideas and Terminology

This chapter introduces the most important basic concepts of Model-Driven Software Development, as well as the motivation for them. We prefer the abbreviation MDSD for Model-Driven Software Development over the less-precise variant 'MDD' (Model Driven Development). The first abbreviation has become more popular in the software modeling community over the past two years.

The Object Management Group's Model Driven Architecture (MDA) is both a flavor and a standardization initiative for this approach. Our focus here is its practicability in software projects. In many respects our concepts and experiences are congruent with those of OMG's MDA vision, but in other respects they differ. We point out the latter and discuss them. Apart from this, the MDA terminology, due to its standardization and pervasiveness, is extremely useful for providing an introduction to this topic, and this is exactly how you should approach the second part of this chapter: MDA provides the basic terminology for MDSD. The chapter's third part introduces the concepts that have been missing until then and which are required to understand the case study.

2.1 The Challenge

In the twenty-first century software is all around us. The software industry has become one of the largest on the planet, and many of today's most successful businesses are software production companies or offer services in the software field.

Software is today a relevant part of the machinery of all technology-based and many service-based businesses. High software development costs have significant economic impact, and bad software design, which impairs the productivity of users, can have even more serious consequences.

Many manufacturers of business software are so involved in dealing with the constantly-changing implementation technologies that productivity effort and risk management fall behind. Neither off-shoring, nor the newest generation of infrastructure software such as integrated development environments (IDEs), EAI[1] or BPM[2] tools and middleware, are much use here. In most cases, productivity problems are the result either of insufficient consistency or openness in

[1] EAI = Enterprise Application Integration

[2] BPM = Business Process Management

the application architecture, or of inadequate management or dependencies between various software components and unsuitable software development processes.

2.1.1 Historical View

The nineteen-nineties were mainly influenced by two software development paradigms. At the beginning of the nineties, these were Computer Aided Software Engineering (CASE) and fourth-generation languages (4GLs). In the second half of that decade, Object-Orientation entered the mainstream.

CASE methods and the corresponding tools were expensive, and proprietary approaches collided with a growing awareness of open standards. Many companies had bad experiences with some manufacturers, so eventually not only the tools but also the model-based software development approach were dumped. Object-orientation did not keep all of its promises, but it did become the foundation of component technologies, and object-oriented languages successfully replaced the previous generation of programming languages.

With the departure of 4GLs and CASE, OO modeling tools became the center of tool manufacturers' attention, resulting in the Unified Modeling Language (UML) notation standard and in tools based on a 'round-trip' philosophy. This enables smooth switching between UML models and the corresponding implementation code. Superficially, UML tools impress with their ability to keep models and code synchronized. However, on closer inspection one finds that such tools do not immediately increase productivity, but are at best an efficient method for generating good-looking documentation[3]. They can also help in understanding large amounts of existing code.

2.1.2 The Status Quo

The boundaries between modern UML tools and Integrated Development Environments (IDEs) are disappearing. For example, some UML tools have 'comfortable' code editors and integrated compilers, while traditional IDEs are equipped with UML modeling components. Software development tools, meanwhile, provide increasingly smart wizards that support users in the application of design patterns, the creation of user interfaces, and the generation of code skeletons for use with popular frameworks.

Although this approach constitutes an improvement compared to older UML tools that were only able to generate empty class skeletons, they strongly resemble CASE tools, as they are similarly inflexible. If, for example, a design pattern changes, today's UML tools are unable to transfer the effects automatically and iteratively to the source code of an application system without losing the abstraction.

Eventually, the weaknesses of mainstream IDEs and UML tools led to the formation of the OMG's MDA initiative. Appropriate tools allow users to define precisely how UML models are to be mapped to combinations of company-specific implementation technology. Unfortunately, in this context some traditional CASE tool manufacturers have spotted a second opportunity to offer their tools in a new package, as commercial MDA products. The tools cannot however be

[3] They offer a different graphic view of the code, but no real abstraction.

customized to meet individual requirements or customer needs, as they still adhere to the 'one size fits all' dogma. Most tools listed on the OMG's Web pages, however, actually deserve the label 'MDA tool'. In parallel with the progress in the field of software development tools, a significant evolution has also taken place in the field of software development methods, which has hardly been addressed yet in MDA.

The speedy propagation of agile approaches demonstrates an increasing resistance to traditional software development methods, which usually require a large amount of manually-created prose text documents. Today it is openly acknowledged that traditional methods required the production of such documentation, but in practice this cannot be reconciled with the market's demand for lower software development costs. Admittedly, agile methods such as Extreme Programming (XP) [Bec00] alone do not offer sufficient guidance for the creation of high quality software, and they do not scale to more complex projects. The odd misbelief that they can compensate for a development team's lack of analytical abilities or software design experience is particularly problematic.

2.2 The Goals of MDSD

Before we proceed to discuss the concepts and terminology of MDSD, we want to make a few comments on the goals of MDSD. However, we can only touch on how these can be achieved here.

- MDSD lets you increase your *development speed*. This is achieved through automation: runnable code can be generated from formal models using one or more transformation steps.
- The use of automated transformations and formally-defined modeling languages lets you enhance *software quality*, particularly since a software architecture – once it has been defined – will recur uniformly in an implementation.
- Cross-cutting[4] implementation aspects can be changed in *one* place, for example in the transformation rules. The same is true for fixing bugs in generated code. This *Separation of Concerns* [Lad03] promises, among other things, better maintainability of software systems through *redundancy avoidance* and *manageability of technological changes*.
- Once they have been defined, architectures, modeling languages and transformations can be used in the sense of a software production line for the manufacture of diverse software systems. This leads to a higher level of *reusability* and makes expert knowledge widely available in software form.
- Another significant potential is the improved *manageability of complexity through abstraction*. The modeling languages enable 'programming' or configuration on a more abstract level. For this purpose, the models must ideally be described using a problem-oriented modeling language.
- MDSD offers a *productive environment* in the technology, engineering, and management fields through its use of process building blocks and best practices. It thus contributes to meeting the goals described here.

[4] Aspects that cannot be easily located in a single module.

- Finally, based on the OMG's focus and history, the organization's primary motivations for MDA are *interoperability* (manufacturer-independence through standardization) and *portability* (platform-independence) of software systems. These goals that can be met only if a standardization – such as the OMG's MDA effort – is achieved. The same motivation has already led to the definition of CORBA[5]. To achieve these goals, the OMG aims at separating the specification of a specific functionality from its implementation on a specific platform. The MDA serves the purpose of providing guidelines and standards that should lead to a corresponding structuring of system specifications in the form of models.

Most of the goals presented here are not new. On the contrary, they represent something like the IT industry's 'Holy Grail': no-one is inclined to believe in beneficial promises anymore, and rightly so. But if you take a look at the history of IT or computer science, you can see that an ongoing evolution is taking place. High-level languages, object-orientation and component systems were milestones on the road toward meeting these goals – and MDSD is another. This paradigm takes us a small – or even a big – step closer to these goals.

2.3 The MDSD Approach

Each software has its inherent construction paradigms, expressed in the source code – an inner structure. How sound and how pronounced this structure consequently is directly influences development speed, quality, performance, maintainability, interoperability, and portability of the software. Those are extremely important key economic factors.

The problem is that it is difficult to recognize the actual construction paradigms on a programming language level, because their abstraction level is much lower. To put it differently, the much-treasured inner structure is present in a cloudy, distributed, and of course also a strongly individualized form. It is no longer directly represented in the system itself. Its quality varies, depending on the skills and interpretation of the developers.

The idea of modeling is not exactly new, and is used mostly for sophisticated development processes to document a software's inner structure. Developers then try to counteract the inevitable consistency problems with time-consuming reviews. In practice, these reviews and also the models are among the first victims when time presses – from a pragmatic point of view, even rightly so. Another approach is 'round-trip' or reverse engineering, which most UML tools offer, which is merely source code visualization in UML syntax: that is, the abstraction level of these models is the same as for the source code itself[6]. Visually it may be clearer, but the essential problem remains the same.

Model-Driven Software Development offers a significantly more effective approach: Models are *abstract and formal at the same time*. Abstractness does not stand for vagueness here, but for compactness and a reduction to the essence. MDSD models have the exact meaning of program code in the sense that the bulk of the final implementation, not just class and method skeletons, can be generated from them. In this case, models are no longer only documentation, but *parts of*

[5] CORBA: Common Object Request Broker Architecture (an OMG standard)

[6] In the meantime, UML tools have been improved to handle the J2EE programming model and can thus represent an EJB Bean through a UML class. However, abstraction cannot be taken any further than that, because the tool does not 'know' the application architecture's concepts. Unique mapping to the source code is also impossible.

the software, constituting a decisive factor in increasing both the speed and quality of software development. We emphasize 'model-driven' as opposed to 'model-based' to verbally highlight this distinction.

The means of expression used by models is geared toward the respective domain's problem space, thus enabling abstraction from the programming language level and allowing the corresponding compactness. All model-driven approaches have this principle in common, regardless of whether the domain is labeled 'software architecture', 'financial service systems', 'insurances', or 'embedded systems'. To formalize these models, a higher-level Domain-Specific Modeling Language (DSL) is required. From this 'bird's eye view', it doesn't matter whether this is a UML-based language or not.

Besides formal and abstract models, 'semantically rich', domain-specific platforms make up the second foundation pillar: prefabricated, reusable components and frameworks offer a much more powerful basis than a 'naked' programming language or a technical platform like J2EE. First and foremost, this means that the generator, which is supposed to transform the formal model, will be simplified once the generated code can rest on APIs of significantly higher quality. The introduction of reusable frameworks, super classes, and components to avoid code redundancy is not a new idea, but in the context of MDSD they serve additionally to intercept the model transformation half-way in the form of a well-formed platform, which causes a significant complexity reduction of the code generators[7].

Figure 2.1 shows the relationships in application development with MDSD.

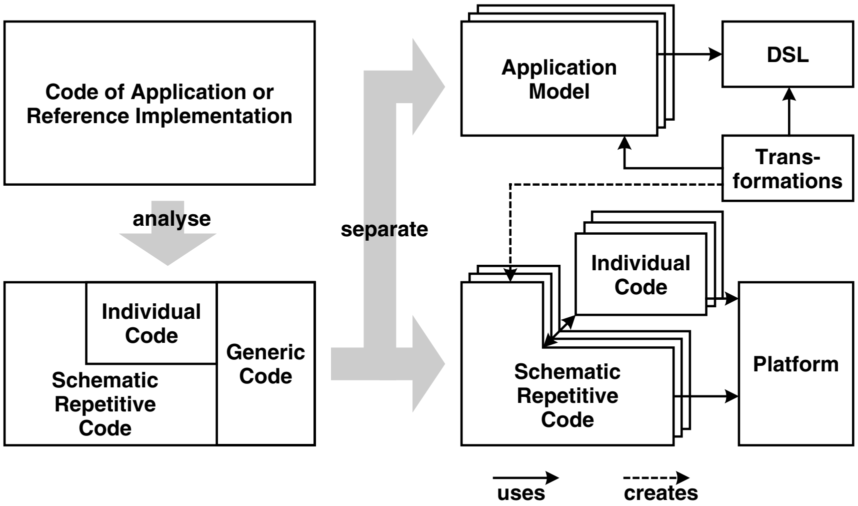


Figure 2.1 The basic ideas behind Model-Driven Software Development

Let's take a look at an existing application or a reference implementation (the upper left corner of the diagram). These are unique items with individual structures. We can restructure the

[7] The transformations become less complex because they don't have to generate code that runs on low-level platforms, but can assume that there is a platform that provides basic services. This reduces the complexity of the transformation, since the 'abstraction gap' is reduced.

code of these application in our minds so that three parts can be separated[8] (the lower left corner): a generic part that is identical for all future applications, a schematic part that is not identical for all applications, but possesses the same systematics (for example, based on the same design patterns), and finally an application-specific part that cannot be generalized. At this point, we won't make any statements about the significance of each part: in extreme cases, the application-specific part can even be empty. Model-Driven Software Development aims to derive the schematic part from an application model. Intermediate stages can occur during transformation, but in any case DSL, transformation, and platform will constitute the key elements. They must only be created once for each domain, for example 'enterprise software architecture' or 'inventory system for insurance' (lower right).

2.4 Basic Terminology

This section introduces the most important concepts and terms of the MDA standard, to establish the basic terminology for MDSD.

Domain-related specifications are defined in Platform-Independent Models (PIMs). To this end, a formal modeling language is used that is specific to the concepts of the domain to be modeled. In most cases, one would use UML that has been adapted via profiles to the respective domain, not least because of its tool support (see Section 6.5). These domain-specific descriptions are completely independent of the later implementation on the target platform. Such target platforms can be, for example, CORBA, J2EE, .NET or proprietary frameworks/platforms. Figure 2.2 illustrates this basic principle.

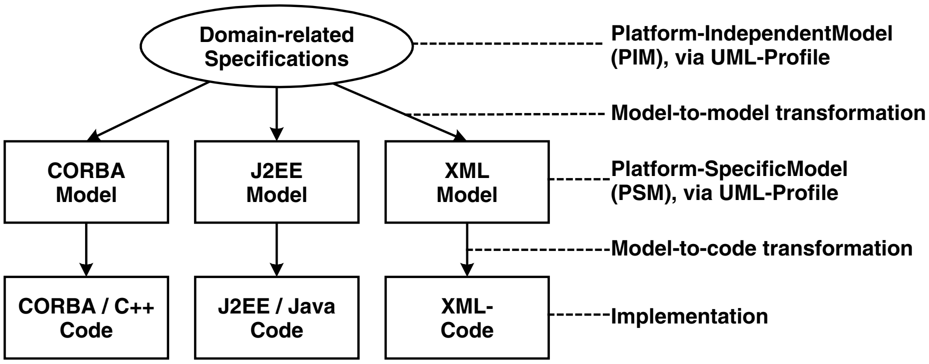

Domain-related Specifications
Platform-IndependentModel (PIM), via UML-Profile
Model-to-model transformation
CORBA Model
J2EE Model
XML Model
Platform-SpecificModel (PSM), via UML-Profile
Model-to-code transformation
CORBA / C++ Code
J2EE / Java Code
XML-Code
Implementation

Figure 2.2 The basic principle of MDA

Via model transformation, usually automated with tools, Platform-Specific Models (PSMs) are created from the Platform-Independent Models. These Platform-Specific Models contain the target platform's specific concepts. The implementation for a concrete target platform is

[8] Where appropriate through refactoring.

then generated with another tool-supported transformation based on one or more PSMs (see Figure 2.3).

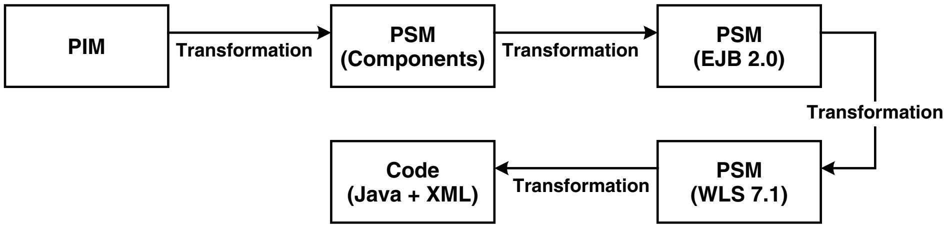
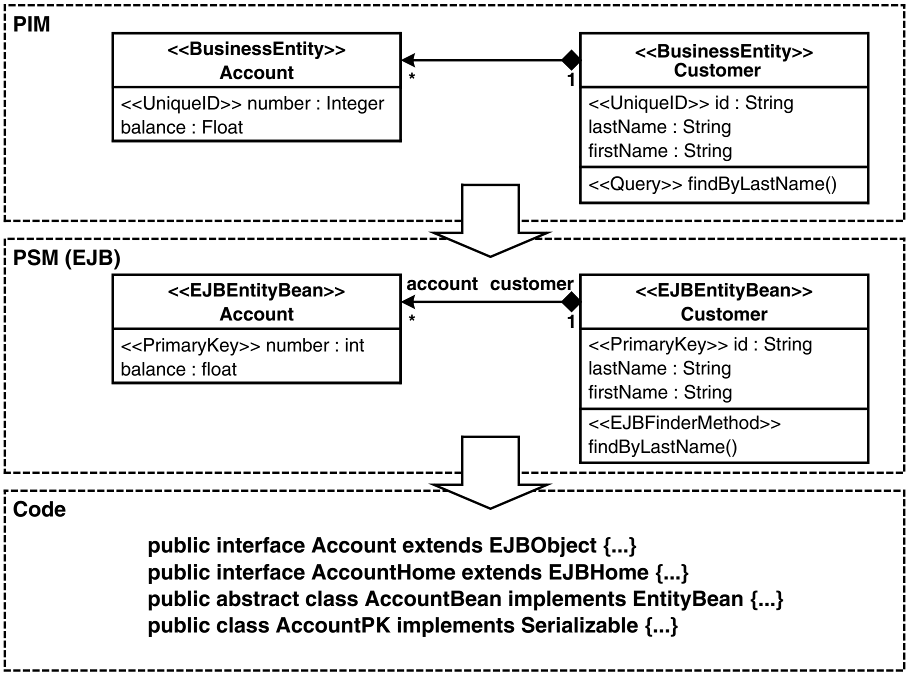


Figure 2.3 PIM, PSM and transformation

It is important to note that a PIM and a PSM are relative concepts – relative to the platform. In the example shown above, the EJB-PSM is specific to the EJB 2.0 platform, yet it is independent regarding its concrete, application server-specific realization.

Let's look at another example. Figure 2.4 shows a small part of a PIM. It shows a class model with two domain classes: *Customer* and *Account*. Both classes have the «*Business Entity*» stereotype, and both have an attribute that is assigned the stereotype «*UniqueID*». The method *findByLastName* features the stereotype «*Query*» under *Customer*.


PIM
<<BusinessEntity>> Account
<<UniqueID>> number : Integer
balance : Float
*
1
<<BusinessEntity>> Customer
<<UniqueID>> id : String
lastName : String
firstName : String
<<Query>> findByLastName()
PSM (EJB)
<<EJBEntityBean>> Account
<<PrimaryKey>> number : int
balance : float
account
customer
*
1
<<EJBEntityBean>> Customer
<<PrimaryKey>> id : String
lastName : String
firstName : String
<<EJBFinderMethod>> findByLastName()
Code

```
public interface Account extends EJBObject {...}
public interface AccountHome extends EJBHome {...}
public abstract class AccountBean implements EntityBean {...}
public class AccountPK implements Serializable {...}
```

Figure 2.4 An example that illustrates the relationship between PIM, PSM and code

The annotation of stereotypes on UML model elements allows us to change or specify the meaning of an element. A class with the stereotype «*Business Entity*» is not just a simple class, but is rather a self-contained entity in business applications. What this means in practice is determined by transformations that define how a stereotype such as «*Business Entity*», for example, is mapped to an existing platform such as J2EE.

Such an extension of the standard language vocabulary of UML through stereotypes is called a (UML) *profile*. It is a standard mechanism specified by the OMG to ensure openness, and is used here to define a formal modeling language. This formalization is mandatory for transforming a UML model into an MDA model. The concepts «*Business Entity*», «*UniqueID*», and «*Query*» are completely independent of the target platform. Dependency occurs through the transformation from PIM to PSM. Here, we find the stereotypes that are specific to J2EE: «*EJBEntityBean*», «*PrimaryKeyField*», and «*EJBFinderMethod*». These are also originally concepts that acquire their meaning through transformations, in this case transformations into the Java programming language.

The transformation eventually turns the PSM into source code, in which the concepts described here can be found in their concrete manifestation.

2.4.1 An Overview of MDA Concepts

The Model

A *model* is an abstract representation of a system's structure, function or behavior. MDA models are usually defined in UML[9]. In principle, the MDA formally considers even classic programming languages as MDA modeling languages that in turn maintain relationships with a platform. Without a doubt this is correct, but we are of the opinion that this approach occasionally hampers the elucidation of concepts, so from now on we will keep the terms *model* and *modeling language* clearly separate from the terms *program* and *programming language*.

UML models are not per se MDA models. The most important difference between common UML models (for example analysis models) and MDA models is that the meaning (semantics) of MDA models is defined formally. This is guaranteed through the use of a corresponding modeling language which that is typically realized by a UML profile and its associated transformation rules. We discuss these mechanisms in greater detail later in this chapter. All in all, this means that the mapping of a model to an existing platform is clearly defined.

The Platform

At first the MDA says nothing about the abstraction level of platforms. Platforms can build on each other, for example an Intel PC is a platform for Linux. Similarly, CORBA, J2EE, or Web Services are possible platforms for an e-business system, and C++ is a possible platform for CORBA. A well-defined application architecture, including its runtime system, can also be a platform for applications. We consider the latter idea of the key concepts for Model-Driven Software Development and discuss it in greater detail later on.

[9] According to the standard with MOF-based models – see Chapter 6.

UML Profiles

UML profiles are the standard mechanism for expanding the vocabulary of UML. They contain language concepts that are defined via basic UML constructs such as classes and associations, stereotypes, tagged values, and modeling rules (constraints) – see Figure 2.5.

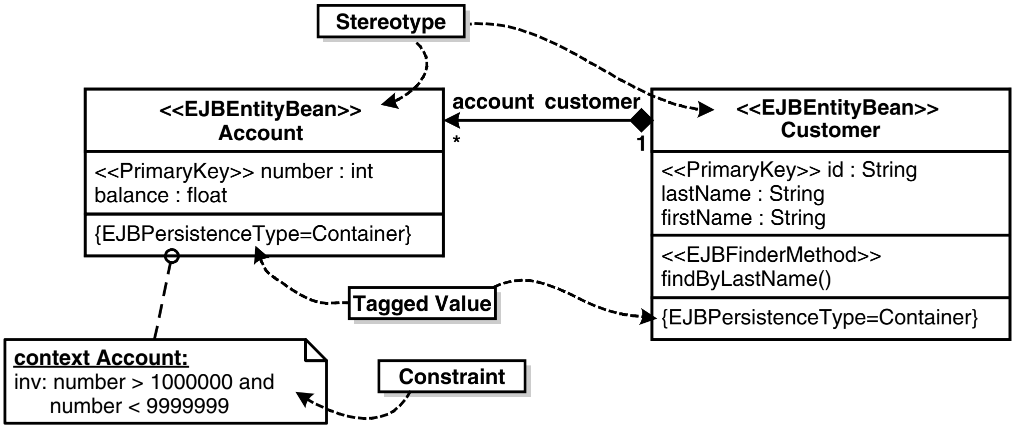


Figure 2.5 Use of a UML profile

A UML profile is defined as an extension of the UML metamodel. A metamodel defines, among other things, the basic constructs that may occur in a concrete model. Conceptually, a model is an 'instance' of a metamodel. Accordingly, the UML metamodel contains elements such as *Class, Operation, Attribute*, or *Association*. The metamodel concept is one of the most significant concepts in the context of MDSD. For this reason, we dedicate a whole chapter to it, Chapter 6. However, at this stage we are content just to gain an intuitive understanding. The relationship between the metamodel and profile is clarified in Figure 2.6, using a simplified example – a UML profile for Enterprise Java Beans (EJB).

In the UML profile, the standard UML concepts *Attribute, Class* and *Operation* are supplemented by the specific concepts *PrimaryKeyField, EJBEntityBean* and *EJBFinderMethod*. In addition, a new UML 2.0 language construct, an *extension*, is used. This is indicated by the filled-in inheritance pointer. To avoid confusion, we made these larger.

Additional extensions are defined through tagged values and modeling guidelines in the form of constraints. A constraint is usually annotated as a comment for the respective model elements: we use the formal constraint language OCL here. Tagged values are rendered as attributes of the stereotype.

A UML profile therefore offers a concrete notation for referencing metamodels from a model, and determines whether a certain model is 'well-formed', that is, valid or not. In short, it defines a formal modeling language as an extension of UML.

Further details of these relationships are elaborated on in Chapter 6.

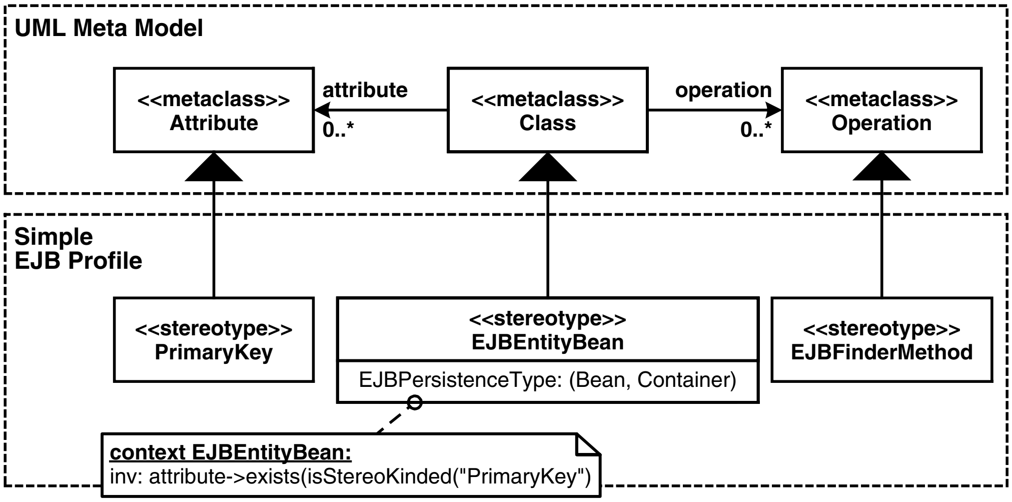


Figure 2.6 UML metamodel and UML profile for EJB (section of)

PIM and PSM

The separation of Platform-Independent Model (PIM) and Platform-Specific Model (PSM) is a key concept of the OMG's MDA. The background to this is as follows: concepts are more stable than technologies, and formal models are potentially useful for automated transformations. The PIM abstracts from technological details, whereas the PSM uses the concepts of a platform to describe a system (see Figure 2.7). The reverse route – the extraction of a PIM from a PSM – is extremely hard to automate, and in some cases impossible. That usually requires manual, intellectual work, which is somewhat awkwardly termed *Refactoring* in the MDA specification. (The meaning of *Refactoring* leans more toward equivalence transformations – see [Fow99].)

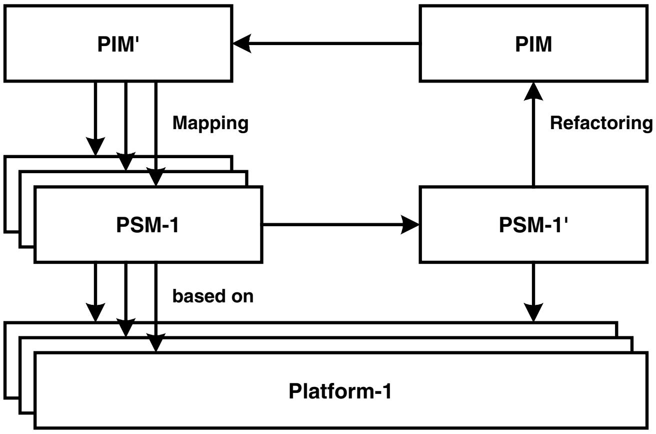


Figure 2.7 The relationship between PIM, PSM and platform

Transformations

Transformations map models to the respective next level, be it further models or source code. In terms of the MDA, transformations must be definable flexibly and formally based on an existing profile. This is a prerequisite for the desired automation of the transformation via generators.

Most of the currently-available MDA/MDSD tools define their transformation rules not between two metamodels, but instead for example use templates for the direct generation of source code, without the programming language's metamodel being formally known to the generator. However, generators exist that attach the transformation rules to the UML profile or, respectively, its corresponding metamodel. Such approaches are absolutely workable in practice, and are described in Chapters 3 and 9. The advantage of a transformation based on two metamodels (source and target) is mostly the elegant mapping from one metamodel to another. We doubt whether this paradigm is feasible for the generation of source code in practice, however.

Current generators solve this problem in a different way, through the use of proprietary transformation languages. In this context, JPython, TCL, JSP, XSLT, or custom script/template languages are used[10]. The generator templates defined with these languages principally work like macros and use the models as input data. As a consequence, at present no interoperability for model transformations exists: standardization is on its way, however – see Section 10.5. Here we will have to wait until standardization has been accomplished.

Chapter 12 provides a deeper insight into the MDA standard.

2.5 Architecture-Centric MDSD

In this section we want to supply the foundations that can enable you to understand the later case study: one flavor of MDSD that is termed *Architecture-Centric* MDSD (AC-MDSD). The approaches described here have evolved in the course of six years' practical experience with many projects, and particularly focus on practical usability.

2.5.1 Motivation

In contrast to the primary goals of the OMG for MDA, interoperability and software portability, AC-MDSD aims at increasing development efficiency, software quality, and reusability. This especially means relieving the software developer from tedious and error-prone routine work. Today developers are confronted with extremely complex software infrastructures: application servers, databases, Open Source frameworks, protocols, interface technologies and so on, which all need be connected to create robust and maintainable high-performance software. Due to increasing complexity in this field, the discipline of software architecture assumes more and more importance.

The existence of a software infrastructure also implies the existence of corresponding infrastructure code in the software systems using it. This is source code, which mostly serves to establish the technical coupling between infrastructure and applications to facilitate the development of domain-specific code on top of it. The J2EE/EJB programming model is a prime

[10] These languages are themselves domain-specific languages for the domain of defining code-generation templates.

example in this context: home and remote interfaces, Bean classes, descriptors – technical code that admittedly contains domain-related information such as method signatures, but which also exhibits a high degree of redundancy. After they have built four or five Enterprise Beans manually, if not before, a J2EE developer will long for a generator to create this type of infrastructure code – and can get this kind of support, typically in the shape of a preprocessor or an IDE wizard.

At best, some infrastructure components will bring their own 'helpers' for the generation of their own infrastructure code[11]. The problem here is that these tools do not 'know' each other, which is why they fall short of the possibility of a holistic and architecture-centric approach, as we will see in the case study.

Ergo, the goal of AC-MDSD must be integrated automation of infrastructure code generation and, as a consequence, the minimization of redundant infrastructure code in application development.

When we talk about infrastructure code, we are not talking about peanuts: measurements [Chapter 18] show that between 60% and 70% of modern e-business applications typically consists of infrastructure code.

2.5.2 Generative Software Architectures

As the adjective *architecture-centric* already implies, software architecture plays the central role in the MDSD flavor discussed here. Actually, a holistic, generative approach for the creation of infrastructure code can only work on the basis of a thoroughly worked-out and formalized software architecture.

You can imagine this as follows: the more and the better a software architecture has been elucidated, the more schematic the source code of applications using this architecture will become. If the architecture's definition consists only of slides representing the system infrastructure (databases, application server, mainframes, networks and so on) and maybe additionally the most important layers, it is likely that two developer teams will realize the same application in entirely different ways – including the implementation of the software architecture: two unique applications will be created.

If we assume however that a team of architects does some groundwork and develop some sort of technical reference implementation that shows the concrete realization of the most important software architectural aspects at the source code level, application developers can use this reference as a blueprint. Since the same technical realizations – notwithstanding domain variations – recur in development practice (for example use of a specific interface technology or an MVC pattern), the majority of the workload would be copy and paste programming. Of course, this sort of programming is much more efficient than individually thought-out code created from scratch.

In essence, the more of a software architecture's definition has been fleshed out in source code, the more schematic and repetitive the application development process will become. Schematic programming means mostly copy and paste, followed by modifications that depend on the domain context. This part of the work is clearly non-intellectual. If we pursue this train of thought, it is not too far-fetched to leave the tedious and error-prone copy/paste/modify job to a

[11] In the case of EJB this will for example be realized in Version 3.0.

generator, which ultimately leads to a *generative software architecture*. Here, all implementation details of the architecture's definition – that is, all architectural schemata – are incorporated in software form. This requires a domain model of the application as its input, and as output it generates the complete infrastructure code of the application – the very code that otherwise would need to be generated via a tedious copy/paste/modify process. To this end, the model only needs to have specific annotations that reference the architectural concepts defined as part of the generative software architecture.

Usually an architecture-centric UML profile is used for modeling in AC-MDSD. Thus a formal, architecture-centric application *design* is created. The model-to-code transformations are defined typically in the form of generator templates, so that the complete infrastructure code can be generated automatically from the architecture-centric design model. It is important to note that the model must already contain all relevant information for the generation of the infrastructure code – it is just a lot more abstract and more compact than the expanded code. The templates can use the entire infrastructure's power and base the generated code on this platform, as described in Section 2.3, simplifying the templates. Since the generation of the code is motivated by technical and architectural concerns, a 'semantic gap' remains: developers must manually create the application's actual domain code, that is, the actual, domain-specific functionality that is *not* infrastructure code.

There are various techniques for the integration of generated and manually-created code. We look at them in detail in Chapter 8 and Chapter 9. Figure 2.8 illustrates these correlations. They are explained further in the next chapter's case study, using a practice-oriented, realistic example.

A generative software architecture is a powerful means to achieve the goals we listed in Section 2.2. Its most important advantages are higher development speed and software quality, better maintainability, and practical reusability – reusability within one application, but of course even more beyond the boundaries of a single application. A generative software architecture can support an entire group or family of architecturally-similar applications – a *software system family*. In effect, AC-MDSD deals with the creation of generative software architectures for software system families, instead of creating unique products.

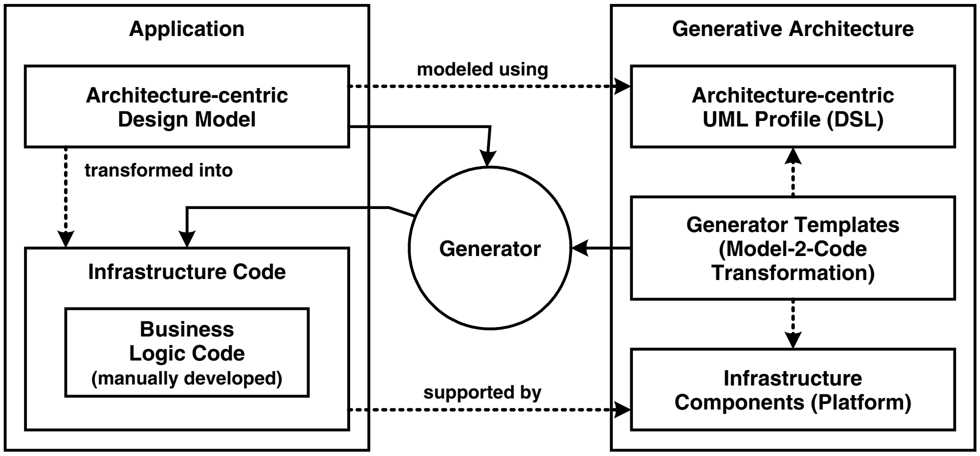

Application
Architecture-centric Design Model
transformed into
Infrastructure Code
Business Logic Code (manually developed)
modeled using
Generator
supported by
Generative Architecture
Architecture-centric UML Profile (DSL)
Generator Templates (Model-2-Code Transformation)
Infrastructure Components (Platform)

Figure 2.8 The principle of architecture-centric MDSD

2.5.3 Architecture-Centric Design

The defined design language (typically a UML profile) contains the software system family's architecture concepts in the shape of a 'platform-independent'[12] abstraction. Designers use this design language to create the domain's application design in the form of PIMs. Other than when dealing with the OMG–MDA vision, they will in most cases deliberately forego the transformation of these PIMs into explicitly visible, platform-dependent UML models (PSMs) when working with AC-MDSD.

Practical project experience has hitherto proved that this simplification is usually more useful than the additional degrees of freedom gained with PSMs. As a consequence, one need not control, manipulate, and enrich the various intermediate transformation results with specific information[13]. This not only allows for a more efficient development, but also avoids potential consistency problems: a manual change of an intermediate model might result in an inconsistency with higher abstraction levels that is not automatically correctable.

Similarly, we forego reverse engineering from the source code to the PIM, which in general is not feasible anyway. A model that has been created 'backwards' from source code is naturally as little abstract as the source code itself. Only its presentation is different, perhaps providing better understandability for some purposes. For specific arbitrary sections of source code, a PIM from which the program could be derived via transformation[14] may not exist – especially if the PIM modeling language focuses on a specific domain such as software architecture for e-business systems. In the context of MDA specifications, this fact is more or less ignored by the OMG however.

Some members of the MDSD tool-builders community anticipate tool-supported wizards or some similar solution that will at least enable semi-automated reverse engineering. In our opinion this is a concession rather than a goal-oriented concept[15] – at least where newly developed software is concerned. Admittedly, this view may first be perceived as being disadvantageous, depending on your personal work preferences, but in truth it is an advantage, as we will learn later on. Basically, AC-MDSD builds on forward engineering.

This forward-engineering based, generative approach allows us to derive conclusions about generated applications from the 'hard facts' of architecture-centric models. A generative architecture can guarantee a loose coupling of components or the absence of access paths between different application layers. For example, it can ensure that a presentation layer, for example a Web user interface, cannot access a database's SQL interface directly.

At this point it's important to note that forward engineering is not to be mistaken for a model that uses the waterfall approach to development. It merely means that design changes must be made to the model instead of the source code, which of course does not mean that the whole application must be modeled at once. We concede that forward engineering does not exclude such an approach, but this does not mean that it is mandatory. In fact, we favor an iterative, incremental process [Oes01].

[12] Platform-independence is a relative term. Here, it refers to the independence of standard software platforms like J2EE.

[13] We are not against the modularization of transformations through successive execution here, yet we do not favor explicitly visible and manipulable intermediate results.

[14] Mathematicians would say that the mapping of a PIM model to a programming language is not surjective.

[15] In the context of adaptation of legacy software for MDSD, reverse engineering can make sense, quasi as bootstrapping.

Let's now examine an example of such a PIM, shown in Figure 2.9. This model does not reveal anything about the technologies that were used – the technological realization of such models is defined only once it is mapped to a concrete platform. A formal UML design language is created through the semantic enrichment of the model with stereotypes, tagged values, and constraints. For AC-MDSD, the abstraction level of this language lies on the level of architectural concepts, which is why we speak of *architecture-centric design*. In other words: the domain of AC-MDSD is software architecture.

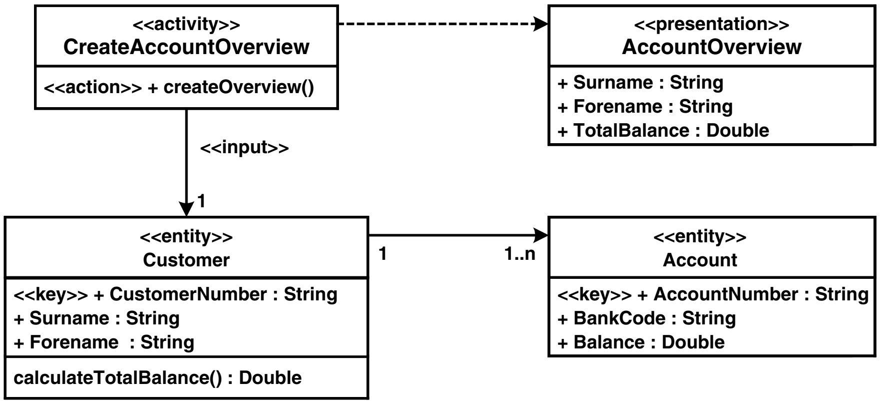


Figure 2.9 An example of architecture-centric design

The domain-related meaning of the diagram in Figure 2.9 is fairly obvious: at its core is an activity, a module for superordinate process models that is able to carry out an action for the creation of a customer-specific account overview. The customer entity serves as input, which is transmitted to the activity. Besides two domain-related attributes, the customer entity possesses an identifying characteristic (a key) and is able to calculate the total balance by adding balances of the associated accounts. The activity, or respectively its action, uses a presentation with three domain-related attributes to display the result.

A standard Java code generator would ignore the annotated stereotypes and generate the signatures of four simple Java classes. In AC-MDSD, the realization of the model on the programming language side is realized by a mapping to a concrete platform. This is illustrated by the two examples that follow.

For an EJB-Based Architecture with HTML Clients

Activity classes are stateless session Beans that implement the interfaces of a server-side process engine. Each action is declaratively transactional. The *entity* classes are Beans with corresponding local interfaces. Attributes of the type *key* constitute the primary key classes. For public attributes, getter and setter methods are applied. Container Managed Persistence (CMP) is used for persistence. The necessary descriptors can be deduced from the model. For associations, access methods are available that are based on the associated model's finder methods. The *presentation* classes specify JSP models that serve to fill JSP/HTML pages. The presentation implementations are activated by a *FrontController* framework.

For a C++/CORBA-Based Client-Server Architecture

For each *activity* class there is an IDL interface. All attribute and parameter types of the design are mapped to corresponding IDL types. A suitable C++ skeleton exists. The activity classes implement the interfaces to a specific workflow system. Actions (action operations) are transactions on an Object Transaction Monitor (OTM). All entity classes are non-distributable C++ classes: their instances are submitted to a RDBMS via object-relational mapping. Attributes of the type *key* serve as primary keys. The presentation classes are Java Swing GUIs that implement the interfaces of a specific client-framework.

By means of this simple example of a model we can easily recognize the main advantages of this approach: architecture-centric models are compact, sufficiently rich in information and do not contain any superfluous details that would impede portability and lower their degree of abstraction. They are therefore more concise and easier to maintain. Moreover, they are better suited for enabling discussions with other project members, as they are not polluted with technical details.

2.5.4 Development Process

Generative software architectures and architecture-centric design can only be applied effectively when the development methodology is adequately adapted. This extremely important issue is not in the focus of the MDA's attention. We dedicate the whole of Chapter 13 to this issue, which illuminates MDSD from a process point of view. Since we are dealing with the special case of architecture-centric design here, preparing the foundations for the following case study, we highlight only a few aspects here.

Separation Between Architecture Development and Application Development

We have already seen that a generative software architecture leads to a modularization of application development: UML profile, generator templates, and infrastructure components on one hand, architecture-centric design, generated infrastructure code, and manually-implemented code on the other.

Quite clearly, the applications depend on the generative software architecture, but not vice versa. This leads us to the consideration of splitting the creation of these artifacts into two separate paths: as in framework development, one team can handle the creation of the generative software architecture (the *architecture development track*) while another team deals with application development (the *application development track*). The dependencies must be alleviated by a suitable synchronization of the iterations, or through release management – more about this topic can be found in Chapter 13. Regardless of the question of whether one wants to assign different *people* to the two paths or not, we are obviously dealing with substantially different activities here, so that a role-oriented view makes sense:

- Architects develop the generative software architecture.
- Designers create the application's architecture-centric model.
- Developers program the application logic and integrate it in the generated infrastructure code.

The Importance of the Reference Implementation

A practical generative software architecture is not realized out of the blue – a blueprint is needed for the code to be generated. This blueprint is called a *reference implementation*. We are referring to a runnable sample that is as concise as possible with respect to actual domain functionality, but which shows the semantics of the architecture-centric UML profile constructs on the source code level. In the next step, generator templates can be derived from such a reference implementation. We will concretize these in the course of a case study, as well as discussing them in greater detail in Chapter 13.

2.5.5 The Properties of Architecture-Centric MDSD

Before we get started with the case study in the next chapter, we'll briefly summarize the properties of architecture-centric MDSD. Methodological aspects come to the fore here: AC-MDSD supports individual architectural requirements. Its focus is clearly the engineering principle and not the integrated development environment (CASE or MDA tool/IDE). In other words, nothing will be generated that hasn't been verified before via a reference implementation. Therefore, we can skip questions that often emerge in the context of generative approaches, such as "How good is the runtime performance of the generated code?" or "How good is the quality of the generated source code?" The generated code is as good (or as bad) as the reference implementation from which the generator templates are derived.

- *Software system families instead of unique items*. AC-MDSD not only aims at increasing efficiency and quality when developing one-off applications, it also aims at the reuse of generative software architectures for architecturally-similar applications that therefore constitute software system families. This aspect is not explicitly a main concern of the MDA.
- *Architecture-centric design*. Other than the MDA, we (usually) work without platform-specific models. Instead we apply platform-independent models in architecture-centric design. This approach, which on one hand poses a limitation, clearly leads to optimization on the other. The maintenance effort for intermediate results is reduced and consistency problems are avoided.
- *Forward engineering*. Contrary to the MDA vision, we deliberately avoid round-trip engineering. Since architecture-centric MDSD models require real abstractions, reverse

engineering is either not possible, or does not make sense. Design changes have to be made to the actual design – that is, the model. Thus the model will always be consistent with the generated source code.

- *Model-to-model transformation for modularization only*. We use a PIM that is as abstract as possible, but ideally is directly (and of course iteratively) transformable into source code. The 'transformation gap' can be modularized via model-to-model transformations, but intermediate models occurring en route are implementation details that are invisible to the application developer.
- *Source code generation without explicit use of the target metamodel*. The generation of programming language source code is essential for AC-MDSD (Chapter 9). However, we believe that model transformations as they are currently being discussed in the context of the MDA standardization are only helpful for model-to-model transformations. The generation of architecturally-motivated infrastructure source code in this manner is very cumbersome, whereas the use of generator templates is proven and can be handled very intuitively. The source metamodel (that is, that of the design language) is, with the exception of the target metamodel, very useful for the generation of source code in order to structure the transformation rules, as our case study will demonstrate.
- *No 100% generation*. As a rule, 'only' 60% to 80% of software is generated from architecture-centric models. We think that 100% generation is possible, and wise, in only very few exceptional cases[16]. Architectural infrastructure code of an application is 100% generated, but the individual/domain-related aspects are supplemented in the target language.
- *Software architecture becomes manageable*. Generative software architecture is per se formal and up-to-date. The developers cannot leave the frame of the infrastructure code that has been set, either accidentally or on purpose. This is clearly an advantage as far as quality is concerned. Developers and designers can immediately detect all changes in the architecture and can handle them in the right place – that is, centrally in the generative software architecture, instead of distributed all over the application. Technical and domain-related aspects are clearly separated. Therefore AC-MDSD makes sure the architecture is really used consistently in an application, and helps to realize architectural changes that cut across the system. This again supports the scalability of the development process. In other words, AC-MDSD is a very useful and powerful instrument for software architecture management.

So where do we go from here? After you have established a stable AC-MDSD infrastructure, it is often useful to cascade additional MDSD-layers on top of it. This approach, called cascaded MDSD, is explained in Section 8.2.8.

[16] This statement is valid for AC-MDSD only, not for MDSD in general.

3 Case Study: A Typical Web Application

After we have established the foundations for MDSD in general, and Architecture-Centric, Model-Driven Software Development (AC-MDSD) in particular, we can now proceed to a hands-on case study to familiarize ourselves with AC-MDSD in practice.

3.1 Application Development

First, we assume the application developer's position and presuppose the existence of a generative software architecture, as described in Section 2.5. This will typically be created iteratively and incrementally in parallel with application development. We will discuss the methodology required for this purpose in greater detail in Chapter 13.

We should mention that this view constitutes a role-based representation. However, we will not say anything about the allocation of roles to people yet, since this is a matter of project organization, which is covered in Chapter 19. Here we focus primarily on categorizing the various activities to help an understanding of the subject matter. Based on an example application, we explain the most important steps, then proceed to describe the relationship between application development and the generative architecture.

An iteration in application development begins with the creation or extension of an application design, in this example using a UML tool. The application design's XMI[1] representation, exported from the UML tool, is transformed into an implementation skeleton via an MDSD generator. The actual business logic is programmed manually and integrated into the generated infrastructure code. To this end, we use *protected regions*, also known as *protected areas*. Syntactically, these are comments in the target language, but are interpreted by the MDSD generator. Each protected region within the generated code possesses a globally unique identifier disguised as a comment, and is thus uniquely linked to a model element. In this way the generator can protect the contents of these regions, insofar as it re-inserts their contents at the correct locations in the generated code during iterative regeneration. This procedure is also quite robust with respect to renamings in the model, because the protected regions' IDs are generated from UUIDs[2] of the model (more precisely, from the XMI format), rather than from names of model elements such

[1] XMI: XML Metadata Interchange. A MOF–XML mapping that is used mostly to serialize UML models in XML form. Almost all UML tools support XMI, which is an interoperable export format.

[2] Universal unique identifiers for model elements.

as class names or something similar. Using this approach, protected region contents will also survive renamings in the model. The content of its protected regions will be deleted only if you delete a model element.

Protected regions are not always the best means for integrating generated and manually programmed code, but this is not our concern yet.

3.1.1 The Application Example

The following example has been taken from an MDA/MDSD tutorial, and has been presented as a 'hands-on' session with great success at various conferences (JAX, OOP, and others). The application was created to illustrate a holistic software development approach, ranging over business process analysis, architecture-centric design, and model-driven code generation, to the implementation of the business logic, while being based on a simple but non-trivial example. The analysis model was taken from tutorial material from oose.de GmbH, and the generative software architecture was built by b+m Informatik AG.

The example describes the development of an information system for a car-sharing company. Figure 3.1 shows the use-case overview of the fictitious application 'Car-Sharing 1.0'.

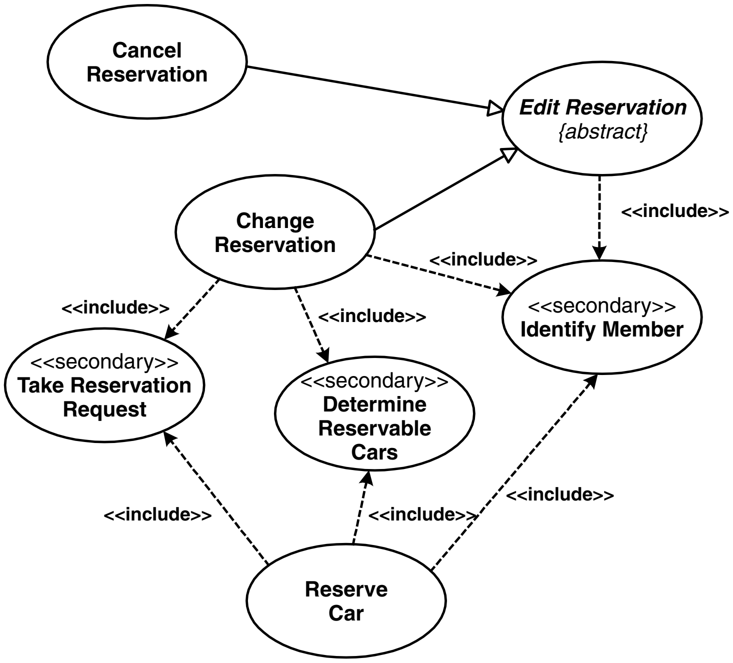

Cancel Reservation
Edit Reservation {abstract}
<<include>>
Change Reservation
<<include>>
<<include>>
<<include>>
<<secondary>> Identify Member
<<secondary>> Take Reservation Request
<<secondary>> Determine Reservable Cars
<<include>>
<<include>>
<<include>>
Reserve Car

Figure 3.1 Use case overview of the car-sharing application

Car-Sharing Version 1.0 implements the system use case 'Make car reservation' and allows car reservations as well as car management. The members of the car-sharing community are registered in the system for authorization and – later – billing purposes. The main purpose of the system is the electronic execution of car reservations through call-center agents.

The architecture of the car-sharing application is a classic three-tier architecture, consisting of a presentation layer, a process layer, and a persistence layer – see Figure 3.2. It is based on the J2EE framework.

The presentation layer uses the MVC pattern based on the Servlet Model 2 architecture à la Struts [STRT]. All HTTP requests from the browser are intercepted centrally by a *FrontController*, then evaluated and dispatched to the respective view to be displayed. The *FrontController* delegates the evaluation and processing of triggered GUI actions, as well as the evaluation of guards in the navigation sequence to the according *SubController*. The *SubController* provides the data required by a *View*'s display in the shape of a *ViewModel.* For the purposes of flow control, this layer uses the Struts framework. Data exchange with the process layer takes place via *ValueObjects*. The process layer offers the presentation layer's controllers stateless transactional services in the form of methods on *ProcessObjects*. These process objects allow controllers to read the view-relevant data and to store newly-received data. At the same time, entities located in the persistence layer are protected from being directly accessed by objects in the presentation layer.

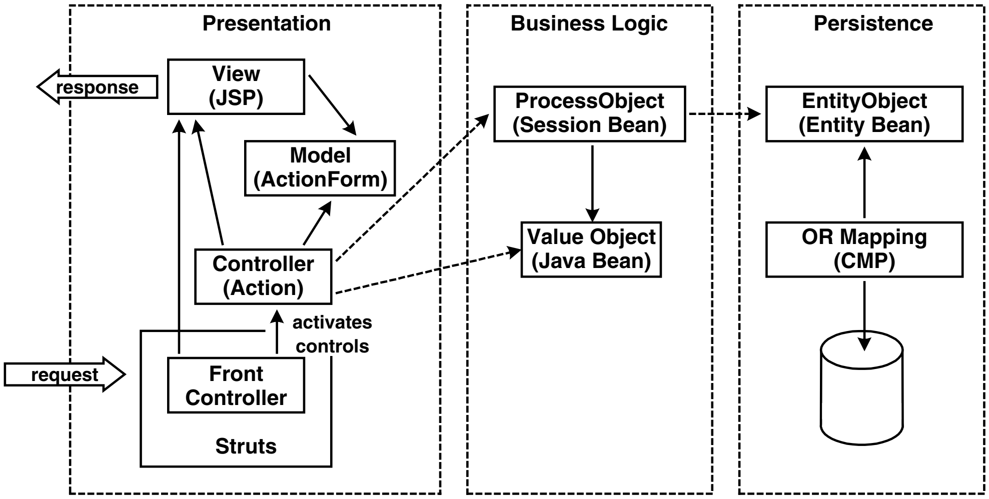


Figure 3.2 Car sharing architecture

The persistence layer possesses a persistent business object model (BOM) that is implemented using Java Entity Beans. Persistence is handled by the CMP mechanism (Container Managed Persistence) in combination with an SQL database. The target platform and runtime environment of the application are built exclusively from Open Source software: a Tomcat Web server [TOMC], the EJB 2.0-compliant application server JBoss [JBOS], and the HyperSonic SQL [HSQL] database. The runtime environment's central element is the Struts framework, which controls the application's processes. In addition, the runtime environment is completed by some super and helper classes.

The creation of the car-sharing application's design in the form of a PIM is conducted with the help of a design language (UML profile) which describes architecture concepts. In the UML profile we can find the concepts that were laid out in the conceptual architecture overview in a simplified form (for example *EntityObject*, *ValueObject* and so on). We discuss the exact profile definition later. Transformation to the target platform (the platform binding) is achieved via a set of generator templates that generate the required source code from the model information. The design language and the platform binding, in the form of templates, make up the generative software architecture (Section 2.5), which can be seen in Figure 3.3.

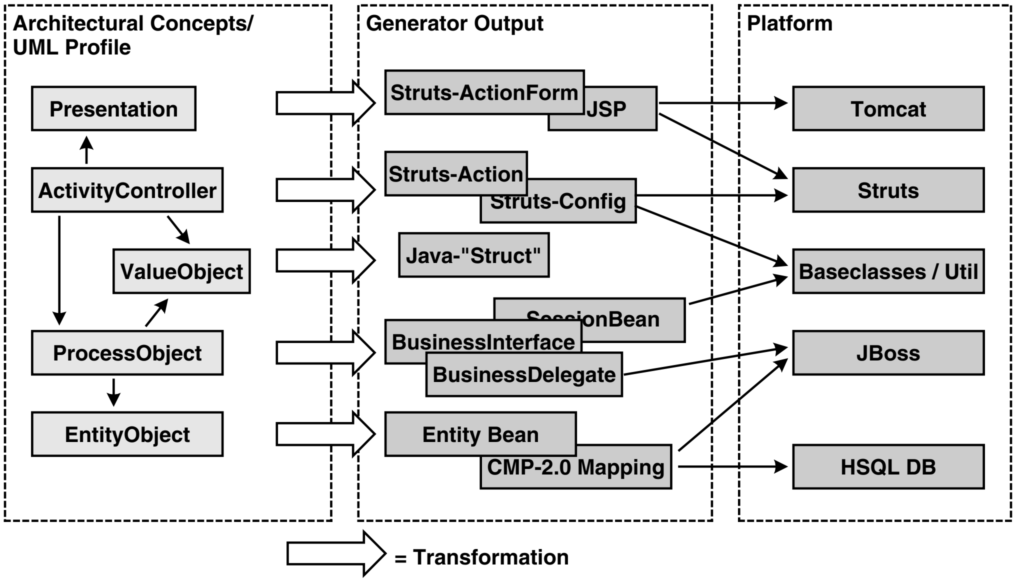


Figure 3.3 Generative software architecture and runtime environment

3.1.2 MDSD Tools

To apply AC-MDSD in practice we need a UML modeling tool and an MDSD generator. The UML tool must be able work with UML profiles for UML language extensions. At present, no mainstream UML tool exists that is able to evaluate modeling constraints, that is, able to check the assertions made on the metalevel in the form of OCL expressions[3]. Checking constraints therefore needs to be supported by the MDSD generator.

The generator tool must read the models provided by the respective UML tool and use them as input for generation. Today, most UML tools are able to save models in XMI format, but XMI quality varies. Thus the MDSD generator should offer predefined and customizable adapters for different modeling tools.

A more detailed discussion of tools and requirements can be found in Chapter 11.

[3] OCL: Object Constraint Language, part of UML.

In our scenario, we use the UML tool Poseidon UML Community Edition from Gentleware [POSE], whose XMI output is transformed into source code by the openArchitectureWare generator framework [OAW] via generator templates. This source code is then further modified in the Eclipse IDE [ECLI]. The generator framework is supplemented with an Eclipse plug-in, so that its use in an integrated development environment is feasible. It also meets the requirements stated above.

The following examples outline the developer's activities at the various levels of application programming that occur in the course of the design/generate/build cycle.

3.1.3 Example 1: Simple Changes to Models

The first example describes a simple change to the static class model of the car-sharing application and the execution of one cycle of design, generate and build. Since the JSPs required for the car-sharing application are completely generated from the information in the class that has the *«Presentation»* stereotype, it is advisable to change something in the presentation layer (see Figure 3.4).

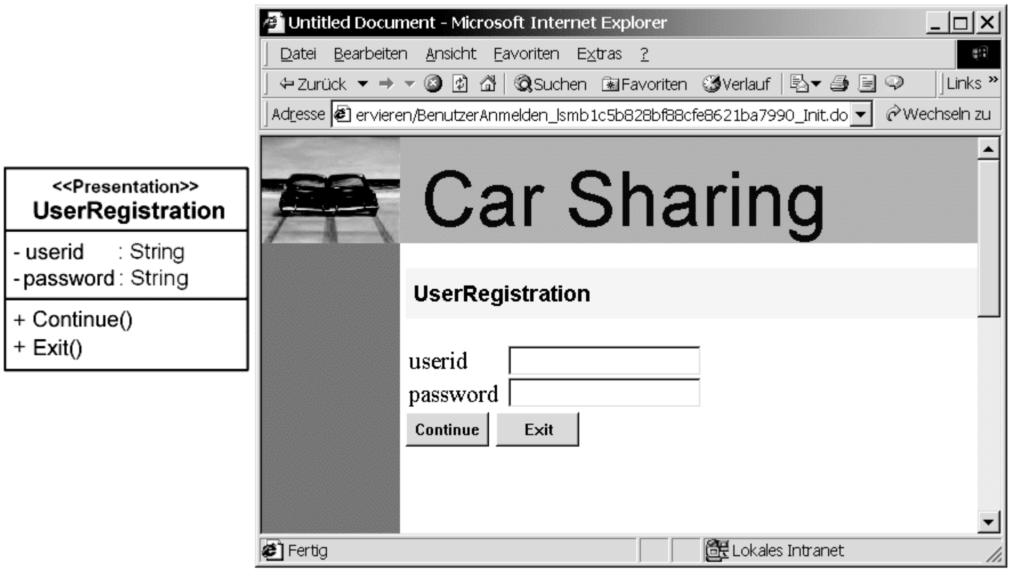


Figure 3.4 Transformation of *UserRegistration* model to the concrete dialog *UserRegistrationView*

The left of Figure 3.4 shows the presentation class *UserRegistration*, and the right-hand side shows the dialog (JSP) that is generated from it, after HTML rendering. The methods of the *Presentation* class have as their counterparts the *Continue* buttons in the browser's JSP presentation. The renaming of the method *Finish* as *Exit* in the *UserRegistration* class results in a change of the respective button's label, as you can see from the dialog. Besides the JSP, the Struts *ActionForm*, which constitutes a *View Model*, is completely generated from the presentation class. Both artifacts are the results of transformations.

The following code shows the generated JSP *UserRegistration.jsp.*

```
...
<html:form action="<%= (String) request.getAttribute("FormAction") %>"
method="Post">
  <table border="0" cellspacing="0" cellpadding="0" >
  <tr>
    <td><bean:message
key="de.amg.carsharing.user.presentation.UserRegistration.userid"/>&nbsp</td>
    <td>
      <html:text property="userid"/>
    </td>
    </tr>
  <tr>
    <td><bean:message
key="de.amg.carsharing.user.presentation.UserRegistration.password"/>&nbsp</
td>
    <td>
      <html:password property="password"/>
    </td>
  </tr>
  <tr>
    <td>
      <input type="hidden" name="registration.jsp.Event" value="Continue">
      <input type="submit" name="Event" value="Continue"/>
    </td>
    <td>
      <input type="hidden" name="registration.jsp.Event" value="Exit">
      <input type="submit" name="Event" value="Exit"/>
    </td>
  </tr>
</table>

</html:form>
...
```

The following listing shows the form class behind it:

```
package de.amg.carsharing.user.presentation;

import org.apache.struts.action.ActionForm;
public class UserRegistrationForm extends ActionForm
{
  private String userid;
  private String password;
  public String getUserId()
{
    return userid;
}
public void setUserId(String aUserId)
{
    userid = aUserId;
}
```

```
   public String getPassword()
   {
    return password;
   }
   public void setPassword(String aPassword)
   {
    password = aPassword;
   }
  }
```

Where simple changes during development are concerned, we work exclusively at the model level. After such changes have been made, the model is exported to XMI format. The generator interprets the XMI and generates the corresponding sources. Building as well as deployment both take place in the IDE or via Ant [ANT].

It is clear that the model here takes the place of source code. – all information regarding the change is kept in the model. It is therefore advisable to integrate the model into the application's release management in addition to the actual sources.

3.1.4 Example 2: Model Changes and Protected Regions

The second example illustrates how individual parts of business logic in protected regions can be supplied. For this purpose, we look at how a parameter that is required for making reservations is determined in the process layer and made available to the presentation layer.

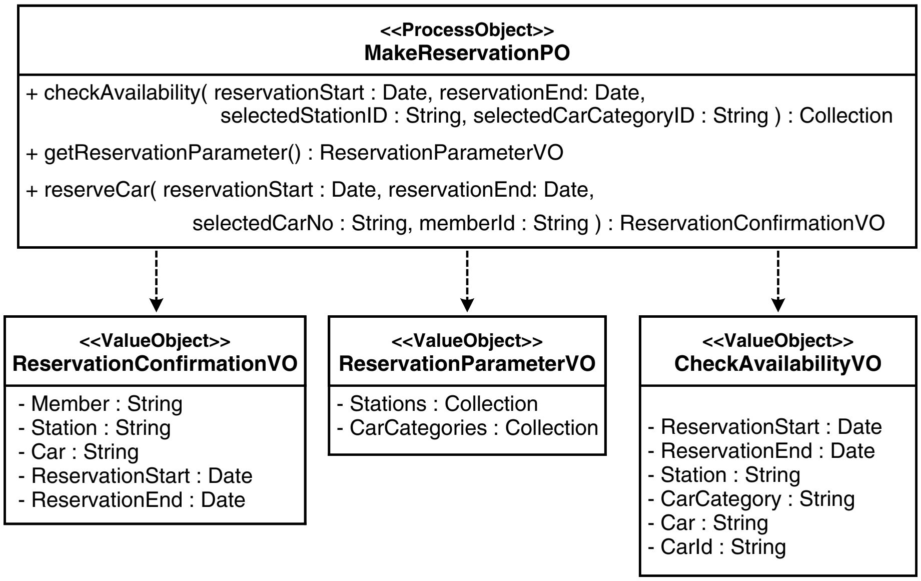

<<ProcessObject>>
MakeReservationPO
+ checkAvailability(reservationStart : Date, reservationEnd: Date, selectedStationID : String, selectedCarCategoryID : String) : Collection
+ getReservationParameter() : ReservationParameterVO
+ reserveCar(reservationStart : Date, reservationEnd: Date, selectedCarNo : String, memberId : String) : ReservationConfirmationVO
<<ValueObject>>
ReservationConfirmationVO
- Member : String
- Station : String
- Car : String
- ReservationStart : Date
- ReservationEnd : Date
<<ValueObject>>
ReservationParameterVO
- Stations : Collection
- CarCategories : Collection
<<ValueObject>>
CheckAvailabilityVO
- ReservationStart : Date
- ReservationEnd : Date
- Station : String
- CarCategory : String
- Car : String
- CarId : String

Figure 3.5 MakeReservation process view

Figure 3.5 shows the part of the model that is needed in the process layer. Here, the *MakeReservationPO* gets the *getReservationParameter()* method, which returns *ReservationParameterVO*.

The *ReservationParameterVO* serves as a data container to enable passing of data from the process layer to the presentation layer. During generation, all classes, Java interfaces and deployment descriptors necessary for the execution of *MakeReservationPO* in a Session Bean are generated from this model. Additionally, a *MakeReservationBusinessDelegate* is generated that is based on the business delegate pattern from the J2EE core patterns [CORE]. The *ReservationParameterVO* is 100% generated, whereas for the method *getReservationParameter()* only the method signature is generated. The implementation must be added manually in the IDE. This is done in protected regions, as the following code excerpt illustrates:

```
public ReservationParameterValueObject
       getReservationParameter()
    throws RemoteException {
    // PROTECTED REGION ID(12Operation_MethodBody) START
    ReservationParameterValueObject vo = null;
    try
    {
      CarSharingModuleComponent component =
             new CarSharingModuleComponentImpl();
      StationHome home = component.getStationHome();
      Collection stations = home.findByAll();
      Collection colStations = new ArrayList();
      for (Iterator i = stations.iterator();
                        i.hasNext(); ) {
        Station station = (Station) i.next();
        colStations.add(station.getName());
      }

      Collection colCarCategories = new ArrayList();

      colCarCategories.add(CarCategory.COMPACT);
      colCarCategories.add(CarCategory.VAN);
      colCarCategories.add(CarCategory.SPORT);
      colCarCategories.add(CarCategory.LUXURY);

      vo = new ReservationParameterValueObject(
               colStations, colCarCategories);
    }
    catch (Exception e) {
      e.printStackTrace();
      throw new RemoteException("Error: "+
            "Registration parameter search failed", e);
    }

    return vo;

    // PROTECTED REGION END
}
```

The decision as to whether protected code regions are required or not must be made at the architecture level when the generator templates are created (see Section 3.2). Additions or changes outside these protected areas are not allowed, because they would undermine the clear separation between modeling and programming on one hand, and between application and architecture modeling on the other: design changes must be made in the design (the application model) and architectural changes – that is, systematic changes to the generated code – must be made in the generative architecture. The generator framework ensures this: changes that are made outside protected regions will get lost in the course of iterative regeneration. This is not intended to restrict the freedom of application developers, but guarantees consistency, as well as regular communication between application and architecture development. The definition of this boundary between generated and non-generated code is pivotal and its handling requires some experience. This subject is described in more detail in Chapter 7.

3.1.5 Example 3: Working with Dynamic Models

Besides the options for generation of source code based on static models, as shown in the previous examples, dynamic models such as activity diagrams and state diagrams can also be used for code generation. This example describes how this can work. Figure 3.6 shows an activity

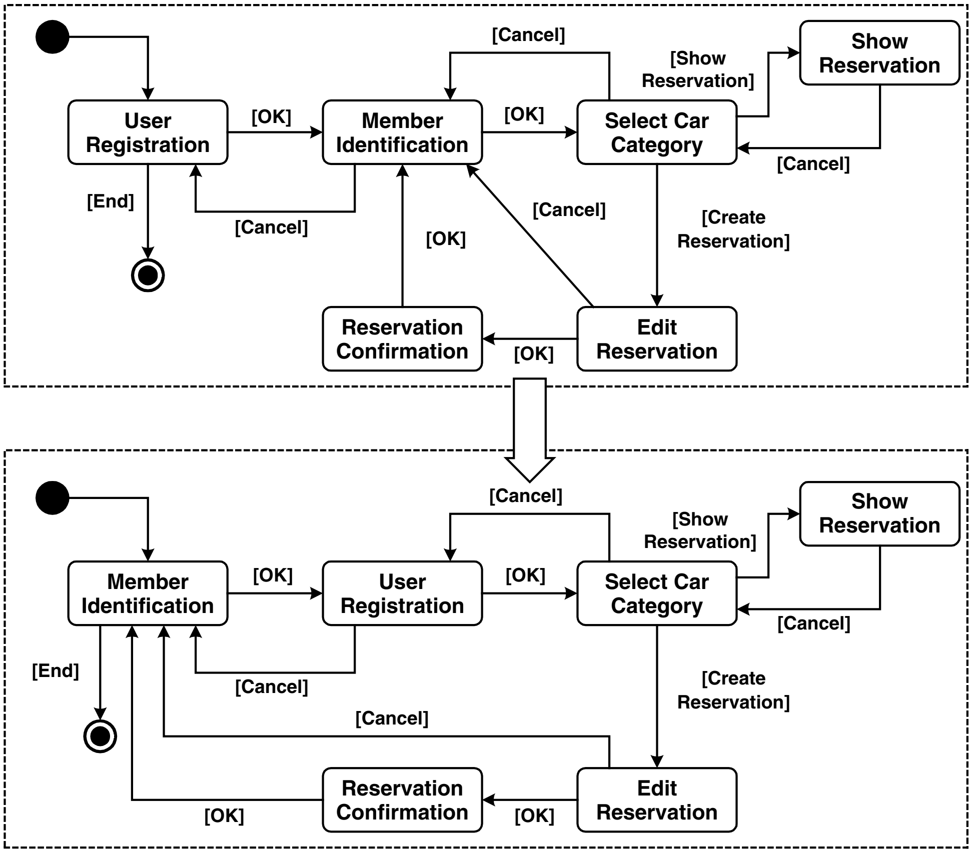


Figure 3.6 Change in the navigation order

diagram before and after a change in the navigation order of the resulting application. In the new version, the step leading to the identification of the calling member must be carried out before a user registration can take place. (Whether this makes sense or not is open to dispute.)

Since we chose Struts as the control flow framework for our example, the necessary flow control configurations must be generated based on the activity diagram. The following excerpt from the Struts configuration shows what this looks like:

```
<!-- ControllerState "UserRegistration" -->
<action
 path="/ UserRegistration_a64aa2a7d0162ba7ffb_Init"
 type="de.amg.carsharing.user.
              presentation.UserRegistrationController"
 name="UserRegistrationForm"
 input="/UserRegistration.jsp"
 scope="request"
 parameter="UserRegistration_Init,a6488aa27d162ba7ffb">
   <forward name="Ok"
            path="/UserRegistration.jsp"
            contextRelative="true"
   />
</action>

<action
 path="/UserRegistration_a64aa2a7d0162ba7ffb_Exit"
 type="de.amg.carsharing.user.
              presentation.UserRegistrationController"
 name="UserRegistrationForm"
 input="/UserRegistration.jsp"
 scope="request"
 parameter="UserRegistration_Exit, a6488aa27d162ba7ffb ">
   <forward name="UserRegistration_To_Exit"
            path="/MemberIdInput_a6affa_Init.do"
    />
   <forward name="Continue"
            path="/SelectCategory_a64aac9_Init.do"
   />
   <forward name="Error"
            path="/UserRegistration_a64aa30f0fbfb_Init.do"
   />
</action>
```

The following XML fragment is an extract from the corresponding Struts *config.xml*, which controls the acquisition of the member ID[4].

[4] The *config.xml* file is used to configure pages and page flow in the Struts framework.

```
<!-- ControllerState "MemberIdInput" -->
<action
 path="/MemberIdInput_a64aa3a062ba7ffa_Init"
 type="de.amg.carsharing.member.
       presentation.MemberIdentificationController"
 name="MemberIdentificationForm"
 input="/MemberIdentification.jsp"
 scope="request"
 parameter="MemberIdInput_Init,a60f06b7f">
   <forward name="Ok"
            path="/MemberIdentification.jsp"
            contextRelative="true"
   />
</action>
```

The control flow can be generated completely from the application design. Further manual manipulations of the Struts configuration are not required. Our experience shows that this is particularly advantageous, because activity diagrams also document the navigation extremely well and can thus be used for discussion with domain experts.

3.1.6 Interaction Between Development and Architecture

Good coordination between application development and architecture is the key to success in MDSD projects. Development of the generative architecture is not finished when the generative software architecture has been delivered. As the application development steps described in our examples show, in most cases requirements for change of the generative software architecture will emerge in the course of the project. On one hand, additional protected code regions are needed to allow individual adaptations. On the other hand, new architecture patterns are identified that must be generatively supported. The team will arrive at an optimal, sustainable solution that meets the requirements only if it manages to accept and incorporate feedback from application development into the generative software architecture. In this respect, the generative software architecture evolves like a framework. Similarly, it must be versioned and made accessible to the projects that use it. We will take a closer look at these topics in the third part of this book.

To examine your own ideas for improvement of the generative software architecture in use, you can test it locally in a ‘sandbox’. The code outside the protected regions will remain unchanged until a regeneration occurs. If the change yields the desired result, it can be made accessible to the entire project through an adaptation of the generative software architecture. The advantage of such an approach is that the generative software architecture is always available in a well-defined and consistent state.

3.1.7 Intermediate Result

When you assume the role of the developer you gain more time to deal with essential tasks – the realization of the project-specific business logic. Tedious copy and paste work for the development of technical infrastructure code that is totally meaningless for business-related programming is taken on by a MDSD generator or the generative software architecture. Correction of errors in technical code is much easier and can be carried out more efficiently compared to non-generative approaches. A bug in the infrastructure code must only be fixed in one place, in the transformation rule of the generative software architecture, similarly to bug-fixing in a framework. After regeneration, all flawed code fragments are replaced with corrected ones.

Because we integrate manually-written business logic code into the generated skeleton, we however lose our application's complete platform independence and automatic portability. Quite clearly, the contents of the protected regions possess dependencies on the programming language Java and the Struts framework. There are patterns (such as *BusinessDelegate*, which has been used here) that help to reduce these dependencies, yet we are nowhere near a realization of the OMG's vision of executable models. Here, the different goals become clear (see Section 2.5): AC-MDSD is pragmatic and emphasizes the enhancement of development efficiency, quality, maintainability and reusability, while MDA emphasizes portability and interoperability.

3.2 Architecture Development

The previous section examined AC-MDSD from the application development perspective. In this context, we assumed the existence of a UML profile (design language), a platform (J2EE, Struts, persistence layer and so on) as well as a generative software architecture that works for us. Now we will look at how these artifacts may appear in detail.

One of the key concepts is redundancy avoidance. Redundancy (artifacts that occur multiple times in different instantiations) cannot be found only on the EJB level, but also in all other layers of modern software architectures: flow control (such as Struts), presentation (such as JSPs and ActionForms), controllers, legacy integration and so on. It is our goal to delegate this redundancy as completely as possible into a generative software architecture that 'knows' all the construction principles and programming models from various layers, not just single parts or aspects. The benefit of this approach will be an enormous increase in application development productivity, as the examples in Section 3.2 demonstrate.

Such a generative software architecture goes far beyond simple generator tools such as XSLT or XDoclet and does not depend on a specific application. It is reusable and supports an entire family of software systems with the same technological properties. The car-sharing application and an insurance application could be among such families as long as they share the same underlying technological principles. In this section we are going to create a manufacturing process for applications that will allow us to automate application development to a great extent based on models. This is similar to the concept of production lines in automotive engineering. The architecture development aspect of our MDSD process deals with the creation of such software 'production lines'. This concept is elaborated in Section 13.5.

3.2.1 The UML Profile

First, we need an architecture-centric UML profile that allows us to create formal MDSD models. Let's take a look at the model in Figure 3.7.

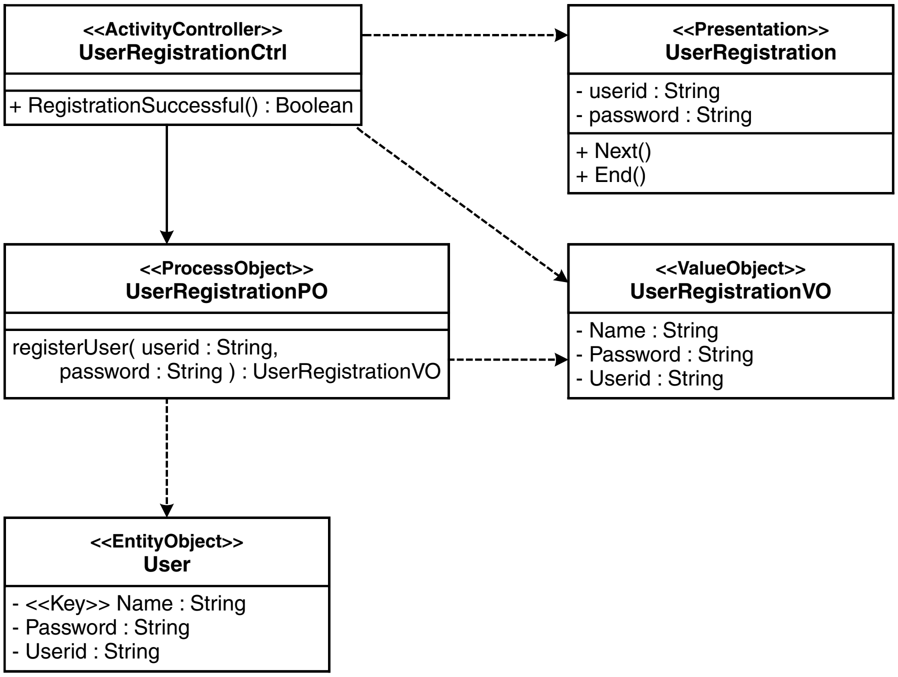


Figure 3.7 Architecture-centric design for *UserRegistration*

The domain-related meaning of this model is clear: The class *UserRegistrationCtrl*, being one step in the control flow, is able to activate a presentation in which a user's ID and password can be entered and committed to the system. For authentication against the system, the controller uses the service *userRegistration()* of *UserRegistrationPO*.

The *ProcessObject* gets the entity representing the user based on the user ID, which constitutes the user's identifying characteristic, and validates the supplied password. If registration is successful, the service issues the controller a *UserRegistrationVO* with the respective access data.

For modeling purposes, a design language is used that captures the architectural concepts used in the application. The focus of this design language is on architectural reusability, and results in an architecture-centric design, completely abstracting from technological details. The model can be transformed into source code for various target platforms via suitable rules.

We have already familiarized ourselves with the most important elements of the UML profile in the left column of Figure 3.3. The transformations in the generator are used to achieve the binding to the platform used in the car-sharing application, shown in the right column of Figure 3.3.

Figure 3.8 shows a section of the formal UML profile definition:

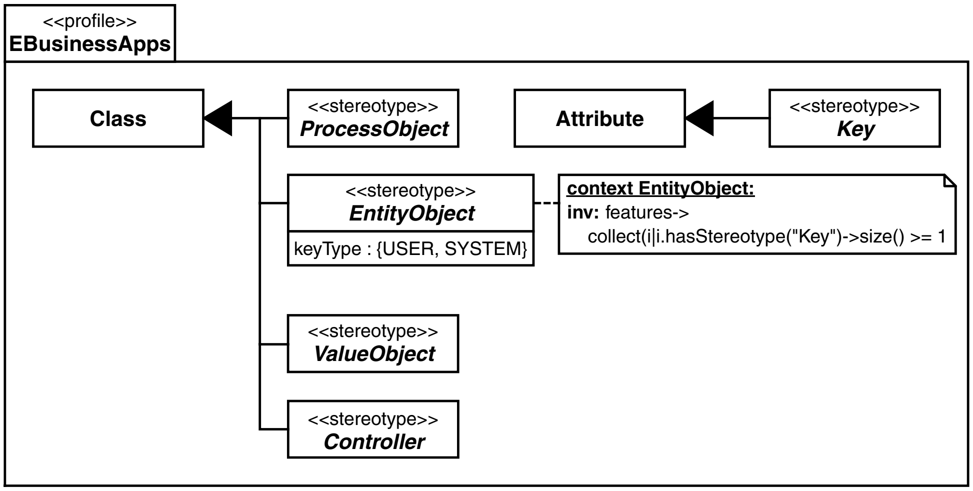


Figure 3.8 UML profile definition for the design language

This profile represents a specialization of the standard UML metamodel (Chapter 6) on the class, attribute, and operation level, leading to a special language profile for the specific requirements found in the car-sharing's application three-tier architecture.

The UML extension mechanism's stereotype and tagged value are used here. For example, the classes that are assigned the stereotype *«Presentation»*, such as *UserRegistration*, are responsible for the presentation and input of data. Classes labeled *«EntityObject»*, such as *User*, constitute the application's persistent business data types, and offer mechanisms for identification and querying. Design constraints (modeling rules) are also an important part of the profile, and can be formulated with the help of the Object Constraint Language.

Ideally, the profile definition, including the constraints, would take place in the UML tool and also be interpreted by it, so that exactly those tagged values defined for a certain stereotype are allowed, and the modeling rules (constraints) of the profile are checked. Unfortunately, many commonly-used UML tools do not yet possess these features, so that the formal UML profile definition still has the character of documentation.

3.2.2 Transformations

After we have defined a UML profile, we can now tackle the actual code generation. This is not a simple task. Fortunately, some partial tasks are of a more general nature, and there are MDSD tools that will do the work for us. This includes the neutralization of the UML tool's XMI output, template expansion control, input/output stream handling, and scanning and persistence of protected code regions for business logic implementation. Most MDSD generators are frameworks and use a template-like language to describe transformations from model to code in a straightforward way.

Using XSLT in this context, one very soon realizes its limitations, particularly when the style sheet directly transforms the UML tool's XMI output. Here, the XMI structures are so deeply entangled and indirectly referenced that XSLT stylesheets very soon become incomprehensible and therefore unmaintainable. The power of a programming language is also missing, and even protected code segments are difficult to realize and unwieldy. Sadly, no standard for MDA (or MDSD) generators or transformations is currently in existence, so the market offers many different approaches with various features. In the following sections we use the Open Source generator framework openArchitectureWare and the car-sharing example to demonstrate how metaprogramming in the context of the generative software architecture works in detail.

Metamodel/Profile Implementation

To enable the generator framework to generate the implementation skeleton for the target platform, it requires a Java implementation of the applied UML profile (see Section 3.2.3). The openArchitectureWare framework makes dealing with this task much easier, as it features a Java implementation of the UML class together with an activity core metamodel that can be specialized via Java inheritance. Hence we must create a Java class of the same name for each stereotype in the profile, which in turn must inherit from the metaclass to which the respective stereotype will be applied, that which is specified in the UML profile as a scope for the stereotype. For example:

```
public class EntityObject extends Class
{}
public class Key extends Attribute
{}
```

Class and *Attribute* are not classes from the *java.lang.reflect* package, but – as already stated – metaclasses supplied by the openArchitectureWare framework.

The clue here is that the generator framework can instantiate this specialized metamodel – at the beginning of a generator run, it creates an instance of the metaclass *EntityObject* for each model element that has the *«EntityObject»* stereotype. Each single element of the input design model (classes, associations, attributes, operations, parameter, activities, transitions) is exactly represented by a Java object of the respective type in the generator's JVM. Non-stereotyped elements will, of course, lead to the instantiation of the corresponding core metaclass. As a consequence, the metamodel implementation will be instantiated, that is, a design model will be transformed into a Java object graph that is ready to be accessed by the generator templates.

Besides the representation of stereotypes, the specialized classes have other important tasks in openArchitectureWare:

- *Tagged values*. A stereotype-specific tagged value in the UML profile is simply mapped to a string attribute of the same name of the corresponding Java metaclass.
- *Service methods for generation*. To simplify template programming and to prevent the template language from becoming a full-blown programming language, helper methods needed for code generation are programmed in Java as public methods of metaclasses. These can then be called from the templates as metaclass properties.

The following listing shows this next step[5]:

```
public class EntityObject extends Class
{
  /** set contains all Key-attributes of this EntityObject */
  protected ElementSet KeyList = null;
  /** returns set with all Key-attributes of this EntityObject */
  public ElementSet Key() throws  DesignException
  {
    if (KeyList == null) {
      KeyList = new ElementSet();
      for (int i=0; i < Attribute.size(); i++){
        if (Attribute().get(i) instanceof Key) {
          KeyList.add(Attribute().get(i));
        }
      }
    }
    return KeyList;
  }
}
```

Attribute is inherited from *Class* here and constitutes the Java representation of the meta-relationship between classes and their attributes: in other words, you can learn which attributes the designer has modeled on the current *Class* via this API (in this case even an *EntityObject*).

The template language of openArchitectureWare does not differentiate between access to attributes and access to methods of a metaclass – they are just properties. Methods hide attributes of the same name. The *Key* property defined here returns all of the *EntityObject's* attributes that have the stereotype *«key»* applied to them. This allows elegant generation of the *PrimaryKey* class in the template, for example.

If the UML tool used does not support design constraints, they can be specified in the metamodel implementation: the generator automatically calls the respective operation prior to actual generation. After instantiation of the metamodel implementation, the generator tests all constraints by calling the *CheckConstraints()* operation of all model elements. As you can see in the next listing, this is how it can be ensured that, for each *EntityObject*, at least one *Key* is defined.

```
// EntityObject.CheckConstraints() defines the
// DesignConstraints for Elements with
// stereotype <<EntityObject>>
public String CheckConstraints() throws  DesignException {
  if(Key().isEmpty())
    throw new DesignException("Constraint violation:
          "+No Key found for EntityObject '" +
          this.Name() + "'");
  return "";
}
```

[5] For common constraints, for example like those in the following listing, openArchitectureWare offers predefined helper functions. To keep our example simple, we do not use these here. The second case study in Chapter 16 illustrates this approach.

In case of a modeling error, descriptive error reports are created instead of an incomprehensible generator exception. This sort of feature is indispensable for productive use of MDSD in real-life projects.

Template Programming

The platform-specific implementation skeleton is generated by templates. Templates are very similar to generated code and can therefore be derived easily from a reference implementation. When templates are created, the constant parts of the reference implementation are copied into the template definitions as plain text and – with the aid of the template language's control structures – combined with the properties read from the metamodel. In this way, for example, all classes and descriptors of the car-sharing application's EJB EntityBeans, with their properties for deployment, persistence, and relationships, are completely generated from classes labeled *EntityObject*, except for the EQLs for business logic-related finders.

We use the template for the generation of the naming entry for an Entity Bean in the platform-specific deployment descriptor *jbossDD.xml* as a simple example here:

```
«DEFINE DeplDescr FOR EntityObject»
  «FILE FullPathName"/"Name"jbossDD.xml"»
    <entity>
      <ejb-name>«Name»EJB</ejb-name>
       «IF needsRemote»
         <jndi-name>
           «FullPackageName».«Name»RemoteHome
         </jndi-name>
       «ENDIF»
       <local-jndi-name>
         «FullPackageName».«Name»Home
       </local-jndi-name>
    </entity>
  «ENDFILE»
«ENDDEFINE»
```

The output for the *User* class from our sample is written to the file *UserjbossDD.xml* and looks like this:

```
<entity>
  <ejb-name>UserEJB</ejb-name>
  <local-jndi-name>
    de.amg.carsharing.user.entity.UserHome
  </local-jndi-name>
</entity>
```

This brief template example already hints at the simplicity and conciseness of the template language. Only the identifiers in uppercase are elements of the template language. The other identifiers inside the « » brackets are properties of the metamodel. The remainder are expanded into the target file as static text strings. The section of the metamodel implemented in Java that is relevant for this template is shown in Figure 3.9:

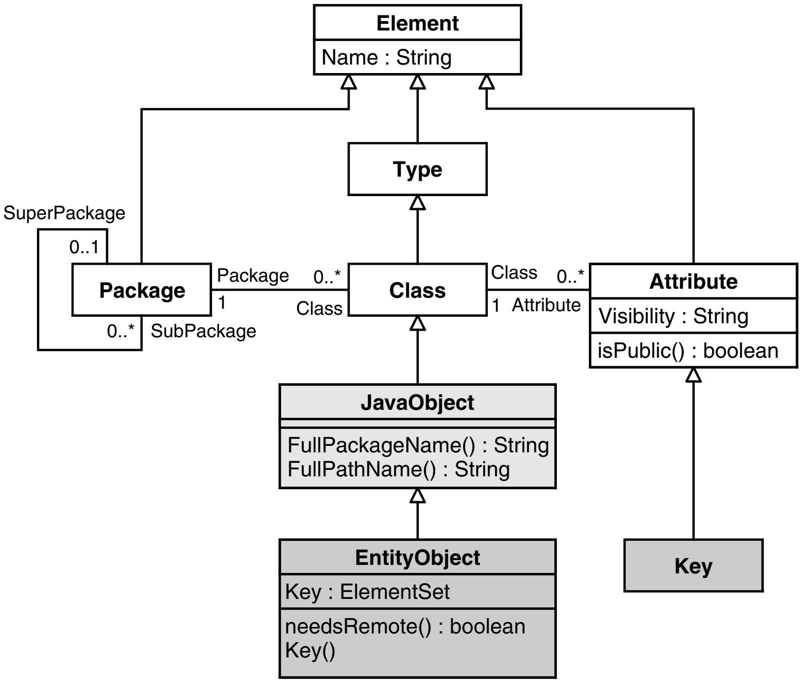


Figure 3.9 The relevant section of the metamodel implemented in Java

The class *EntityObject* corresponds with the design language's stereotype of the same name. The other classes are part of the core metamodel of class diagrams (see Section 3.2.3). The design of the example in Figure 3.7 would deliver exactly one instance of the class *EntityObject*. The entity instance with the name *User* would have three associated attribute instances with the names *Name*, *Password* and *UserID*.

The set property *Key* provides, as we have seen, a collection of key instances of the associated attributes. The Boolean property *needsRemote* lets you inquire whether the entity is callable remotely, in which case corresponding remote interfaces must be generated for the existing platform. The string properties of the super class *JavaObject*, *FullPackageName* and *FullPathName*, traverse the design's package hierarchy and return target language-conforming strings for the generation of Java import statements or file names, including their paths. Viewed from the template's perspective, these are reusable services. The new super class has been inserted because the properties not only make sense for *EntityObject*, but also for *ProcessObject*, *Value Object* and other metaclasses. However, it is abstract and therefore cannot be instantiated directly.

Even this small example proves that metamodels are actually a pivotal issue in MDSD. for this reason, we have dedicated the whole of Chapter 6 to it.

The *DeplDescr* template is defined in a special template file, a simple text file with the suffix .tpl, with the aid of a *DEFINE* block (*DEFINE … ENDDEFINE*). It relates to a class of the metamodel (EntityObject) via *FOR EntityObject*. When a property access takes place, Java-side inheritance can be used so that all properties (for example *FullPackageName*) of *JavaObject* and its super classes *Class, Type* and *Element* are at its disposal.

The *FILE* block (*FILE … ENDFILE*) enables direct expansion of templates into a file, while the file name in the example is created dynamically through a combination of string properties and string constants. In the metamodel implementation, *FullPathName* is a method, and *Name* is an attribute. This makes no difference regarding the type of access from the template's perspective.

Conditional expansion of parts of the template is supported via the *IF ... ENDIF* block. For our example this means that either a remote or a local home interface can be generated alternatively into the *jboss.xml* deployment descriptor.

A particularly useful feature of this template language is its support of polymorphism at the template level: at generator runtime, template definitions of the same name are bound via the *dynamic* type of the model element, similar to methods in Java. This is one of the most important OO concepts and serves to avoid 'type switches' (using i*nstanceof*) that are hard to maintain and often distributed all over the code.

The entire template language[6] of the openArchitectureWare framework consists of less than thirty constructs.

3.2.3 The Mode of Operation of the MDSD Generator

Figure 3.10 shows how the openArchitectureWare framework processes a generative software architecture.

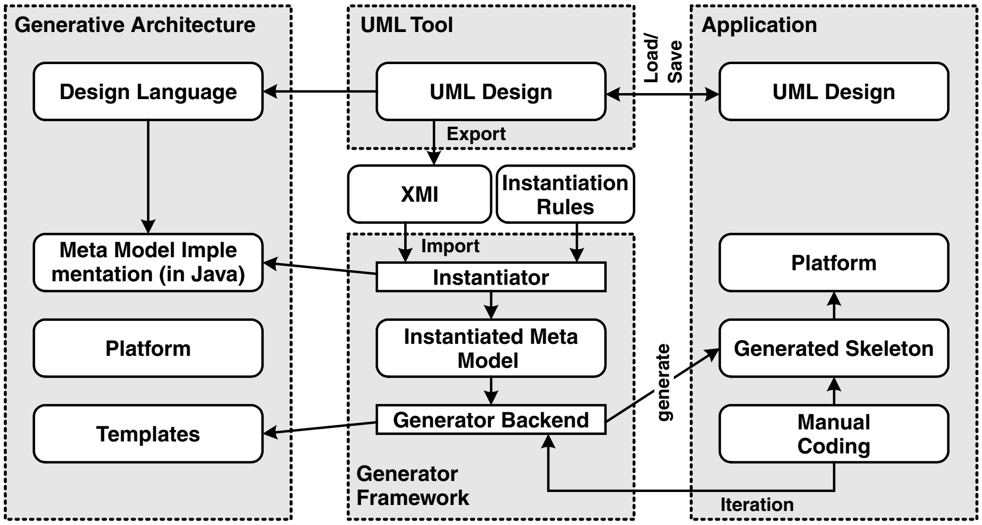


Figure 3.10 How the openArchitectureWare framework works

[6] The openArchitectureWare template language is called *XPand*.

The components and their functions are as follows:

- The generative software architecture contains all the necessary modules for use by the generator.
- *Design language*. A UML profile is often used as the design language: stereotypes, tagged values, and constraints serve to extend the standard UML with domain-specific concepts[7].
- *UML design*. The UML design is the model of a concrete application of the software system family. The design language is used for modeling.
- *XMI input*. The UML design is exported to an XMI representation via the modeling tool. The design's XMI representation can be further processed by the generator. Each model element must be assigned a UUID (universally unique ID).
- *Metamodel implementation* (in Java). The MDSD generator features a freely-configurable metamodel. This is implemented in Java, which means that precisely one Java class exists in the metamodel for each standard UML element and for each stereotype in the design language. This enables the generator framework's instantiator to use the XMI input information to instantiate the metamodel using reflection APIs. For this purpose, it uses an instantiation rule that defines which XMI element is mapped to which metamodel class. From this point on, the UML design is available as a Java object graph in the heap of the generator's JVM. The objects of this graph are simply instances of the metamodel's classes: this technique is comparable to the DOM tree that is instantiated when XML documents are parsed – compiler builders speak of an 'abstract syntax'. The instantiated metamodel constitutes the generator backend interface and shields the templates from the complex XMI structures. At the same time, it supports the Java-based development of helper methods for the generation and testing of the modeling rules of the UML profile.
- *Templates*. The openArchitectureWare framework uses a template language that, together with the metamodel implemented in Java, is an object-oriented generator: the metamodel's constructs translate themselves. The template language allows a simple and elegant formulation of the desired transformations based on the metamodel – see the examples in this section. The templates are dynamically connected with the instantiated Java metamodel via the generator backend and control the actual source code generation.
- *Generator backend*. The backend interprets the templates and does the file handling, as well as the scanning of protected regions and the preservation of contents, the existing manually-written code, in the newly generated skeleton. To ensure that nothing gets lost, for example during renaming of classes in the design, the generator uses the UUIDs in the XMI representation to identify the protected regions.
- *Instantiation rule*. This generator allows the mapping between XMI representation and metamodel to be defined in the form of a XML file. Thus XMI formats, for example from different UML tools, and the metamodel can vary independently of each other. In principle, it is even possible to process non-UML XML inputs. Due to the abstract syntax concept (the metamodel), the templates are not affected by a change of the concrete input format.
- *Runtime system*. Logically, the runtime system or the platform are part of the generative software architecture, since it does not depend on a concrete application. However, each

[7] Any other modeling languages can also be used.

generated application uses the runtime system. In other words: the generated code – method calls, *extends* or *implements* relationships and so on – depends on the platform.

A more elaborate version of this process is described in Section 11.1.2.

3.2.4 Bootstrapping

Some kind of bootstrapping process is required to create the metamodels, templates, and profiles described above initially. It is not sensible – and difficult – to begin a project with the development of the templates before a generative software architecture is present. Instead, the code to be generated later should be 'handmade' first, to act as a blueprint from which the templates will later be extracted. A runnable reference implementation provides the basis for this. Static code fragments can be transferred to the templates one for one. The variable parts of the code are worked out with the help of the template language based on the metamodel. Thus the generation of templates becomes a task that deals essentially with the elegant replacement of text, and no longer with the definition of architectural concepts. This differentiation simplifies the execution of both subtasks.

Chapter 13 details the process-related aspects of Model-Driven Software Development further.

3.2.5 Adaptations of the Generative Software Architecture

Changes and extensions to the functional requirements will be necessary in the course of an application's lifecycle, but requirements relating to architectural aspects of the application will also change, for example due to the software's use on a different application server, or migration to a new version of EJB or Struts. Whereas normally all affected classes must be manually adapted, the use of an MDSD generator allows these changes to be made in one place only. The transformation is adapted accordingly in the templates, and the new infrastructure code is regenerated. Manual adaptations are required exclusively in the protected regions of the source code, and only if the structural change affects the programming model – that is, the way in which the manually-developed code interacts with the generated code.

Even a small excerpt from the example used here demonstrates clearly how the developer's work can be simplified. If the structure of descriptors changes because of a new version of the EJB component model, or because another application server is used, only the template shown below need be adapted, rather than all of the application's **jbossDD.xml* files for all *EntityObject* classes. If we assume a migration of the JBoss container to the container of a Bea Weblogic server, for example, the *«EntityObject» User* in our example would require, among other things, the following descriptor:

```
<weblogic-enterprise-bean>
  <ejb-name>UserEJB</ejb-name>
  <entity-descriptor>
    <persistence>
      <persistence-use>
        <type-identifier>
          WebLogic_CMP_RDBMS
        </type-identifier>
        <type-version>6.0</type-version>
        <type-storage>
          META-INF/weblogic-cmp-rdbms-jar.xml
        </type-storage>
      </persistence-use>
    </persistence>
  </entity-descriptor>
  <local-jndi-name>
    de.amg.carsharing.user.entity.UserHome
  </local-jndi-name>
</weblogic-enterprise-bean>
```

To propagate these changes for all classes of type *EntityObject*, we change the template for the generation of the descriptor files as described in the following listing, then re-run the generator.

```
«DEFINE DeplDescr FOR EntityObject»
«FILE FullPathName"/"Name"weblogic-ejb-jarDD.xml"»
  <weblogic-enterprise-bean>
    <ejb-name>«Name»EJB</ejb-name>
          <entity-descriptor>
      <persistence>
        <persistence-use>
          <type-identifier>
            WebLogic_CMP_RDBMS
          </type-identifier>
          <type-version>6.0</type-version>
                      <type-storage>
            META-INF/weblogic-cmp-rdbms-jar.xml
          </type-storage>
        </persistence-use>
      </persistence>
          </entity-descriptor>
    «IF needsRemote»
      <jndi-name>
        «FullPackageName».«Name»RemoteHome
      </jndi-name>
    «ENDIF»
    <local-jndi-name>
      «FullPackageName».«Name»Home
    </local-jndi-name>
  </weblogic-enterprise-bean>
«ENDFILE»
«ENDDEFINE»
```

The architectural aspects' requirements can not only necessitate changes to existing structures, but can also require extensions. To explain the necessary steps for the expansion of a generative software architecture, a tagged value for business classes should be introduced. This tagged value should be labeled *KeyType* and can either assume the value *USER* or *SYSTEM*. Via *KeyType*, the type of the business class' unique key can be determined. In the case of *KeyType==SYSTEM* an attribute and a unique key are generated, otherwise, for *KeyType == USER*, the key is determined through identification of an attribute with the stereotype *«Key»*. In the example provided in Figure 3.11, the business class *User* possesses the *KeyType == USER* and the attribute *Name* labeled as *«Key»*.

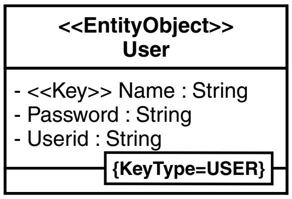
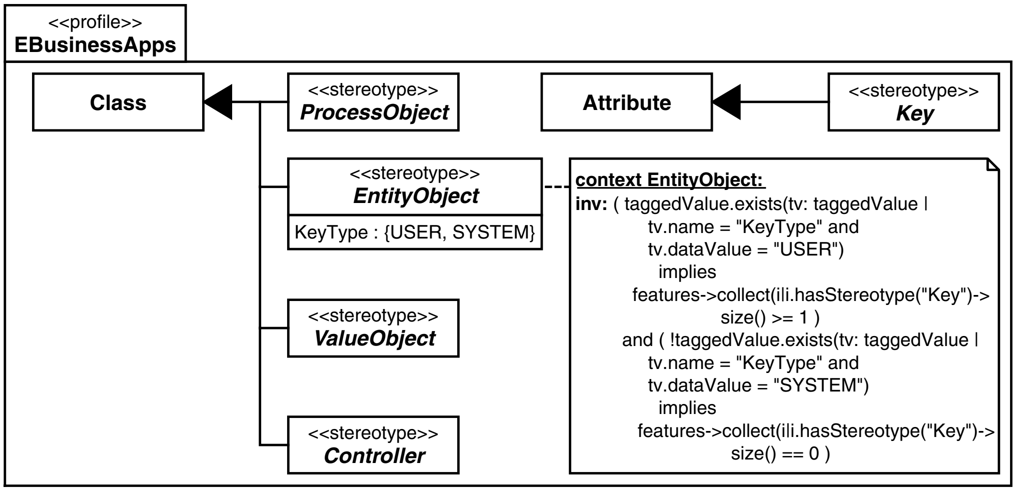


Figure 3.11 Extended EntityObject User

Due to the extension of the profile element *«EntityObject»*, the modeling constraint must be adapted accordingly: *«Key»* attributes must and may only be defined in the case of *KeyType == USER*. Figure 3.12 shows the respectively adapted formal profile definition with tagged value and OCL constraint.



```
context EntityObject:
inv: ( taggedValue.exists(tv: taggedValue |
            tv.name = "KeyType" and
            tv.dataValue = "USER")
              implies
          features->collect(ili.hasStereotype("Key")->
                    size() >= 1 )
        and ( !taggedValue.exists(tv: taggedValue |
            tv.name = "KeyType" and
            tv.dataValue = "SYSTEM")
              implies
          features->collect(ili.hasStereotype("Key")->
                    size() == 0 )
```

Figure 3.12 Profile definition with constraints

In our generative software architecture's implementation, the metamodel's tagged value is introduced into the class *EntityObject* as a property, which is set by the generator framework during instantiation of the metamodel. Thus the implementation of the *CheckConstraints()* method must be extended respectively:

```
public class EntityObject extends JavaObject {
  public String KeyType = "USER"; //TaggedValue Default
  ...
  // EntityObject.CheckConstraints()
  // defines the DesignConstraints for
  // Elements with stereotype <<EntityObject>>
  public String CheckConstraints()throws DesignException {
    if( Key().isEmpty() &&
        KeyType.equals("USER")) {
      throw new DesignException("Constraint "+
            +"violation: No Key found for "+
            +"EntityObject '" + this.Name() + "'");
    }
    return "";
  }
  ...
}
```

The higher expressive power of the UML profile is also reflected by the templates. Here, the following transformations must take place, depending on the *KeyType*, as indicated below for the Entity Bean class:

```
«IF KeyType == "SYSTEM"»
  // init-method
  private void init() {
    long time;
    time = System.currentTimeMillis();
    setImplId(String.valueOf(time) + "+" +
              System.identityHashCode(this));
  }
  public «Name»PK ejbCreate() throws CreateException {
    init();
    ...
«ELSE»«REM KeyType==”USER”»
   public «Name»PK ejbCreate(
          «EXPAND Attribute::Signature FOREACH Key
                             USING SEPARATOR ", "»)
           throws CreateException {
     «FOREACH Key AS CurKey EXPAND USING SEPARATOR "\n"»
       setImpl«CurKey»(«CurKey.asPARA»);
     «ENDFOREACH»
      ...
«ENDIF»
```

You can see what the generated Entity Bean class looks like in the implementation directly from the template. Depending on the *KeyType*, either the *«Key»* attributes will be set in the constructor, or a system-side ID is assigned.

As we have shown, changes and extensions of architectural aspects need only be made in a single place in the generative software architecture, rather than in many distributed places in the applications’ code.

3.2.6 The Boundary of Infrastructure Code

Up to this point we have shown which tasks must be taken on in the context of architecture development and how the infrastructure code is defined. But how can the manually-developed code – typically, the business logic – be implemented within this skeleton, and how can we maintain it if iterative regeneration and structural changes occur? There are various approaches to the integration of generated infrastructure code and manually-written business code, and we expand on these in Chapter 9.

The MDSD generator used in this example supports protected regions: that is, it is possible to designate certain areas of code in which developers implement the business logic. To preserve the code during regeneration, it is necessary to mark these areas as unique. To this end, the relevant protected regions are assigned unique, constant IDs from the model, the UUIDs from the UML tools' XMI output. The definition of a protected region of the template might look like this:

```
«PROTECT CSTART "//" CEND "" ID Id"Operation_MethodBody"»
  //add custom initialization here ...
«ENDPROTECT»
This leads to the following generator output:
// PROTECTED REGION ID(12aaaeOperation_MethodBody) START
  ReservationParameterValueObject vo = null;
  try {
    CarSharingAutoModuleComponent component =
        new CarSharingModuleComponentImpl();
     ...
  } catch ( … ) { … }
  return vo;
// PROTECTED REGION END
```

3.2.7 Structuring Metaprograms

The templates introduced here – together with the properties of the metamodel implemented in Java – constitute one possible implementation technique for MDSD transformations, in this case distributed across two languages. We are effectively dealing with metaprograms here, since they serve the creation of programs. It should be kept in mind, however, that metaprograms are programs too. This means that on this (meta-)level software is also created in real-life projects – software that must be structured so that it can grow iteratively and incrementally.

Here, mechanisms such as those we know from object-orientation are required. For example, construction of components is desirable. There might for example be a need to switch the component for the generation of the Entity layer to facilitate a migration from EJB 1.1 to EJB 2.0. Inheritance and polymorphism are useful allies here too. The availability of such features says a lot about how good your MDSD tool really is.

More information on this topic can be found in Chapter 11, as well as in the second case study in Chapter 16.

3.3 Conclusion and Outlook

The practicability of the OMG–MDA approach is often partially met with skepticism, which may not be totally unfounded. There are quite a few people who consider MDA to be merely a 'discipline for theorists'. However, the pragmatic version of architecture-centric MDSD introduced here has proved its practical value over many years and in projects of differing scope and size, and early adopters have come to use this approach productively.

Some think the introduction of generative approaches will limit their personal freedom, or they fear being locked in by the generator supplier. Such prejudices typically emerge due to bad experiences with CASE approaches, or through missing or bad information. The approach itself does not require the assignment of roles to specific people, it merely describes the tasks that come with certain roles, such as developer and architect. The allocation of roles is the sole responsibility of the team or project management.

Besides a suitable methodology, the availability of tools that support realization of the required concepts is significant for the successful use of AC-MDSD. In our view, this support is not yet optimal. Better support on the UML tool side through distributed modeling, profiling, generator integration and OCL constraint support on the metalevel are particularly desirable. However, there are promising attempts to provide for example debugging or traceability at the metalevel as part of MDSD generators.

Architecture-centric MDSD is not the only MDSD variant. For example, profiles for business-related domains focus on much narrower application domains, yet their generation potential is usually much higher (often 100%). We deal with this more comprehensive topic in the remaining parts of this book. However, we want to point out that the tools introduced in this chapter can also be used for this purpose, especially as they do not depend on a concrete domain.

4 Concept Formation

Different approaches to Model-Driven Software Development exist, as we have seen in part and will see further in this chapter. Each approach comes with its own terminology, a phenomenon that is largely the result of differing intentions and histories. This is not really critical in practice, but it can lead to confusion and hamper communication. We therefore aim to create a common, conceptual superstructure, a unified MDSD terminology. We believe that this is helpful in gaining a deeper insight into the subject matter of this book and as a basis for further chapters.

4.1 Common MDSD Concepts and Terminology

Certain techniques, sub-areas or specific flavors of MDSD are not at all new. Terms like ‘generative programming’, ‘domain-specific modeling’, ‘product-line engineering’, and especially ‘code generation’ have been established for a long time. although they vary greatly in popularity. The OMG started a standardization process for certain core concepts with its MDA initiative, albeit with a primary focus on interoperability and portability. The MDA soon achieved a comparatively high degree of popularity and thus overshadowed the techniques listed above to a certain extent, yet without entirely overshadowing them. We therefore recognize the need for a unified common context, including its terminology, and we venture to create both. This conceptual context is Model-Driven Software Development, and the standard nomenclature of the OMG will serve as a basis as far as seems sensible and possible.

We will develop the common concepts and their relationships – that is, the MDSD concept space – in the form of a static UML model that we will expand and refine step-by-step.

4.1.1 Modeling

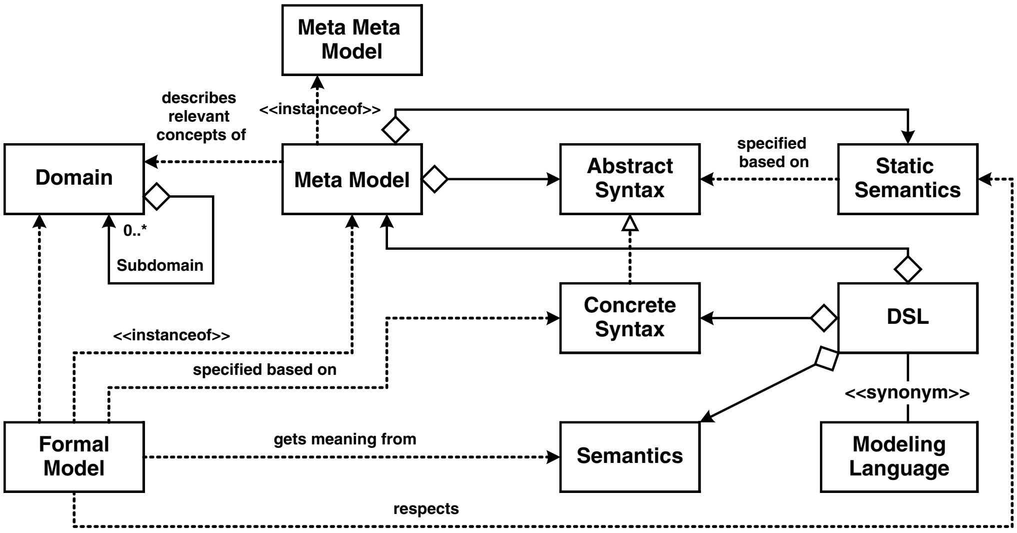


Figure 4.1 Concept formation: modeling and DSLs

The Domain

The starting point in MDSD is always a *domain*. This term describes a *bounded field of interest or knowledge*. To internalize and process this knowledge, it is useful to create an ontology of a domain's concepts. Domains can be motivated technically as well as professionally, if one wishes to make such a distinction. The case study in Chapter 3, for example, resides in the 'architecture for business software' domain, because it contains concepts like *Entity, SystemUseCase, Controller* and *Presentation*. A 'professional' domain could be 'insurances' with concepts like *insurance product, rate, loss or damage, service, policy holder, insurance contract* and so on. Further examples of domains might include 'embedded systems«, 'EAI' or 'astronomy'.

Domains can be composed of smaller *subdomains*. Two kinds of subdomains can be distinguished:

- Technical subdomains describe single parts or aspects of an entire system for which a specialized modeling language is appropriate. Typical examples include GUI layout and persistence.
- A comprehensive system can be broken down into *partitions* or content increments. In an insurance domain, for example, partitions could be defined for single sections or product types, such as a 'life«, 'vehicle«, 'liability' and so on.

The Metamodel

In the context of MDSD, it is absolutely mandatory to be clear about the structure of a domain (that is, its ontology), so that one can formalize this structure or its relevant part. This is the basis for every automation. Formalization takes place in the form of a metamodel. The UML profile in Section 3.2.1 is an example of such a metamodel. It specializes the basic UML metamodel with the relevant concepts of the domain. In general a metamodel is not necessarily UML-based, however.

The metamodel compasses the *abstract syntax* and the *static semantics* of a language, and is an instance of the *meta meta model*.

The Meta Meta Model

The term *meta* is relative. A metamodel describes concepts that can be used for modeling the model (i.e. in the instances of the metamodel). Consequently, the metamodel must itself have a metamodel that defines the concepts available for metamodeling. This is the role of the *meta meta model*. Meta meta models are important in two respects: for people defining metamodels, it defines the language used to do so. For tool integrators, the meta meta model is even more important, since it is the basis for integration among metamodels. So, as a tool builder, you typically require metamodels to be built using a specific meta meta model. Knowledge of the meta meta model is hardcoded into the tool.

You can find more details about models, metamodels and meta meta models in Chapter 6.

Abstract and Concrete Syntax

While the *concrete syntax* of a language such as Java specifies what a parser for the language accepts, the *abstract syntax* merely specifies what the language's structure looks like. An abstraction is introduced, for example, from such details as the spelling of keywords. One could therefore say that the concrete syntax is the realization of an abstract syntax. It's interesting that various concrete syntax forms can have a common abstract syntax. Put anther way, the metamodel of a domain can be expressed in different notations, for example in a graphical UML-based notation or in a textual one.

From a technical viewpoint, the abstract syntax of a language is instantiated typically by the parser— that is, it is used by the compiler to represent the input (the program source code) in the heap of the compile process in order to further work with it. This paradigm is familiar from the XML sphere: an XML document is formulated in the concrete syntax of XML, from which a generic XML parser instantiates a representation in memory – the DOM[1] tree. The DOM itself is the abstract syntax of XML.

UML is another example: it possesses a graphical notation consisting of small boxes and arrows as its concrete syntax, while its abstract syntax contains constructs such as *class, attribute, operation, association, dependency* and so on, and the relationships between these constructs.

The question naturally emerges: how the abstract syntax or the metamodel of a domain can actually be specified or written? For this purpose a meta meta model usually exists. In the context of the OMG standard, this is the MOF – the *meta object facility* (Chapter 6). Metamodels can be described using this original form. UML profiles constitute a special case in this context – in other words, MOF offers only one possible concrete syntax for the specification of metamodels.

[1] DOM: Document Object Model.

Static Semantics

The static semantics of a language determine its criteria for well-formedness. A typical example from the world of programming languages is the rule that variables must be declared. The syntax of a language (both abstract and concrete) typically cannot determine this, – that is, the parser does not recognize an undeclared variable as an error – but the compiler's static analysis will fail[2].

In Chapter 3 we saw how the static semantics of UML profiles can be defined formally using OCL expressions that build on the abstract syntax of the language, that is, the class structure of the profile. In the context of MDSD, static semantics are particularly important. They serve to detect modeling errors in terms of the formalized domain.

Domain-Specific Languages

We now have the concepts needed for understanding the notion of *domain-specific languages* (DSL). A DSL serves the purpose of making the key aspects of a domain – but not all of its conceivable contents – formally expressable and modelable. To this end, it possesses a metamodel, including its static semantics, and a corresponding concrete syntax. That alone is not enough: the dynamic semantics required to give meaning to the constructs of the metamodel are still missing. The semantics of a DSL are relevant in several respects: on one hand, the modeler must know the meaning of the language elements at their disposal to be able to create reasonable models, while on the other, automatic transformations of the models must execute exactly these semantics. More about this later.

The semantics of a DSL must either be well-documented or be intuitively clear to the modeler. This is made easier in that the DSL adopts concepts from the problem space, so that a domain expert will recognize its 'domain language'[3].

Often the term *modeling language* is used synonymously with DSL. We prefer the term *DSL*. because it emphasizes that we always operate within the context of a specific domain.

DSLs can vary in their power and complexity. Simple textual configuration options with validity tests can constitute a DSL, while at the other end of the spectrum DSLs can be graphical languages with corresponding language-specific editors.

Two classes of DSL editors exist: generic tools, such as UML tools that are configured via a profile, and custom-made DSL-specific tools.

Formal Models

We will now talk about *formal models* in the context of MDSD. Formal models are the starting point for automated transformations – even a purely interpretative approach requires a formal model. A formal model needs a DSL, and is thus obviously connected with the respective domain. It is formulated in the DSL's concrete syntax and it constitutes an instance of the given metamodel at least conceptually, and in most cases also technically.

[2] Conceptually, static semantics belong to the syntax rather than to the semantics of a language.

[3] In an architecture-centric context, the domain expert is more of a software architect, because the domain here is software architecture.

A formal model is therefore a sentence formulated in the DSL, and obtains its meaning from the DSL's semantics. It is clear therefore that in MDSD the context of the domain is of the utmost importance.

Let's now look at a few examples:

- The architecture-centric designs from the case study in Chapter 3 are formal models in the context of MDSD. Their domain is the architecture of business software.
- A Java program is an instance of the programming language Java, or rather of its corresponding metamodel. Java also possesses semantics. But what is the domain of Java? One could say that it is 'Turing-calculable functions' – that is, in principle, 'everything that can be done with computers«. The same is true for Executable UML, a directly executable variant of UML described in Chapter 12. In the context of MDSD, such a domain is of little help, because our approach here is to formalize concepts of a higher-level problem space in order to set them apart from the abstraction level of a programming language.
- A Powerpoint slide per se is not a formal model in terms of MDSD, although it possesses a very generic metamodel (*rectangle, arrow, eclipse, text*), yet there are no semantics for such models. Once a real DSL is defined in a Powerpoint-based, concrete syntax, it would then be possible further to process Powerpoint slides that are accordingly well-formed as formal models and, for example, generate code for graphical user interfaces (GUIs) from them.

4.1.2 Platforms

We now expand our ontology of the domain MDSD by the next partition: we represent the problem space with the help of the DSL, allowing formal models to be processed (transformed) further or interpreted. To this end, we need something on the 'other side' – that is, in the solution space – that supports the transformation or respectively interpretation – something on which the code that has been generated by the transformation builds.

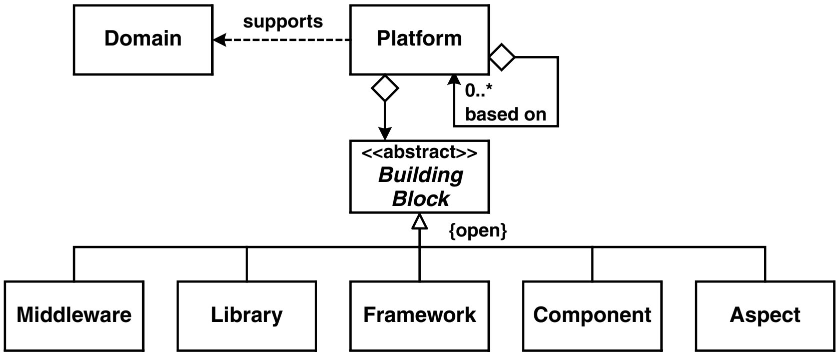


Figure 4.2 Concept formation: platforms

The Platform

The term *platform* is used in the MDA context (Chapter 2 and Chapter 12) as well as in software production line engineering. It is general enough to be useful for the description of MDSD. The platform has the task of supporting the realization of the domain, that is, the transformation of formal models should be as simple as possible. In the case study in Chapter 3 we used J2EE with Apache Struts as a platform, plus some superclasses and helper classes that we created ourselves. The domain's DSL (the architecture-centric UML profile used here) describes the problem space (*entity, controller, presentation*), but not the solution space, the platform. Clearly, the easier the transformations are to build, the more powerful is the platform. If we removed Struts and our helper classes from the platform, the effort required in mapping the dynamic constructs of our DSL (its activity diagrams) as part of the code generation would be much greater. Platforms can also be cascaded.

In the extreme case of interpretation, the platform assumes the role of a virtual machine (an interpreter) for executable models, so that the model transformation becomes trivial.

Building Blocks

A platform can be founded on existing building blocks. These can be middleware, libraries, frameworks, components, or aspects in terms of AOP[4].

4.1.3 Transformations

After looking modeling and platforms, we can now connect these two partitions in our conceptual space:

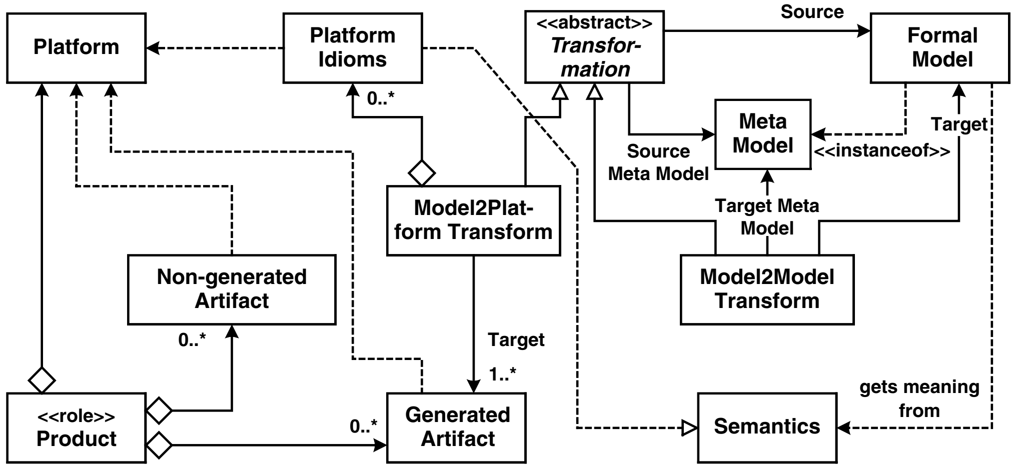


Figure 4.3 Concept formation: transformations

[4] Aspect-Oriented Programming [Lad03].

Transformations

MDSD transformations are always based on a source metamodel, since the source model to be transformed is exactly one instance of this metamodel. The transformation rules can only be based on the metamodel's constructs, and this is its main purpose, as the transformations implement the DSL's semantics.

We distinguish between model-to-model transformations (*Model2ModelTransform*) and model-to-platform transformations (*Model2PlatformTransform*), the latter often also called *model-to-code transformation*.

A model-to-model transformation creates another model. However, this model is typically based on a different metamodel than the source model. Such transformations generally describe how the constructs of the source metamodel are mapped to the constructs of the target metamodel. The MDA implements this approach with its query/view/transformation specifications, as described in Chapter 12.

A model-to-platform transformation, in contrast, 'knows' the platform and generates artifacts (*generated artifacts*) that are based on the platform. Generated source code that fits into an existing framework would be one example. For this class of transformation, a target metamodel is not needed, because usually we are dealing with simple text replacements exclusively. The template definitions in Chapter 3 fall into this category. Note that in addition to transformations, interpreters can also be used to execute models (see Section 8.4).

Platform Idioms

The fact that model-to-platform transformations can use the complete knowledge about the platforms provides them with a powerful tool, *Platform-specific Idioms (Platform Idioms)*, which can be used transparently by the transformations. In the case study in Chapter 3 we generated code that uses the Business Delegate pattern to keep the domain model clear of the EJB programming model. The use of this pattern did not need to be specified anywhere in the model – the knowledge of where and how the pattern is to be applied is stored in the transformations alone.

The Product

MDSD pursues the goal of creating a software *product* in part or in whole through one or more transformations. The product can be an entire application or merely a component to be used as a building block elsewhere. Such a product aggregates the platform, generated, and sometimes even non-generated artifacts, in terms of MDSD. Non-generated artifacts can for example be application-specific helper classes or manually-programmed business logic.

4.1.4 Software System Families

The next MDSD partition looks at the correlations between a domain's products and addresses the aspect of reusability.

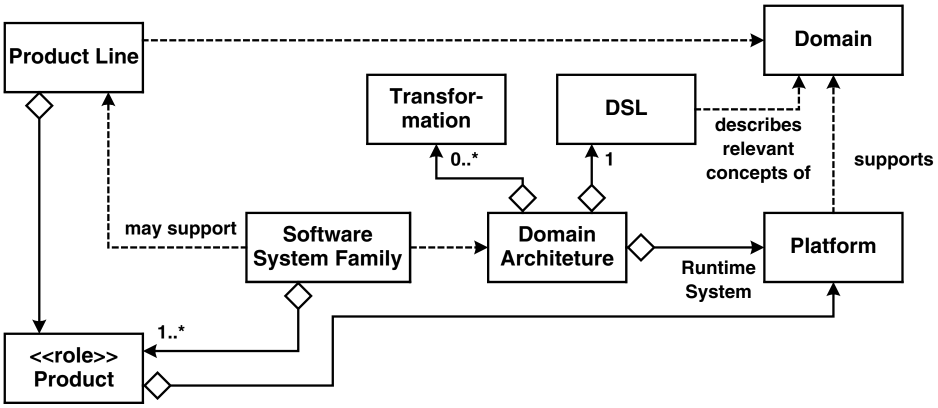


Figure 4.4 Concept formation: domain, product Line, software system family

The Domain Architecture

The metamodel of a domain, a platform, and the corresponding transformations, including the implemented idioms, are the tools that are needed to make the transition from the model to the product, whether completely or partially automated. The aggregation of these items is what we generally call the *domain architecture* – the central MDSD concept. Other than the architecture of a platform, a domain architecture determines which concepts are supported formally (although not necessarily the concrete syntax) and how these are to be mapped to an existing platform. The platform assumes the role of the runtime system in this context: a domain architecture is always relative to a platform. The generative software architecture in the case study in Chapter 3 is one example of a domain architecture.

Software System Families

Obviously a domain architecture is suitable for building all the products that can be expressed with the given metamodel and that are realized on the same platform.

The set of all products that can be created with a certain domain architecture is commonly referred to as a *software system family.* In other words, the software system family uses the domain architecture for its realization, and the domain architecture is reusable for all products of the software system family. The domain architecture must be flexible enough to allow the expression of the differences (variabilities) between various products that make up the software system family.

The Product Line

A *product line* is a set of complementary single products. From a user's perspective, the products in a product line can constitute alternatives – that is, they be applicable in different but related contexts – or can complement each other content-wise and thus define a 'suite«. It is important to notice that the products of a software system family do not necessarily share any technical commonalities. A software system family can form the basis of a product line, but doesn't have to.

4.2 Model-Driven Architecture

The contents, direction, and trends of MDA are dealt with in Chapter 12. This section discusses how this standard can be conceptually placed in the general framework of MDSD described in the previous section. Figure 4.5 shows the placement of MDA in the concept space of MDSD.

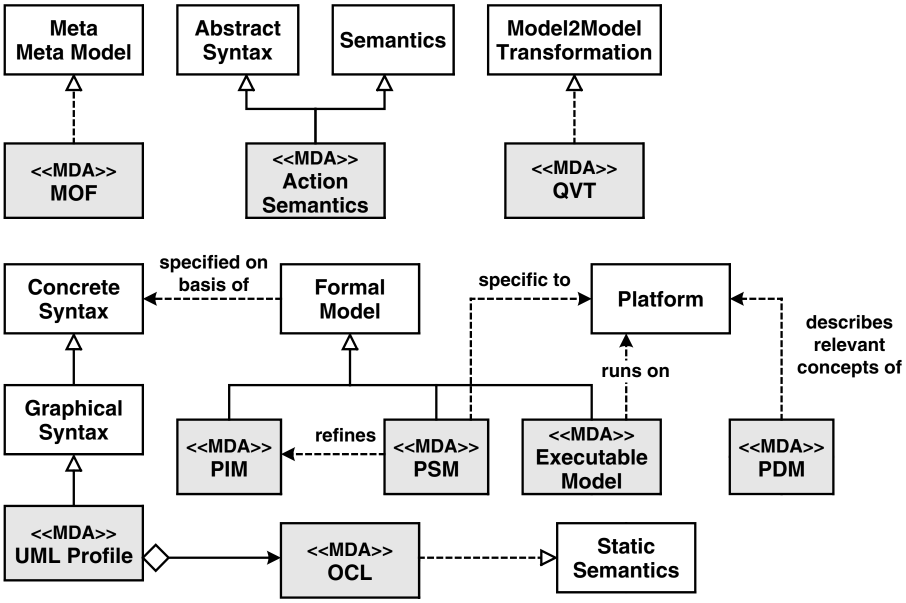


Figure 4.5 Concept formation: placement of MDA concepts

Ontologically, MDA is a specialization of MDSD with the following characteristics:

- Software system families and product lines have no direct equivalent in MDA terminology, and the terms are not directly relevant in this context.
- MDA uses MOF as its meta meta model – that is, as a means for the definition of metamodels.
- MDA expects DSLs to be based on MOF. Any notations and metamodels are feasible as long as they have been defined with the help of the OMG meta meta model. In practice, MDA recommends the use of UML profiles as a concrete syntax for a DSL. The DSL is therefore predisposed to use UML at its core. Accordingly, the static semantics are specified by OCL expressions.
- Various perspectives on formal models are defined: a domain model can be specific (PSM) or non-specific (PIM) relative to a platform. The MDA recommends that transformations between models are carried out in several steps, but it doesn't prohibit a direct PIM-to-code transformation.

- To be able to describe even the final transformation, that connecting with the platform, as a model-to-model transformation, the platform must also be described via a metamodel. For this purpose, PDMs – Platform Description Models – are used.
- At this stage no standardized transformation language exists. The OMG's QVT is expected to be finalized by the end of 2006 (see also Section 10.5). Its goal is mainly the description of transformations between source and target metamodel for model-to-model transformations.
- Executable UML models stand out and are one of the main objectives of many MDA representatives: they are more or less directly executable on a suitably powerful and generic platform – that is, they are interpreted by a UML virtual machine, or completely compiled via transformations, so that they can be executed on a lower-level platform. In contrast to a domain like 'insurance business«, which has a professional focus, we are dealing with a 'domain' here that corresponds to the expressiveness of a programming language, so a UML profile is not needed. This obviously increases the semantic gap between the modelling language and the professional domain.
- The action semantics of the UML are an essential building block for executable UML, because they allow the specification of algorithms in an abstract form. When a (tool-specific, hence non-standardized) concrete syntax is used, you can use action semantics to program like in any other programming language, although the program is integrated with the static model's content.

4.3 Architecture-Centric MDSD

AC-MDSD is one of the main issues of this book, and we are now able to introduce the terminology properly. Figure 4.6 shows a classification of the approach in the general context of MDSD:

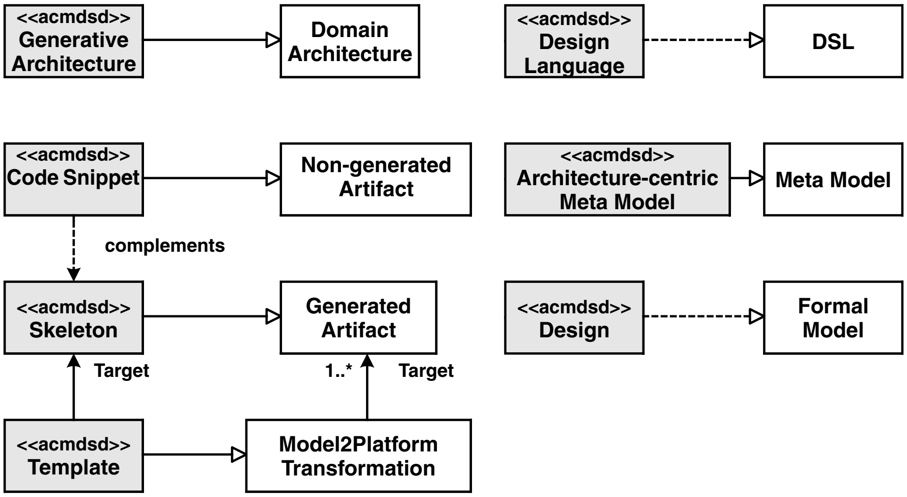


Figure 4.6 Concept formation: classification of the AC-MDSD concepts

AC-MDSD is a specialization of MDSD that conceptually overlaps with MDA. It builds on the following cornerstones:

- The domain is architecturally motivated, for example 'architecture for business software' or 'component infrastructure for embedded systems«'.
- The products to be created are usually complete applications.
- From a black box viewpoint, usually only singe-step model-to-platform transformations exists – or more precisely, model-to-code transformations. However, these can be internally structured (white box), serving modularization purposes for sequential execution of several transformations.
- The DSL's metamodel therefore contains architectural concepts that are as abstract as possible, as described in Chapter 3.
- The DSL is also called a *design language*. Usually, UML profiles are used here, sometimes combined with additional textual specifications.
- The formal models are also called *designs*.
- Typically, the model-to-platform transformation is a template that shows great similarity to the generated code, and can thus be extracted easily from a reference implementation (Section 2.5).
- The transformation does not aim to create the complete application, but merely an implementation framework that contains the architectural infrastructure code, the *skeleton*.
- The non-generated, implementation code (»business logic«) is manually implemented in the target language to create a *code snippet*. For this purpose, the generated skeleton may contain protected regions to supplement the application logic that persists after iterative regeneration. Alternatively, generated and non-generated code can be integrated using suitable design patterns, as described in Chapter 9.
- Design language, templates, and platform constitute a *generative architecture*. Here, we are obviously dealing with a special domain architecture that supports the software system family.

With the creation of a platform that provides the most important architectural concepts, AC-MDSD tries to minimize the gap between model and target platform. The metamodel used for application modeling can be tightly aligned with this target architecture/platform – hence the name. In consequence, one can easily achieve a single generation step instead of having to apply several sequential transformation steps.

4.4 Generative Programming

As we are discussing this approach here for the first time, we do not only look at its relationship with MDSD, but also want to provide a brief overview of its motivation, history, and primary focus.

The term Generative Programming (GP) has been in use for several years. The term became popular mainly through Krzysztof Czarnecki's and Ulrich Eisenecker's book *Generative Programming* [EC001], which defines GP as follows:

Generative Programming is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.

The driving factors in GP are:

- Adherence to industrial production paradigms such as those that of automotive manufactuting. The metaphors of a *production line* and an *order form* are widely used.
- GP claims to produce complete products (applications) from specifications – 100% automation.
- GP emphasizes the creation (configuration) of applications from predefined atomic components.
- Generation of products that are optimized for specific aspects such as performance or code size.

The goal of GP is therefore the creation of precisely fitted and optimized products from a model such as a formal requirements specification. To illustrate this, let's take a look at the generative domain model:

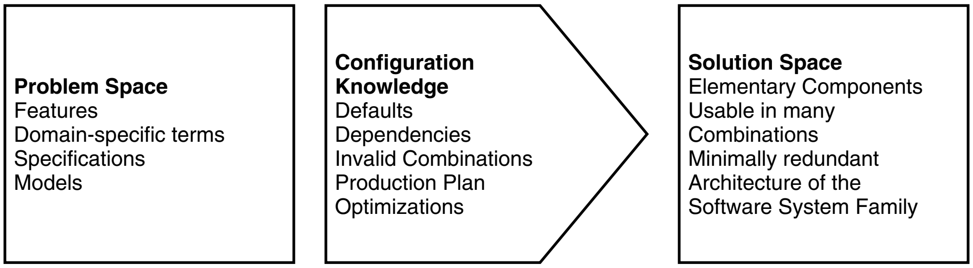


Figure 4.7 Domain model of Generative Programming

The formal requirements of an application are defined in the domain's problem space. This can be done via different models and specifications, among others, with the help of feature models (see Section 13.5.3). In the solution space, the respective application – the product – is implemented through elemental components. These must combine well and be non-redundant as far as their functionalities are concerned. In this context, components can also be aspects in terms of AOSD. The relationship between them is established by configuration knowledge, which includes among other aspects useful defaults, dependencies, and illegal combinations: products with illegal specifications will therefore not be created. Moreover, the configuration knowledge contains the production plan as well as possible optimizations. Thus it also contains the generator.

Let's now look at the approach and its terminology in the context of its classification in MDSD (Figure 4.8):

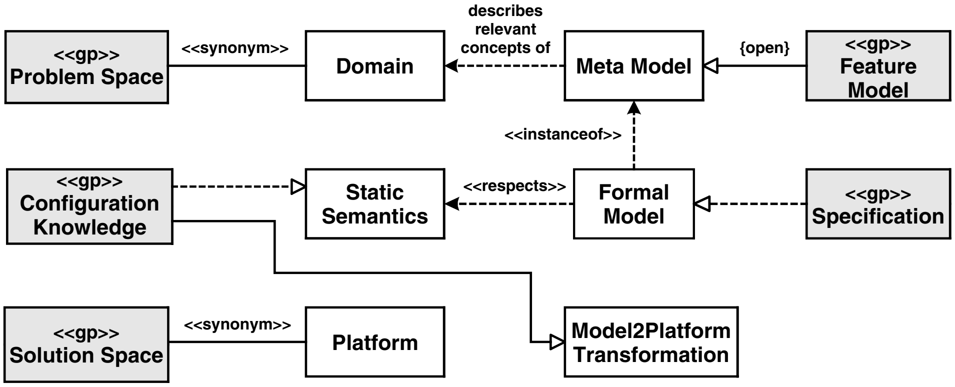


Figure 4.8 Concept formation: classification of Generative Programming concepts

Ontologically, GP is a special form of MDSD with the following characteristics:

- The idea of the software system family plays a central role in GP. It is assumed that a domain is modeled via feature models (Section 13.5.3) and single products are generated on this basis.
- Traditionally, the idea of (UML) modeling is less pronounced. Instead an – often textual – DSL is defined based on domain analysis, which serves to make products of the family describable.
- Feature models often serve as a basis for DSL or the metamodel, although this is not mandatory. In principle, any type of metamodel or DSL can be used in GP.
- If a feature model is used to describe the specification of the product, it assumes the role of the formal model in the context of MDSD.
- The domain is also called a *problem space*, whereas the platform and the components that constitute the product are termed the *solution space*.
- The configuration knowledge is stored in a generator that performs a one-step model-to-code transformation, as in AC-MDSD. The static semantics (the recognition of invalid product configurations) are similarly realized with configuration knowledge.
- The platform typically consists of maximally combinable and minimally redundant components, which ultimately realize the expressive power of DSL.
- The tools used in GP are often feature modeling tools. Of course this is not inevitable: depending on the DSL, other tools can be used as well. In the context of C++ template meta programming, for example, the C++ IDE would be the modeling tool of choice.

Even though the definition of GP does not enforce it, static generation techniques are often applied. This is due to GP's emphasis on products optimized for efficiency (performance or footprint). The configuration of frameworks or the creation of a virtual machine is fairly uncommon. Nevertheless, it is important to understand that GP is not simply code generation. GP should also not to be equated with C++ template meta programming, which merely constitutes *one* implementation technology for GP.

Traditionally, GP has focused more on the creation of small but highly efficient products. Large, distributed enterprise applications or families have been of lesser interest. For details about more recent developments in the GP field, please see Krzysztof Czarnecki's Web site [CH05].

4.5 Software Factories

The term *Software Factories* has been coined by Microsoft and is described extensively in Jack Greenfield and Keith Short's book of the same name [GS04]. In a nutshell, a Software Factory is an IDE specifically configured for the efficient development of a specific kind of application, such as applications in a specific domain. The configured IDE makes the use of domain-specific models, DSLs, frameworks, and patterns as simple as possible. The concept of Software Factories is thus the industrialization of software development 'from craftsmanship to manufacturing'. Software Factories are described by some people as 'doing product lines the Microsoft way' – for some detail about product-line engineering, see Section 13.5. While the product line aspect of that statement is certainly true, Microsoft is working on making sure the approach is not considered to be a Microsoft-only concept. For example, the respective workshop at the OOPSLA 2005 conference [SFW05] ensured that people from outside Microsoft were on the program committee. However, the public perception is still that it is very much Microsoft-centric.

Since the concept of Software Factories looks at the complete product-line engineering process, it is much wider in scope than 'just' Model-Driven Software Development, although DSLs, modeling. and transformations are an important ingredient. We will therefore look first at the overall approach, then at its DSL-specific aspects.

4.5.1 The Software Factory Schema

The cornerstone of the whole concept is arguably the Software Factory Schema. This defines the viewpoints that are useful and necessary for building a system of the respective kind. For example, an enterprise system might encompass the following viewpoints:

- Presentation, including form layout and workflow
- Component structure and business data model
- Persistence mapping
- Deployment viewpoint

For each of these viewpoints, the schema identifies core artifacts, as well as the most efficient way of producing them. Such ways could include manual programming, using patterns in specific ways, using frameworks that are extended or configured, as well as designing and subsequently using DSLs and then generating various artifacts such as code or configuration files. The viewpoints can depend on each other and thus form a directed graph – in other words, the deployment viewpoint depends on the component structure: you cannot deploy what you haven't defined.

The schema is therefore a conceptual framework or 'recipe' for separating the concerns in the respective application domain, based on abstraction level or its position in the architectural or development process. The schema also identifies the commonalities as well as the differences among the applications in the domain addressed by the schema.

4.5.2 The Software Factory Template

The schema is basically a structured document. However, to be able to configure the development environment for the respective kind of applications– and such tool configuration is the ultimate goal of Software Factories – we must make all this 'tool usable'. This configuration for the IDE is called a Software Factory Template. It can be loaded into your IDE (usually Visual Studio) to configure the IDE for developing the respective kind of application. Thus, for example it:

- Provides the necessary frameworks or libraries.
- Contributes certain kinds of projects whose structure is suitable for the factory.
- Delivers build scripts.
- Extends the IDE with new DSL editors and transformations.

We also need to have the necessary tools to build some of these artifacts in the first place. While building frameworks requires nothing specific from an IDE, this is different for DSLs. Tools for defining metamodels, concrete syntax, and transformations are required.

4.5.3 The Role of DSLs and Their Relationship to MDSD

Up to this point we have looked at the general approach to software (product line) development proposed by Software Factories. From this approach, it is obvious that it does not make sense to compare such an approach to MDSD directly. However, we can compare the use of models as well as the construction of the respective infrastructure in Software Factories to MDSD. This is the goal of this section.

In general, Software Factories use the concepts defined in Section 4.1 without major changes or renamings. Domain-Specific Languages are used to build models. Those languages are often – but not necessarily – graphical. Visual Studio provides tools to define the metamodels as well as the concrete syntax and editors – remember its tooling focus. Microsoft does not use any of the OMG standards for their infrastructure: DSLs are not UML based, and metamodels are not based on the MOF, but rather use the MDF, the metadata framework for that purpose.

From the application developer's perspective, models are first-class artifacts in development projects, and editors and transformations integrate seamlessly into the IDE. From the perspective of the infrastructure developer, metamodels, editor definitions and transformations are first-class artifacts, and the tools to build them are seamlessly integrated into the IDE. Microsoft also consistently follows the approach of not modifying generated code. Integration can happen using patterns, as is described in Section 8.3.1. In the .NET environment, these have specifically added the concepts of partial classes, which means that a class definition can be spread over many files, based on the idea that some of these files are generated and some are handwritten.

4.6 Model-Integrated Computing

Model-Integrated Computing started in the technical computing area, more specifically in the context of distributed realtime and embedded systems (DRE systems). Such systems are used in many domains: important examples are industrial monitoring and control systems, defense, and avionics. As a consequence, you come across the term MIC used mainly by practitioners in those industries and their associated research institutions. For example, Vanderbilt University's Institute for Software Integrated Systems (ISIS) is very involved in MIC. ISIS is also the builder of a very popular MIC tool called GME – the Generic Modelling Environment [GME]. Section 11.3.5 includes two screenshots taken from GME.

Technically MIC is conceptually quite compatible with MDSD, and specifically aims at using 'real' DSLs rather than UML profiles, and several models to describe the various aspects of a system. There are several points that should be specifically mentioned, though:

- Models are at the center of the complete lifecycle of systems, rather than just during their development. Analysis, verification, integration, and maintenance are also addressed.
- Since MIC is traditionally used for dependable systems, the verification of models is a primary concern, for example using simulation techniques.
- Model-to-model transformations are important, not so much because of the MDA-like multi-step transformation approach, but rather to be able to transform (certain aspects of) models into different representations, so that various analysis, verification, and simulation tools can use them.
- As exemplified by GME, building 'meta tools' – tools that can be used to build modeling tools efficiently – are a cornerstone.

Note that MIC is also now supported by the OMG. Currently, this comprises the MIC PSIG and the yearly industry-oriented workshop. (The OMG's MIC initiative is not related to MDA.)

4.7 Language-Oriented Programming

The term Language-Oriented Programming has recently been used mainly by Sergey Dmitriev and his company JetBrains, the makers of the IntelliJ IDE. They are working on a new product called MPS, the 'meta programming system' [MPS]. This tool lets you define your own languages integrated in the MPS IDE. This means that defining a language also entails defining the respective editor, compiler (and transformations), and debugging support. In the context of MPS the languages are typically textual. Again, domain-specific metamodels play a central role – the metamodel is the first step in defining the languages.

MPS is an example of what Martin Fowler calls a 'language workbench' in his articles on DSLs [Fow05]. He argues that the fate of DSLs basically depends on how easy it is to build new languages and integrate them into everyday development environments. He therefore considers language workbenches the 'killer app' for DSLs. In addition to MPS, other similar tools exist:

- GME, introduced in Section 4.6, can quite well be considered a language workbench – albeit focusing on graphical languages. The same is true for MetaEdit+ and Xactium's XMF Mosaic [XMF].
- Historically, the 'intentional programming' research project lead by Charles Simonyi had the same goals. According to [EC00] the tool they built – which has not ever really left Microsoft – must have been quite impressive.
- Charles Simonyi, as well as a couple of other people, have meanwhile founded a company called Intentional Software. The community expects that they will build a comparable tool.

4.8 Domain-Specific Modeling

Domain-Specific Modeling (or DSM) is primarily known as the idea of creating models for a domain in a DSL suitable for that domain. In this respect, DSM is mostly about the modeling aspect of MDSD. However, generation techniques have been in use for some time in the DSM community. It can be observed that the discrepancies between DSM and MDSD are beginning to dwindle. One of the best-known players in the DSM space is the Finnish company Metacase Consulting, with their tool MetaEdit+. A screenshot taken from MetaEdit+ can be seen in Section 11.3.5.

5 Classification

We established a uniform terminology for MDSD in Chapter 4, so we can now take on the classification of related topics.

5.1 MDSD vs. CASE, 4GL and Wizards

One remarkable characteristic of Model-Driven Software Development is that the development environments used are by no means generative and static, but that in fact *any* target architectures, modeling and target languages, interfaces, and runtime components can be supported.

In contrast, a CASE or 4GL tool will predetermine at least one component of a domain architecture – and in most cases, all of them:

- DSL (modeling language)
- Transformations
- Platform and target architecture

Such tools focus on a domain that is *not* specific: they try to adhere to the dogma of 'one size fits all' – one premeditated combination fits all applications. This assumption is completely unrealistic in practice and causes significant problems. Typically, 80% of an application can be created fairly quickly in this manner, whereas the remaining 20% will eventually require 80% of the total effort. This is because the tools' inflexibilities enforce workarounds to combat them. Individual architectural requirements and interfaces cannot be applied here, let alone domain knowledge.

MDSD means an explicit abandonment of all 'one size fits all' approaches. Its emphasis is clearly on development *methodology*, not on development *environment*.

As a rule, all the aspects one wishes to generate will be implemented manually and verified at least once. Only in a second step is the domain architecture derived from them, which will then generate specific features automatically based on an input model. Questions that arise in the context of traditional, generative approaches can thus be put aside:

- How good is the (generated) system's runtime performance?
- How good is the quality/legibility of the generated source code?

…and so on. All these factors are as good or as bad as the reference implementation from which the transformations are inferred.

It is of course not necessary to start from scratch: as soon as a software system family is generatively applicable, its usefulness multiplies with each application that can use its technological basis: that is, any application that is a ‘member’ of that family.

MDSD cannot be compared to a code wizard or pattern expander. The ‘useful helpers’ part of commonly-used development environments, or the pattern expansion of some UML tools, allow for automatic generation of class skeletons, for example for EJBs, or for the generation of class structures (depending on the design pattern). Other than in MDSD, this step can usually be carried out only once. The repeatable transformation achieved with MDSD, while simultaneously maintaining the customization made, is missing. Moreover, the extra abstraction level introduced via MDSD is lost.

5.2 MDSD vs. Roundtrip Engineering

Roundtrip engineering is the concept of being able to make any kind of change to a model as well as to the code generated from that model. The changes always propagate bidirectionally and both artifacts are always consistent. The transition from code to model (the reverse engineering) is especially interesting in this context.

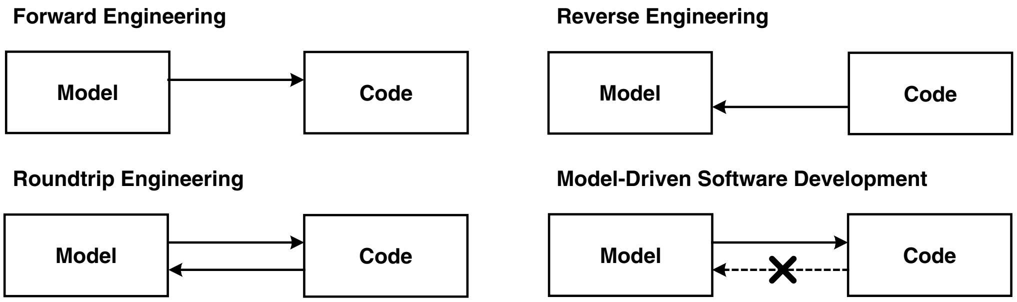


Figure 5.1 Forward/reverse/roundtrip engineering and MDSD

In the context of these approaches the model typically possesses the same abstraction level as the code (that is, ‘one rectangle per class«). It is actually the visualization of a program’s structure. In such a scenario, it is both feasible and useful to track changes to the code in the model automatically.

MDSD takes a different approach: the model is definitely more abstract than the code generated from it. Thus it is generally impossible to keep the model consistent automatically after a manual change of the generated code. For this reason, manual changes to generated code should be avoided. A precise definition that states which parts are generated and which are implemented manually is therefore necessary. To obtain the desired code without using round-trip engineering, you can resort to various other methods [Fra02]:

1. *Abstraction*. The abstraction level of decisions is raised to model level. This only makes sense if a corresponding abstraction on the model level can be identified.
2. *Tagging the model*. This involves the adoption of decisions in the code into the model without raising the abstraction level. This procedure is called 'tagging' the model with implementation decisions. It soon leads to contamination of the models with implementation concepts that are not derived from the modeler's domain or the domain expert, and therefore constitutes a potential source of errors. When tagging the model is used, it should preferably be done via introduction of a technical subdomain (see Section 8.3.3), so that modelers and domain experts are spared the implementation concepts.
3. *Separation of code classes*. This involves the adaptation of the target architecture in such a way that manually-created code must be written into classes specifically created for this purpose.
4. *Tagging the code*. This consists of the introduction of protected regions to the code, and is accomplished through the use of special tags that protect the code placed between them from overwriting during regeneration. This is a pragmatic procedure for blending generated and manually-created code at generation time. Different variations of this procedure are in existence, including procedures that not only allow the insertion of manually-created code, but also allow the optional replacement of generated instructions.

This list reflects the various solution's elegance in strictly descending order. Tagging the model still allows a clear separation of responsibilities and enables a fully-automated regeneration without further manual treatment. Tagging the code should only be applied with care, because it hampers versioning, amongst other things.

5.3 MDSD and Patterns

Patterns (architecture patterns, design patterns, idioms) don't have anything to do with MDSD specifically. Patterns are documented best practices for solving specific recurring problems. There are, however, some interesting relationships between the two fields. We outline them in this section.

5.3.1 Patterns and Transformations

The relationship of patterns with MDSD stems from the fact that transformations are a form of 'formalized best practices' insofar that structures in the target model (and correspondingly in generated code) corresponding with the structure of a pattern's solution are often created through transformations. The figure below illustrates the dependency between GUIs and entities mapped to, for example, the implementation of the observer pattern (see Figure 5.2).

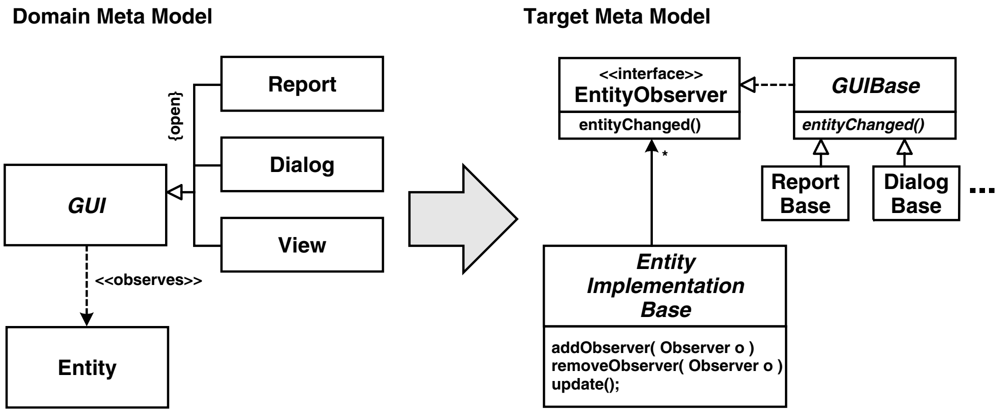


Figure 5.2 The use of patterns in transformations

However, in this context it is important to understand that a pattern doesn't just consist of the solution's UML diagram! Significant parts of a pattern explain which forces affect the pattern's solution, when a pattern can be applied and when it cannot, as well as the consequences of using the pattern. A pattern often also documents many variations of itself that may all have different advantages and disadvantages. A pattern that has been implemented in the context of a transformation does not account for these aspects – the developer of the transformations must take them into account, assess them and make decisions accordingly.

Another important issue is that the use of MDSD allows additional alternatives for solving specific problems. In the case of the Observer pattern, for example, nobody would consider hard-wiring the dependencies and notifications into the code, because this would be both extremely inflexible and a lot of extra work. In specific circumstances it might be the best solution (with respect to performance or footprint) to use the latter approach and generate the necessary code from models that describe the dependencies. Patterns have not been described with code generation in mind, so MDSD might make additional solutions to the problem described by a certain pattern feasible that would not have been considered seriously in a non-generative environment.

An MDSD transformation can also serve to generate a solution structure (including its behavior) into a model or code. However, the consideration or whether and how a pattern is applied must still be made by the developer – the developer of the transformation, that is, not the developer who uses the transformation in application development.

5.3.2 Patterns and Profiles

Some UML tools and also MDA (or more precisely, the EDOC pattern profile, see Section 12.2.7) define tool-supported macro definitions at the UML level that serve to package models as 'patterns«. This is misleading, because a real pattern, as we have already explained, is much more than merely a UML macro. In addition, in many tools these 'patterns' can only be expanded once, so that the compaction is lost afterwards.

5.3.3 Patterns Languages as a Source of DSLs

Pattern languages use a collection of patterns to describe a potentially complex (technical) design or architecture: examples are EAI applications [Fow04], remoting infrastructures [VKZ04], and component containers [VSW02]. Such a collection of patterns is highly structured, and the dependencies among the patterns are clearly defined: usually they have to be read in sequence, because a specific pattern builds on its predecessor(s). Often the patterns are aligned with the main structural artifacts of the system they describe, or illustrate its most important behaviors. Thus, among other things, pattern languages are a conceptualization of the class of systems they describe.

As a consequence, such pattern languages are a good start for mining elements for metamodels that are needed for a DSL that can describe the relevant class of systems. Let's look at remoting patterns as an example: if you wanted to build a DSL for configuring/generating remoting middleware infrastructures, the pattern language helps you identify key concepts you might need to represent in the DSL, such as:

- The invoker
- Interfaces
- Client and server request handlers
- Object identification
- Lifecycle alternatives such as lazy/eager acquisition, pooling, or leasing
- Asynchronous communication using fire and forget, sync with server, poll objects, or result callbacks

For each of these, the patterns in the pattern language describe 'hot spots' that might need configuration when describing such a system, in order to be able to generate it. The concepts, their relationships, as well as their configuration alternatives can quite easily be refactored into a metamodel that underlies a DSL for remoting infrastructure description and configuration.

5.4 MDSD and Domain-Driven Design

The term *Domain-Driven Design* (DDD) became popular mostly through the book of the same name written by Eric Evans [Eva03]. When it comes to developing a domain-specific platform, this approach has something in common with MDSD: Evan's DDD do not use DSLs, nor does he recommend generating code. Instead, Evans describes techniques, patterns, and process elements that aim at the creation of 'good«, and mostly UML-based, models of a domain, and at the creation of code that preserves or expresses an application's design as faithfully as possible.

Although there is no strong interrelation between MDSD and DDD, it is nevertheless useful to learn more about DDD to increase the quality of the modeling process, generated code, and the programming model.

5.5 MDSD, Data-Driven Development and Interpreters

In data-driven development essential parts of application functionality are defined using data structures that are read and interpreted by a framework. Traditionally these data structures are defined in a relational database that also stores the application data. The aspects described with such data structures are often those that vary from application installation to installation, allowing for easy customization of the application at a customer site, often even without the need to restart the application after a change.

In an enterprise application, for example, special tables in the database define data structures (and thus the structure of the application data tables in the same database), field validation rules, or the structure and workflow of forms.

Just as in MDSD, you have to define a metamodel that defines the structure of the data that you use to configure the application. Instead of using that data *before* runtime to generate the application artifacts, however, frameworks are used to customize the system dynamically at runtime. The consequences are obvious:

- With data-driven approaches, you can change the application dynamically without the need to regenerated/restart.
- Performance might be slightly worse because of the framework overhead.
- If your platform, such as in J2EE, requires the presence of specific artifacts such as deployment descriptors in order to take advantage of specific platform features such as security, you might be required to actually generate these artifacts.

Note that while the data-driven development legacy is largely ignored in the MDSD discussion, the two areas are actually quite closely related. The conceptualization of a problem domain in the form of metamodels is a core concept in both. MDSD platforms also often contain data-driven aspects for which the MDSD generator generates the input data, an approach that is explained in Section 7.6.

An essential aspect of data-driven development is interpretation of models, as the above discussion shows. In fact, the frameworks that work with the data in the enterprise application example above can be thought of as an interpreter. However, interpreters are associated traditionally more with executing behavior, such as mathematical calculations in the insurance domain. We take a closer look at interpreters in Section 8.4.

5.6 MDSD and Agile Software Development

An iterative-incremental process is a strong ally for MDSD, and strict timeboxing helps to implement the feedback loop between architecture development and application development smoothly. One of the highest priorities in agile software development is the development of runnable software that can be validated by both stakeholders and end users– as it is also in MDSD. MDSD encompasses a number of techniques and methods that enable the use of principles of agile software development in complex projects. These techniques support agile requirements management and the regular validation of software under construction. These issues will be addressed in more detail in Chapter 13.

It is not the goal of MDSD to dictate a particular (agile) method. As long as the few, but strict, MDSD rules for iterative software development are observed, the micro-activities of the development process can be governed by any agile methodology. In practice, the assignment of roles in agile teams is based on the strengths and abilities of individuals rather than on rigid job descriptions.

MDSD emphasizes the importance of models. These have the same significance as source code rather than that of optional documentation. The 'production' or generation of a system via domain architecture is automated to the same extent as the automation of 3GL language compilation. The issue of agility concerns the creation of the domain architecture as well as the modeling and implementation of an application.

5.6.1 The Agile Manifesto and MDSD

In the remainder of this section we explore how well MDSD and agile development match each other. Let's first look at the agile manifesto, which can be found at *http://agilemanifesto.org.*

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions over processes and tools.*
- *Working software over comprehensive documentation.*
- *Customer collaboration over contract negotiation.*
- *Responding to change over following a plan.*

That is, while there is value in the items on the right, we value the items on the left more.

Let's now analyze these statements one by one in the context of MDSD.

Individuals and Interaction vs. Processes and Tools

This statement first and foremost expresses a high esteem for people. After all, it means that no over-formal processes that don't heed people should be established. A team should define its own development process, suited to its specific conditions, and continue to evolve it over time. Interaction between team members takes precedence over formal document-centric processes.

The use of tools such as versioning systems or compilers is obviously not being not criticized here. Under the premise that a part of the programming in MDSD is done via DSL, the generator replaces the compiler and there is no contradiction with agile development.

Working Software vs. Comprehensive Documentation

In project practice it is more important to deliver runnable software instead of good-looking documents such as requirements, concepts, architecture, design. In MDSD, the model is the source code. Diagrams are not just adornments, but a central artifact. The diagrams and the software will not drift apart and are always up-to-date, because the application is directly generated from the model.

The creation of runnable software is noticeably accelerated through MDSD because tedious, recurring implementation tasks are automated. We are not propagating a waterfall-like process in which the domain architecture is implemented first and then, in a second project phase, an application is built using it. A closer interlocking of both aspects allows timelier creation of runnable, while not necessarily complete, applications.

Customer Collaboration vs. Contract Negotiation

This aspect expresses the wish to allow the customer to participate as much as possible in application development. Particularly, a fast response to changing customer requirements should be possible in the course of the project instead of having a fixed contract right from the start. (See also the next section).

Here MDSD can have a considerable advantage over traditional iterative, incremental development. This is especially true if a non-technical DSL is applied that can be (re-)used to communicate with the customer, thereby shortening feedback cycles: a DSL is otherwise independent of whether MDSD is applied or not.

Responding to Change over Following a Plan

This valuation is about incorporating the (changing) requirements of the customer flexibly in the course of a project, instead of insisting on formally-defined requirements that are written down at the start and which may no longer be relevant for the customer. MDSD makes this procedure much easier:

- When domain-related requirements change the generative approach allows these changes to be implemented much faster and more consistently than in traditional software development.
- Technical aspects implemented by the transformations can be adapted in one place, and the change is automatically propagated in the entire application.

5.6.2 Agile Techniques

In this section we briefly address the interplay between MDSD and agile techniqes.

Pair programming is a technique that is mostly known from the Extreme Programming (XP) field, in which two developers share one terminal and implement the application together. The advantage is that errors are quickly detected, because one of the developers implements the details while the other has the overall concept in mind and recognizes errors. Of course this also works for MDSD. During modeling (depending on the DSL), developers and domain experts could even sit in front of a terminal together.

Another important technique is *test-driven development*, also known as *test first*. The idea here is to first implement tests, then develop the application against them, until all tests pass. In the context of MDSD, such an approach is of course also possible, in principle. However, due to the models' additional specification level, an even wider range of possibilities exists.We discuss these in detail in Chapter 14.

The fact that an application design is transformed into a standardized implementation is seen often as a restriction of the developer's freedom, as well as a restriction of their ability to respond to customer wishes. On the other hand, a significant part of the refactoring effort is focused on approximating the desired architecture with the implementation, especially in agile projects. In this respect, the codification of architecture in the transformations is only a logical next step in this train of thought. Refactoring as an agile technique can basically be applied to models, platforms, transformations, as well as of course to manually-implemented code.

Not only do we think that MDSD and agility are not opposing each other, we are even of the opinion that MDSD can help to scale agile techniques through the explicit codification of architecture and domain knowledge, as well as through the separation of domain architecture and application development.

Part II
Domain Architectures

In the first part of this book we came to know the practical side of MDSD, and in Chapter 2 we defined a domain architecture – the core concept of MDSD. The case study demonstrated what a domain architecture can look like in practice.

In this part of the book, we discuss the *construction* of domain architectures. In this context we address technical questions, rather than questions that relate to the development processes: the latter are detailed in Part III. A central question will be: 'Which engineering approaches can be recommended for finding DSLs?'

The next five chapters introduce techniques and best practices that are relevant for the development of domain architectures, beginning with metamodeling as the key to DSL definition. This is followed by the special role of target software architecture in the context of MDSD and the details of model-to-model transformations and code generation.

Many of the fairly technical questions with which a domain architect is confronted, such as those concerning code generation, are generic – that is, they are for the most part domain-independent. In other words, many problems can be solved with generic tools that can be reused in any, or at least related, domains. Chapter 11, on tool selection and architecture, gives you some background in that respect. The principles and best practices conveyed there are suitable for construction as well as for selecting MDSD tools, and are therefore not restricted to those developing MDSD tools themselves.

Finally, we take a deeper look into the MDA standard in Chapter 12.

6 Metamodeling

Metamodeling is one of the most important aspects of Model-Driven Software Development. Metamodeling knowledge is needed for dealing with the following MDSD challenges:

- Construction of domain-specific modeling languages (DSLs): the metamodel describes the abstract syntax of such a language (see Chapter 4).
- Model validation: models are validated against the constraints defined in the metamodel.
- Model-to-model transformations: such transformations are defined as mapping rules between two metamodels.
- Code generation: the generation templates refer to the metamodel of the DSL.
- Tool integration: based on the metamodel, modeling tools can be adapted to the respective domain.

This list helps to justify why we dedicate an entire chapter to this subject.

6.1 What Is Metamodeling?

Metamodels are models that make statements about modeling. More precisely, a metamodel describes the possible structure of models – in an abstract way, it defines the constructs of a modeling language and their relationships, as well as constraints and modeling rules – but not the concrete syntax of the language. We say that a metamodel defines the *abstract syntax* and the *static semantics* of a modeling language (see Chapter 4). Vice versa, each formal language, such as Java or UML, possesses a metamodel.

Metamodels and models have a class-instance relationship: each model is an instance of a metamodel. To define a metamodel, a metamodeling language is therefore required that in turn is described by a meta meta model. In theory, this abstraction 'cascade' can be continued ad infinitum, but in practice other steps are taken, as we will soon learn.

In the context of MDSD, the domain's DSL is defined by a metamodel. The concrete syntax – that is, the concrete form of the textual or graphical constructs with which the modeling is done – is conceptually irrelevant: it must merely render the metamodel in an unambiguous way. The distinction between abstract and concrete syntax is very important here, because the metamodel (and *not* the concrete syntax) is the basis for the automated, tool-supported processing of models. On the other hand, a suitable concrete syntax is the interface to the modeler – without it, no models could be created – and its quality decides what degree of readability the models have[1].

As a consequence of this decoupling, the metamodel and the concrete syntaxes of a DSL can maintain a 1:*n* relationship: the same metamodel can be realized by a graphical as well as a textual syntax.

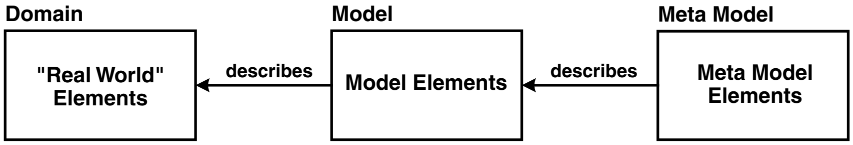


Figure 6.1 Relationship between the real world, model and metamodel

In principle models can be described in an arbitrary modeling language. Language selection should be made based on the language's suitability for the domain to be described. In real life, this decision is often determined by the question of whether or not practically usable tools are available for the modeling language, which means that today UML is used for modeling in many cases. It is therefore of particular relevance to look at metamodeling in the context of UML.

The *meta* relationship is always to be seen relative to a model. An absolute definition of the term *metamodel* does not make sense in theory, but in practice it is quite useful. For this reason, the OMG defines four metalevels. These are shown in Figure 6.2 and are described further in the following sections.

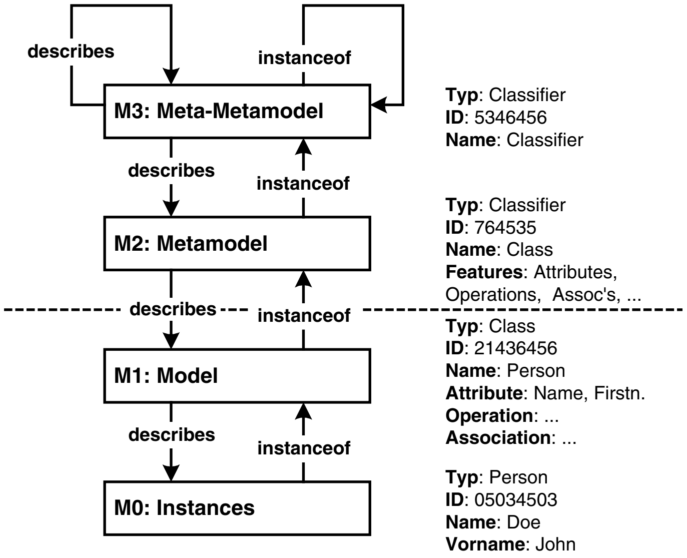


Figure 6.2 The four metalevels of OMG

[1] What has been said so far is not only true for models/modeling languages, but analogously also for programming/programming languages.

Below the dashed line we find ourselves on familiar ground as software developers. In M1, in the model, a class is defined. This class is given a name, *Person*, and a number of attributes, in this case *name* and *first name*. *Instances* of this class are created in M0, usually at program runtime: in the example in the figure, the person with the (internal) ID 050 34503, the last name *Doe* and first name *John* – or more precisely, the attributes *name* and *first name* have the value *Doe* and *John*, respectively, for this instance. During the instantiation of a class, therefore, values are assigned to attributes of the class. Note that a class can have more than one instance. The model (here: the class *Person*) is defined via a language – in our case UML – even though this is not shown in Figure 6.2.

We now move up one metalevel. In M2, the metamodel, the constructs that are used in the M1 model are defined. The elements of the M1 model are thus instances of the elements of the metamodel at the M2 level. Since we use classes in the M1 model, the construct *Class* must be defined in M2. This is actually the case in the UML metamodel[2].

The construct *Class* in the UML metamodel is now an instance of the meta meta element *MOF Classifier*. MOF classes are defined in M3. The meta object facility (MOF) is the OMG's meta meta model (see Chapter 12). The MOF serves to define modeling languages at M2, such as for example UML. The idea behind this is that UML will not remain the only modeling language, but that additional domain-specific and possibly standardized modeling languages will be defined that are based on the MOF. The MOF is also able to define non-OO modeling languages. We provide an example of this ability below.

There is no metalevel in the OMG model above the MOF – basically, the MOF defines itself.

Figure 6.3 shows a (simplified and incomplete) excerpt of the MOF.

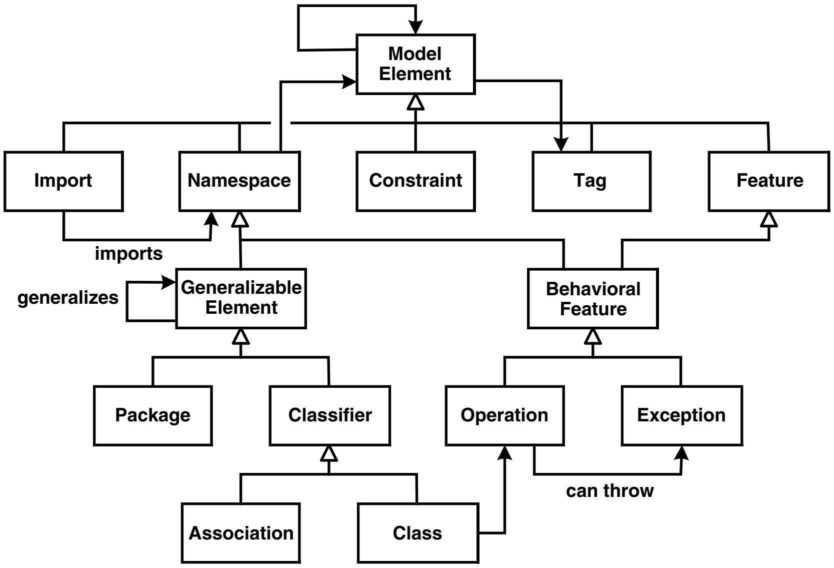


Figure 6.3 An excerpt from the MOF

[2] As we use UML as a language in M1, M2 must define the language UML – the UML metamodel is applied here as M2.

As the name MOF implies, this is a meta meta language based on the object-oriented paradigm. For this purpose, the MOF borrows the UML's class core, thus using the same concepts and the same concrete syntax.

Whenever we expand the UML metamodel, for example through a derivation of a metaclass *MyMetaClass* from *UML::Class*, we do this by means of the MOF. The inheritance relationship between the two metaclasses is the inheritance relationship as it is defined in the *MOF::Classifier* or its super class *MOF::GeneralizableElement,* respectively.

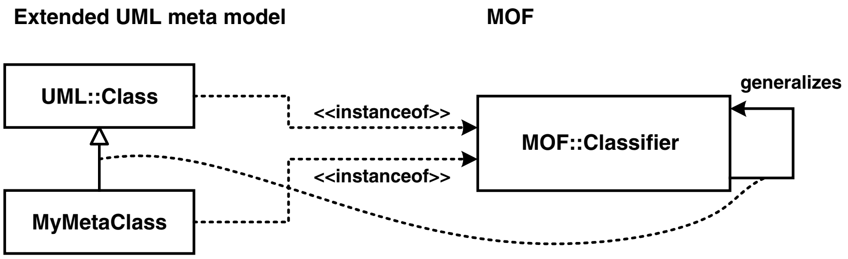


Figure 6.4 Metamodel expansion in relation to the MOF

6.2 Metalevels vs. Level of Abstraction

Models can have different relationships to each other. This chapter illuminates the *meta* relationship, which states that the metamodel defines the concepts with which a model can be created.

On the other hand, models can also be located on different abstraction levels, even though they are located on the same metalevel. Typically, transformations are used to map models at a higher abstraction level to models with a lower abstraction level. Each of the models is (inevitably) an instance of a metamodel. The metamodels of the two models are therefore different, yet the models as well as the metamodels can be found on the same metalevel. Figure 6.5 shows this.

6.3 MOF and UML

UML is an instance – an application – of the MOF. Various details must be considered.

First, UML existed *before* the MOF. UML was originally *not* formally defined – that is, it was defined purely verbally. The MOF was defined later to specify UML formally based on the MOF. The problems that emerged from this sequence were cured in later UML revisions, so that UML can now be called a MOF language in good faith.

The notation for MOF models is the concrete syntax of UML. Occasionally, this can lead to confusion. Formally, this problem can be solved through the specification of namespaces/packages for model elements, yet the potential for confusion remains.

It should also be observed that the MOF contains a number of model elements that are also present in UML. For example, both languages possess an element called *Class*. Even though the elements have the same name and often superficially describe the same feature, they are not identical – if only because they are located on different metalevels.

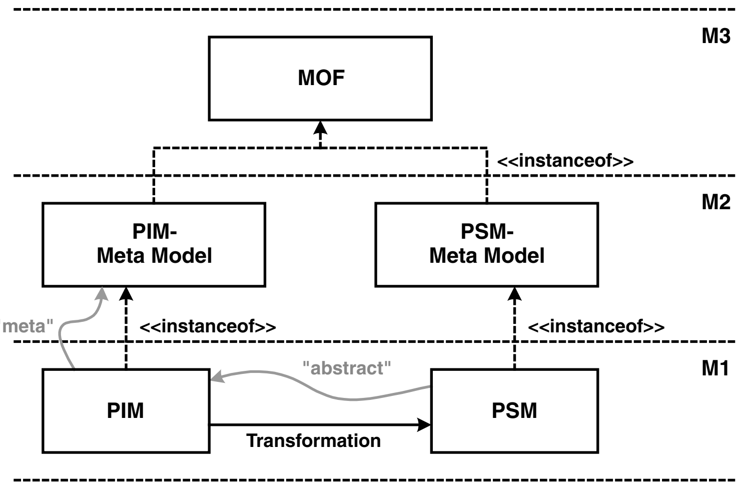


Figure 6.5 *Meta* versus *abstract*

6.4 Extending UML

In the context of software development, often one will not start by defining a completely new M2 language based on the MOF. It is more likely that one will start with the UML metamodel and extend it as needed. To carry out this extension, there are three options:

- Extension based on the UML's formal metamodel.
- Extension using stereotypes/profiles (by means of UML 1.x).
- Extension using stereotypes/profiles (by means of UML 2).

We look at each of these alternatives in the following sections. In practice, one would mainly use the stereotype/profile mechanisms, due to the number of available tools for the definition of UML-based metamodels.

6.4.1 Extension Based on the Metamodel

This type of extension expands the UML's metamodel. To this end we apply, as always in modeling, the language of the next-higher metalevel, which in this case is the MOF. Such an extension can take place within a tool only if the tool possesses an explicitly-represented, disclosed MOF-based metamodel.

To define, for example, one's own kind of class, you would create a new M2 class that inherits from the UML metaclass *UML::Class*. Figure 6.6 illustrates this process.

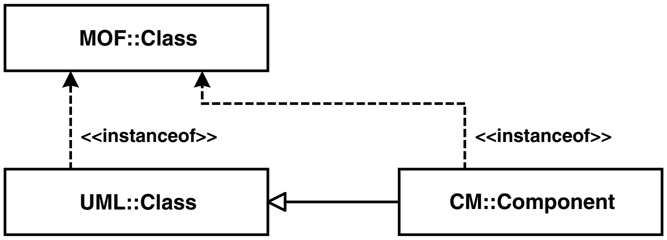


Figure 6.6 UML adaptation through extension of the UML metamodel

Here, a new language construct is defined – the *CM::Component*. This is a subclass of the *Class* element of the UML. As we explained in the previous section, an inheritance mechanism of the MOF is used here too, since after all we are dealing with a MOF model of the UML version extended by us here.

It is theoretically possible to assign a graphical representation – a concrete syntax – to each language element we define, as is illustrated by (e) in Figure 6.7. This is often impossible in practice, however, because the tool does not support it. Other types of representations can be used, most of them based on stereotypes.

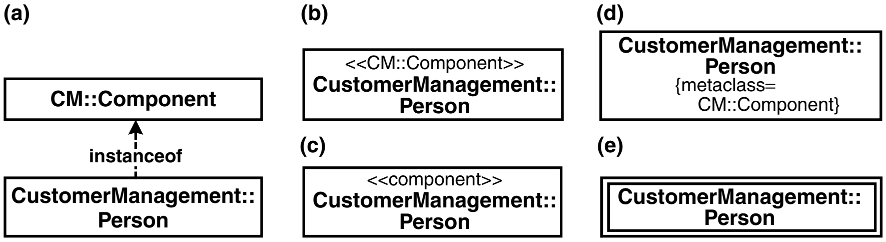


Figure 6.7 Representation of the metamodel expansion through stereotypes

Figure 6.7 (a) shows a *CustomerManagement::Person* class as a direct instance of the metaclass *CM::Component*. (b) uses the name of the metaclass as a stereotype, while (c) uses an abbreviation agreed by convention, (d) a tagged value stating the metaclass, and (e) an individual graphical notation. The approach (c) has proved to be the most practicable in real life, while (e) is a viable alternative if the tool allows this option.

The *CM::Component* cannot be distinguished from a UML class – apart from its formal type – because it neither adds nor overwrites any attributes and operations, and does not define constraints. This is not necessarily always true: we can define new attributes for our own metaclass. These will typically be represented by tagged values in the target models, as can be seen in Figure 6.8.

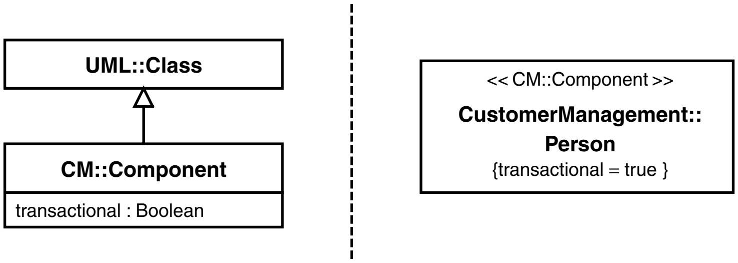


Figure 6.8 Tagged values as concrete syntax of metamodel attributes

The type of adaptation of modeling languages introduced here – the extension of the metamodel using the MOF – doesn't only work in a UML context, but also for all other MOF-based modeling languages,. The mechanism based on profiles, which is introduced further below, is in contrast restricted to UML, since it is defined as part of UML itself.

It is important to point out that there is no switching to another metalevel when the metamodel is extended via inheritance. Figure 6.9 shows this.

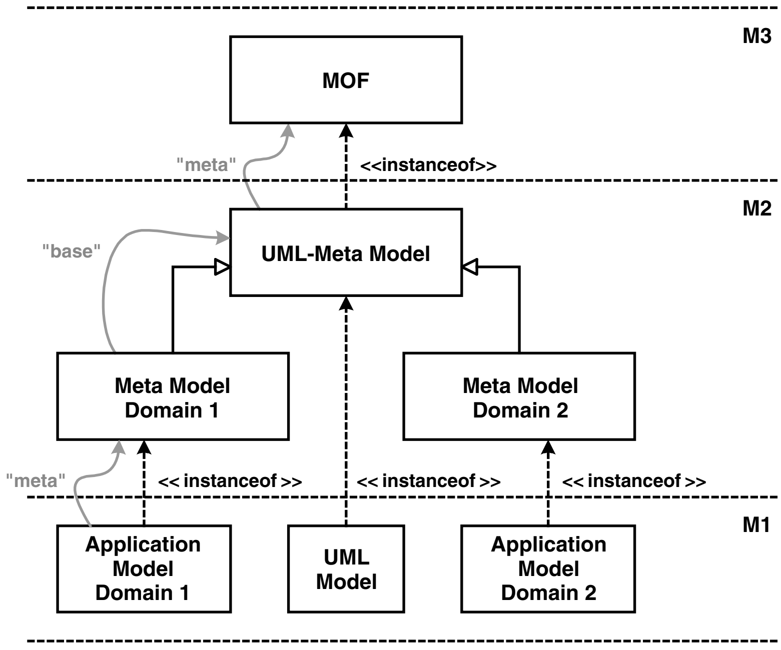


Figure 6.9 Inheritance inside the M2 layer

Figure 6.9 also shows that the prefix *meta* is in principle always relative to a model[3]. When a metamodel is extended, the origin is called the *basic metamodel*.

6.4.2 Extension With Stereotypes in UML 1.x

Extension with stereotypes is a UML-specific functionality, defined as part of the profile mechanism. This means that the UML *itself* is a way in which the UML metamodel can be extended to a certain extent, or, more precisely, be specialized without being required to using the means of modeling language definition provided by the MOF. One reason for this is probably that when UML was originally defined, the MOF did not exist, so some other means of extension had to be provided. So far this extension mechanism works with UML only, so other MOF-based languages must define their own extension mechanisms.

Figure 6.10 shows the definition of the stereotype *CM::Component*, including the tagged value *transactional*.

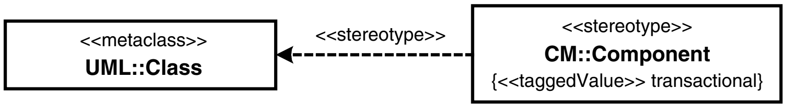


Figure 6.10 Definition of a stereotype in UML 1.x

It is important to note that the diagram in Figure 6.10 is formally an M1 model of the MOF hierarchy, since it is a UML model, and not a part of the UML's metamodel. Semantically, it is at the M2 level, because quite clearly a UML metaclass (*UML::Class*) is specialized here.

Serious limitations or this approach when compared to the metamodel's extension via MOF are that tagged values are not typed (all tagged values are *Strings*) and no new meta associations between existing metamodel classes or stereotypes can be defined. The advantage, however, is its usability in the field of generic UML tools.

6.4.3 Extension With Profiles in UML 2

With the definition of UML 2.0 the stereotype mechanism has been extended and placed in the context of a more comprehensive profile mechanism (see also Section 6.5 and Chapter 12). The concept of *extensions* is pivotal here. An extension is a new symbol, and thus a new language construct, of UML. It is rendered as a filled inheritance arrow, as shown in Figure 6.11.

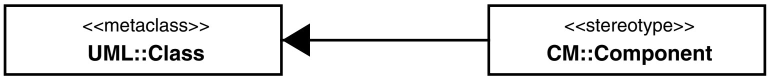


Figure 6.11 Definition of a stereotype in UML 2.x

[3] Levels 0–3 have fixed names only in the context of the OMG.

It should be emphasized that we are not dealing with inheritance, implementation, stereotypical dependency, or association here, but with a completely new UML language construct that is also defined formally in the UML metamodel.

A stereotype can have attributes. As in UML 1.x, these are rendered as tagged values in the model in which the stereotype is used (see Figure 6.12). From UML 2 onwards a tagged value can be assigned a type, thus all tagged values are no longer strings per se.

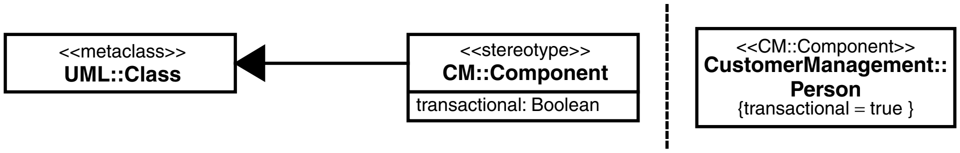


Figure 6.12 Tagged values in UML 2.0

Another difference between UML 2.0 and UML 1.x is that a model element can now have multiple stereotypes simultaneously. It then possesses the attributes of all stereotypes as tagged values[4].

6.5 UML Profiles

Profiles support adaptation or extension of UML to fit professional or technical domains. One might also say that UML is not a language, but a language *family*: in this case, UML profiles are elements – concrete languages – in this family. The objective is that UML tools and generators can process profiles like plug-ins: one first loads a specific profile, then modeling can take place based on the profile. To make this work smoothly in practice, a clear-cut separation between model, profile, transformations and tools is mandatory. For this purpose, the OMG defines a profile mechanism for the UML. (Here, too, we are dealing with a UML-specific mechanism.)

Principally, UML profiles consist of three categories: stereotypes, tagged values, and constraints[5]. Profiles can extend UML's valid constraints – that is, further constrain them – but cannot relax their restrictions. In UML 1.x the construct of the profile is only defined verbally. In UML 2.0, the concept of the profile based on the UML metamodel is defined formally. Here we also find a definition of the extension concept mentioned in the previous section.

Figure 6.13 shows the metamodel of the profile definition of the UML 2.0 specification – which itself can serve as an example of metamodeling. We omitted explicitly marking up the namespace for each element, since it's all part of the UML metamodel[6].

[4] Strictly speaking, tagged values are no longer tagged values, but the representation of the stereotypes' attributes. Since they still look like tagged values, though, they are still termed tagged values.

[5] UML 2.0 formally defines this a little differently – see below.

[6] In the interest of brevity, some constraints are left out.

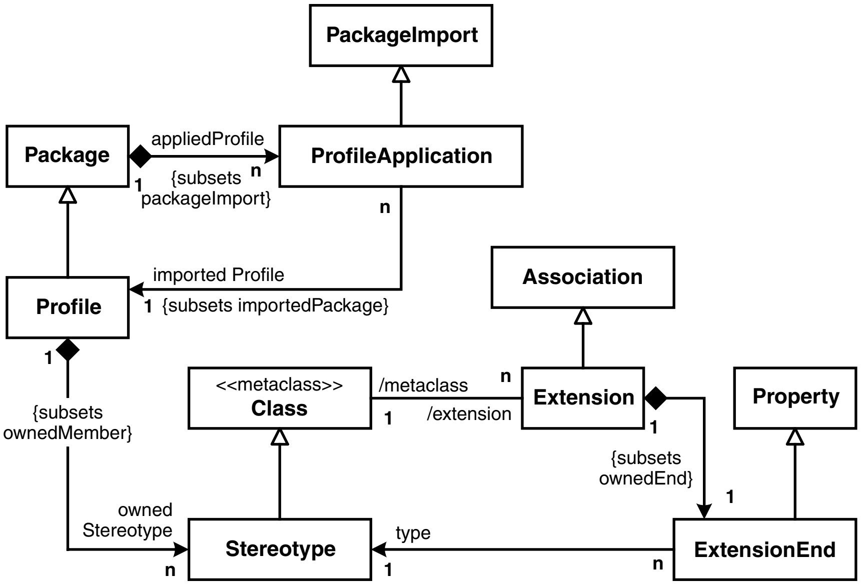


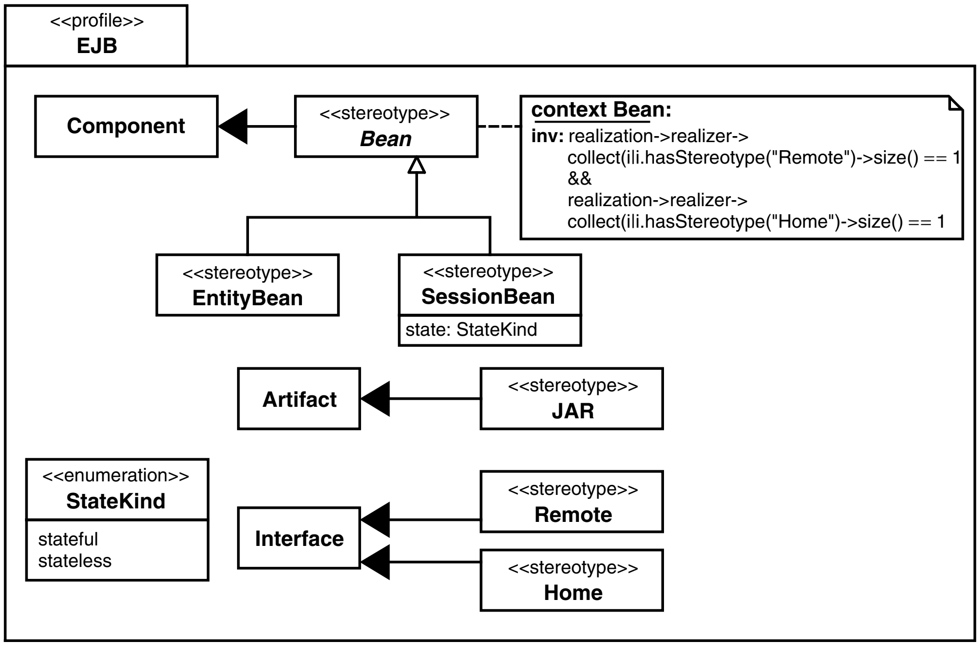
Figure 6.13 Metamodel of the profile concept

According to Figure 6.13, a *profile* is first defined as a specialization of a *UML::Package*. Packages can be profiled through the use of a *ProfileApplication*, a specialization of *PackageImport*. More loosely one could say that when a package imports a profile package, this means that the profile is applied to the importing package. A profile contains a number of stereotypes – a stereotype is a specialization of *UML::Class*. In this context, the extension (see Section 6.4.2) is a specialization of *UML::Association* in which one end of the *UML::Association* must reference a stereotype.

UML now offers the linguistic options for expressing profiles via UML as well as for notating its use with application models. Figure 6.14 shows the definition of an (extremely simplified) profile for EJB[7].

This diagram should be more or less self-explanatory after the explanations above. However, some interesting aspects should be mentioned in this context. On one hand, stereotypes can be *abstract*, which conceptually means the same as for abstract classes: they cannot be directly annotated to model elements; they merely serve as a basic (meta-) class for further stereotypes. Stereotypes can also inherit from each other. Constraints that are defined for stereotypes mean that these constraints must be valid for classes to which the stereotype is applied. In our example in Figure 6.14, this means that a *Bean* must implement exactly one *Remote* and one *HomeInterface*. Additionally, this example demonstrates how *Enumerations*, which are used only for typing a tagged value in this case, are modeled.

[7] The OCL that is used for the constraints in this model are explained later.



```
context Bean:
inv: realization->realizer->
     collect(ili.hasStereotype("Remote")->size() == 1
     &&
     realization->realizer->
     collect(ili.hasStereotype("Home")->size() == 1
```

Figure 6.14 A simple EJB profile

A profile is not independent. Instead it always depends on and uses a *reference metamodel*. This can either be the UML metamodel or an existing profile. The profile is unable to change or remove the existing definitions in the reference metamodel, but the profile mechanism is a well-defined back-door through which new constructs (stereotypes) and their properties (tagged values) can be added. The same is true for additional modeling rules (constraints) that further restrict the constructs' interplay, and are thus able to formalize the well-formedness of models of the specific language.

This can also serve as a basis for the adaptation of UML tools, so that the developer is alerted to profile-specific modeling errors as early as during modeling. As a rule, most of the currently-available UML tools are not as advanced as this yet. In many tools, modeling rules are still supported – if at all – by proprietary mechanisms such as scripts or plug-ins. Until this changes, the following options for dealing with profiles in practice are available:

- The constraints in the profile only serve documentation purposes: if necessary, they are merely notated non-formally.
- The formalization (implementation) of profile constraints is carried out via specific UML tool mechanisms.
- Testing for well-formedness is left to the MDA/MDSD generator, which can validate a profiled model independently of the UML tool, should this be required. In this case, the modeling rules would have to be 'taught' to the generator. If an OCL interpreter is used for this purpose, it is even possible to evaluate an OMG-conformant formal profile definition.

6.6 Metamodeling and OCL

OCL is the abbreviation for Object Constraint Language. This is a side-effect-free, declarative language for the definition of constraints (restrictions) such as modeling rules for MOF-based modeling languages. Constraints enrich models with additional information about the validity of model instances. Constraints are suitable for application at the M1 as well as the M2 levels.

Let's assume we have a UML model that contains an association between people and cars, as shown in Figure 6.15. A person can either have the role either of driver or passenger. While anyone can be passengers, drivers must be by definition at least eighteen years old and hold a driver's license. How can we express this in UML?

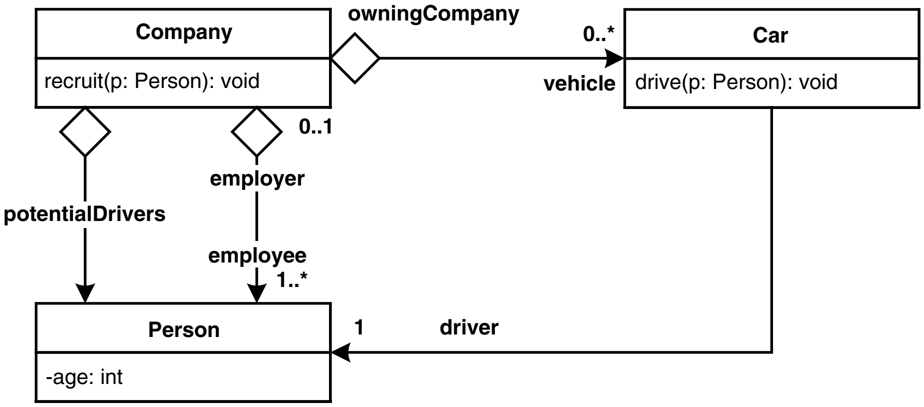


Figure 6.15 A sample model to illustrate OCL

Apart from the suboptimal option of defining a subclass of people called *AdultPersonwithDriver'sLicense* and its driver-association, the only other option is to use a constraint. In the following examples, constraints are described verbally and via OCL.

For all instances of *Car* it holds that drivers of a car must be at least eighteen years old (invariant).

```
Car
driver.age >= 18
```

For all instances of *Company* it holds that a company's potential drivers are all those employees who are older than eighteen.

```
Company
potentialDrivers = employees->select( age >= 18 )
```

For the operation *drive()* of the class *Car* it holds this it can only be called when no driver is seated in the vehicle and the person passed as the argument is older than eighteen (precondition). After the operation has been carried out, the person passed as the argument takes on the role of driver (postcondition).

```
Car::drive( p : Person )
pre : ( driver == null ) &&
      ( p.age >= 18 )
post: driver = p
```

For the operation *recruit()* of the class *Company* it holds that after the operation has been executed, the list of employees has grown by one, and the added person is now part of this list.

```
Company::recruit( p : Person )
pre : -- none
post: (employees.size =
```

As should be clear from these examples, constraints written in OCL are both more precise and more concise than free text. In particular, they are formally coupled with the model. OCL does have special meaning in the metamodeling context. This is because metamodels should be extremely precise and tool-processable: a constraint written in natural language can't be processed by a verification tool.

First and foremost, OCL constraints are modeling language-independent. This especially means that OCL constraints can be used at various metalevels. The example given above uses OCL in a concrete UML model, that is, at the M1 level. Here, it affects the instances of this model's elements: in general, a constraint in M_n affects M_{n-1}. OCL is particularly significant in the context of model-driven development, because it can also be used at M2, for example in the context of a metamodel extension. Figure 6.16 shows an extension of the UML metamodel with an OCL constraint.

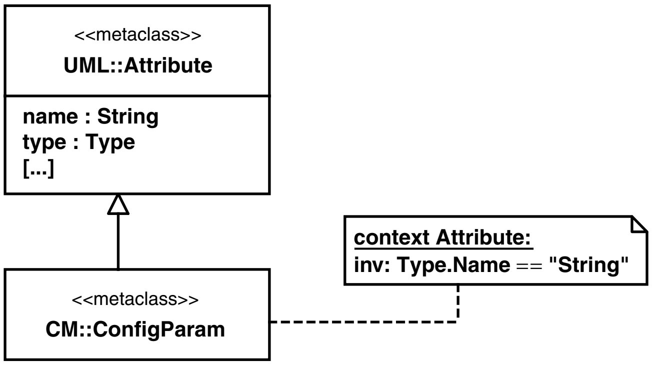


Figure 6.16 OCL constraints at the metamodel level

6.7 Metamodeling: Example 1

We now develop our own metamodel for demonstration purposes that has nothing to do with the UML, that is, one that doesn't extend the UML metamodel. For our example, we are going to use the feature models known from generative programming and the FODA method ([EC00], [FODA]). This example is introduced in more detail in Section 13.5.

Figure 6.17 shows the metamodel of such feature models. Please note that this is the metamodel of the feature *model*. We will not discuss its graphical representation as a diagram here.

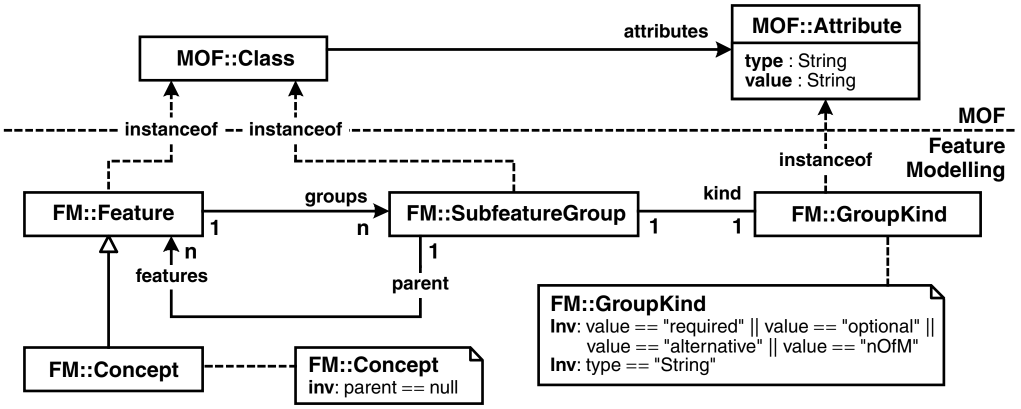


Figure 6.17 A metamodel for feature models

We first define a *Feature* as an instance of *MOF::Class*. A feature can have a number of subfeature groups. A *SubfeatureGroup* is also a *MOF::Class* and contains various *subfeatures*. A subfeature group has a kind, which can be *required, optional, alternative* or *n-from-m,* modeled using the attribute *kind*. Here, *GroupKind* gets the attributes *Type* and *Value* through inheritance from the super metaclass *MOF::Attribute*. Alternatively, one could have also defined *SubfeatureGroup* as an abstract metaclass and the various kinds as concrete subclasses.

The diagram in Figure 6.18 shows an excerpt of the example-feature model in Section 13.5.3 as a UML object diagram based on the metamodel we just defined. This diagram shows very clearly why it is important to use a suitable graphical notation, in this case that of the feature diagrams. This is much more readable and easier to create than a (theoretically adequate) UML object diagram. The acceptance of domain-specific modeling is often mainly a question of the suitable graphical notation, and of matching tool support.

Feature models can be enriched by further information. For example, one can determine whether a feature is considered *final* or whether possibly additional features may be added, if necessary in connection with a new *SubfeatureGroup*. In the latter case, such a feature is called *open*. The metamodel can easily be extended, and its rendering in an instance diagram is obvious, as Figure 6.19 indicates.

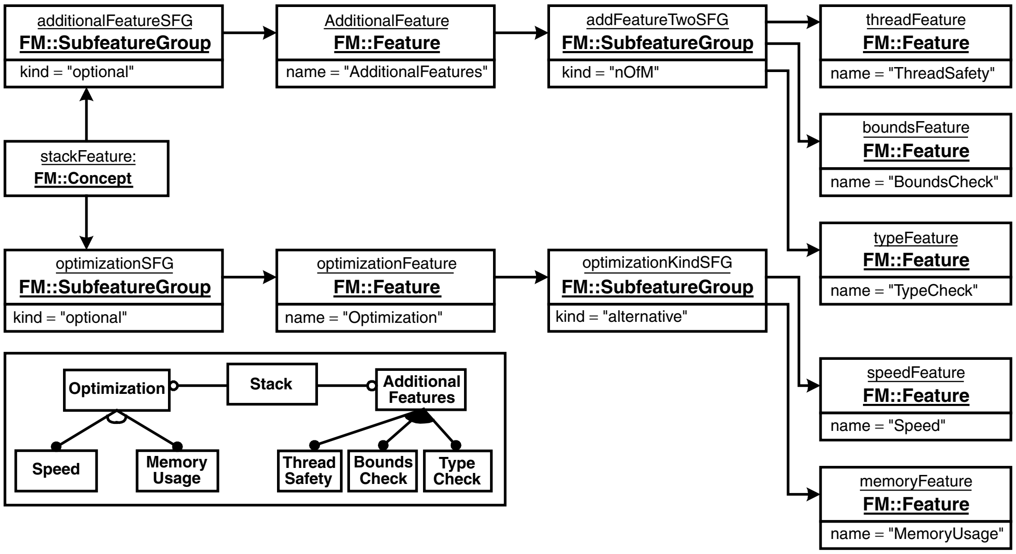


Figure 6.18 Feature model visualized using the concrete syntax of UML object diagrams (the affected part is shown in a feature diagram in the lower left corner)

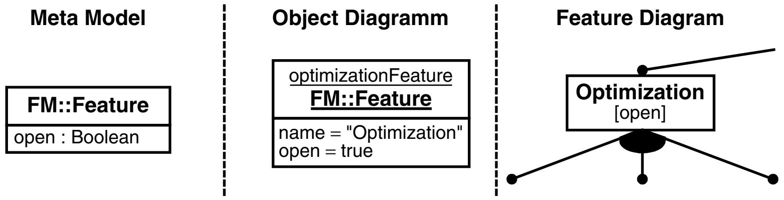


Figure 6.19 Metamodel and concrete syntax

The use of feature models is discussed in depth in Section 13.5.3.

6.8 Metamodeling: Example 2

An extremely simplified component infrastructure [VSW02] usable for small devices and embedded systems (see Chapter 16 and [Voe02]) will serve as another example for metamodeling. A central ingredient of applications based on this infrastructure are – obviously – components. During architecture definition, it makes sense to define what a component is, which is why we start with the definition of a metamodel for components of this infrastructure[8].

Figure 6.20 shows a simple example of a concrete model that uses the component concept. It shows component dependencies in a mobile phone SMS messaging application. First, we want to express the fact that a component can offer a number of services that are defined as *provided ports*. A *provided port* is associated with an *interface* that defines the available operations. Furthermore, a component must convey which resources it needs. This is accomplished by assigning a *required port* to the component. This port, too, has an *interface*. In this case the interface specifies which operations the component *requires* from other components.

In addition, a component has a number of *configuration parameters*. To simplify matters, these are attributes of the component class that must be of the type *String*, as they are read from a configuration file at system start-up.

Finally, there are special types of components that only use services and don't offer any: applications.

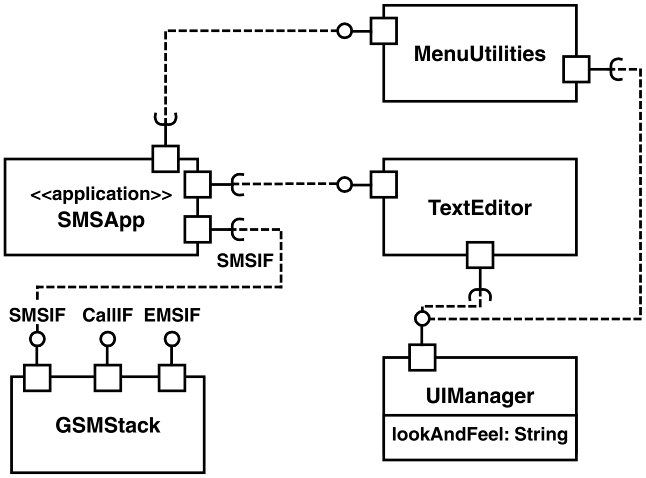


Figure 6.20 An example of a simple component-based system

The example in Figure 6.20 features an application *SMSApp* that defines three required ports. These are linked to interfaces that define the respective other ports. For example, the service interface of the *TextEditor* component is needed for the user to input an SMS. The *TextEditor* as well as the *MenuUtilities* need the *UIManager* to be able to access the screen. Figure 6.21 shows the metamodel of this architecture.

This metamodel formally expresses what we described above in words, at least in part. The coupling of an instance diagram (the SMS application shown in Figure 6.20) with the metamodel is accomplished via stereotypes and graphical notations: the ports are – following UML 2.0 – modeled as small rectangles on the component's edge. Attributes of such components are by definition configuration parameters. As components, applications are assigned the stereotype *«Application»*.

[8] Since we map the concepts of the target architecture in the metamodel, this is an example of architecture-centric, model-driven development.

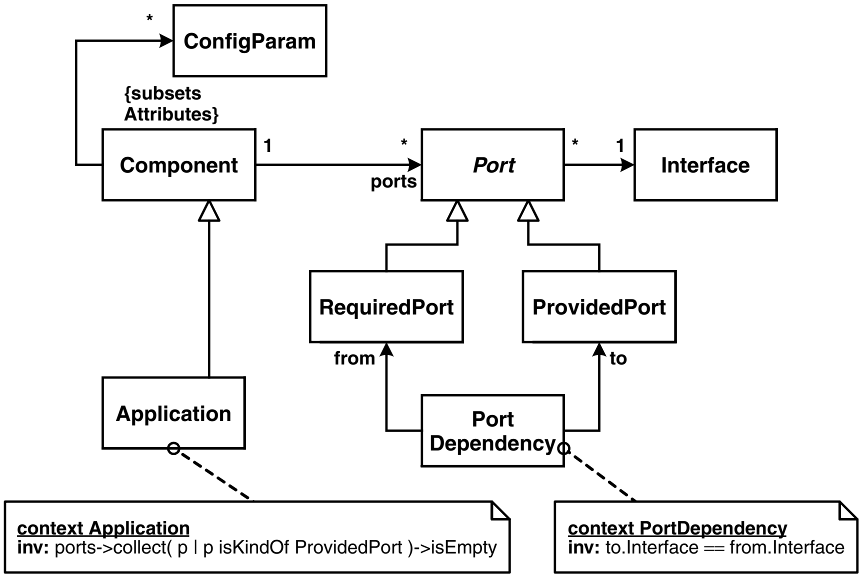


Figure 6.21 Metamodel for the description of components

To avoid having to create a completely new metamodel from scratch, the UML metamodel will serve as a basis for our own. We remember the statement ‘A configuration parameter is an attribute of type *String*’. In slightly different words: the metaclass *Attribute* occurs in the UML metamodel. It has an attribute named *Type*. We are merely saying that the metaclass *ConfigParam* is a subclass of *Attribute*, whose attribute *Type* must have the value *String*. The diagram in Figure 6.22 illustrates this, as well as the other consequences of basing our metamodel on the UML metamodel. Note the use of OCL for defining the necessary constraints.

We can now proceed analogously for the other elements of our metamodel. Figure 6.22 shows the result, while we introduce the namespace, respectively the package CM (for *Component Model*).

What is the actual benefit of this explicit metamodeling? As always, modeling only makes real sense if the models don’t end up collecting dust in a drawer: they must be usable in software development, true to the MDSD principle. This applies to metamodels too, of course. These should be implementable and support the further development process. Therefore, it is important not to just ‘draw’ a metamodel in the form of a diagram, but to adapt the development tools using the metamodel too.

Effective, domain-specific modeling can only work if a suitable modeling language is available for the domain to be modeled and this language is ‘understood’ by the development tools.The aspects listed at the beginning of this chapter – *model validation, transformation, code generation* and *tool adaptation* – are relevant here.

We next take a closer look at the first of these aspects, model validation. The remainder are illustrated in the case study in Chapter 16, which expands the component example featured above.

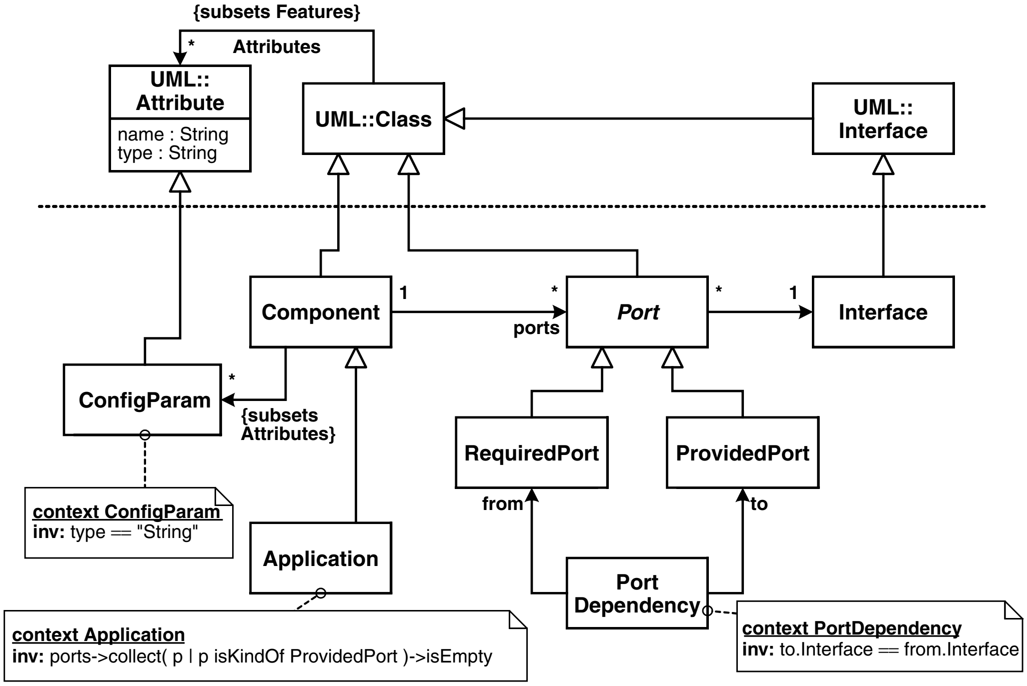


Figure 6.22 Component metamodel connected to the UML metamodel

6.9 Tool-supported Model Validation

Tool support for metamodeling varies widely. It is possible to distinguish between the following alternatives:

- *No support*. Most UML modeling tools offer hardly any support for metamodels. This is not meant as negative criticism – they are simply not made for this purpose. They are implicitly based on the UML metamodel, which is unchangeable. Of course, this leaves the option of coupling a model to a metamodel via a stereotype, yet no further support (or validation) is provided. Practically all widely-used UML tools fall in this category. However, there is a slow yet noticeable tendency toward growing support for UML profiles.
- *Separate tools*. Some tools are applied *after* a model has been created with a normal UML tool. Typically, the model is exported from the UML tool using XMI (XMI is an XML mapping for MOF, see Chapter 12) and further processed on this basis. Such tools include model validators, transformers, and code generators – almost anything that covers the full range of these tasks, often not limited to UML/MOF – and can even handle any modeling language. One example of this category is the Open Source generator openArchitectureWare described in Chapter 3.
- *Integrated (meta-)modeling tools*. Other than normal UML modeling tools, integrated (meta-)modeling tools are actually internally based on a metamodel. With the help of the

> tool, the user can not only adapt the metamodel, but can also create new models based on this metamodel. The tool will then adapt its interface and ensure that only valid models can be created. In most cases, validation takes place in real-time, that is, during input. Examples of such tools are MetaEdit+ [MC04] or GME [M. Völter, A. Schmid, E. Wolff, Server Component Patterns, John Wiley & Sons, 2002].

Most common is a combination of a UML tool and a separate generator/validation tool. Unfortunately, integrated metamodeling tools are still largely ignored by the market.

Let's now look at model validation via openArchitectureWare. Its functional principle is illuminated in Figure 6.23.

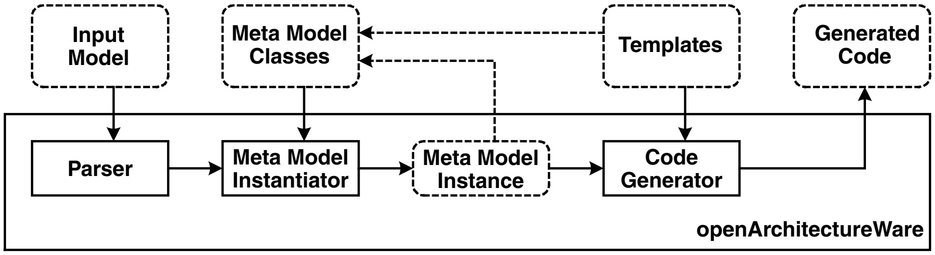


Figure 6.23 Functional principle of the *openArchitectureWare* generator

The generator uses any model as its input data. This is parsed by the parser, and the resulting parse tree is then instantiated by the metamodel instantiator using the configured metamodel. The input (model) format is interchangeable, because different parsers can be used in the generator. In our example, XMI is used, as in many cases. After instantiation of the metamodel, the model is available as an object graph of Java objects in the generator's memory. The object's Java classes correspond to the metaclasses of the metamodel. The actual code generation via templates can now take place, as we explained in our first case study in Chapter 3. We said above that metamodeling is, after all, a means for defining the 'language' available to the modeler. This especially includes the definition of modeling rules and the respective validation of concrete models.

Let's return to our component example above. The generator we use possesses an explicit, configurable metamodel. This is implemented in Java. The principle has been explained in detail in Section 3.2. Thus it should be clear what a metamodel adaptation looks like: we create a subclass of the corresponding metaclass and configure in the generator that instances of the new metaclass are mapped to the newly-implemented metaclass in the model. Here the example for *ConfigParam*:

```
package cm;
public class ConfigParam extends Attribute {
}
```

Using a configuration file (not shown here), we tell the generator that all UML attributes with the stereotype «*ConfigParam*» are in reality configuration parameters, which is why it should instantiate the subclass *ConfigParam* instead of *Attributes*. From the generator's perspective, this is not a problem, because as always in OO programming, an instance of a subclass can be used if a variable is typed with the superclass – polymorphism.

So far this class *ConfigParam* is not of much use to us, especially since we haven't contributed much to model validation at this point. *ConfigParam* is for example missing the constraint that the type of a *ConfigParam* always has to be *String*. To check such constraints, all metaclasses possess an operation *CheckConstraints* that is called by the generator once the *entire* metamodel has been instantiated. This is the primary place where model validation takes place. If this operation detects a problem it throws a *DesignError-Exception* that is then reported to the developer, indicating that the processed model is not consistent with the metamodel. This is the code for *CheckConstraints* of the class *ConfigParam*[9]:

```
public String CheckConstraints()
              throws DesignError {
  if ( !Type().Name().toString().equals("String") ) {
   throw new DesignException(
             "ConfigParam Type not String" );
  }
  return super.CheckConstraints();
}
```

To gain a better understanding of what is happening here, it is helpful to look at the UML metamodel used by the generator and extended by *ConfigParam*, as shown in Figure 6.24.

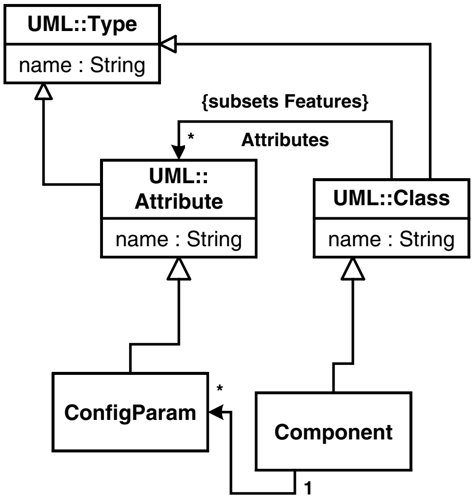


Figure 6.24 *ConfigParam* excerpt from the metamodel

[9] In this example, *Type* and *Name* are attributes of the metaclass *Attributes*. Unfortunately in this case the generator uses attributes spelled with capitals at the beginning of a word, which is a little confusing, but outside our control.

Since we are operating in the context of the class *ConfigParam*, the expression *Type()* provides the instance of the *UML::Type* object by following the inherited *Type* association of the class *Attribute*. The type has an attribute *Name* of the type *String*, which contains the name of the type. Note that the implementation of the constraint does not happen declaratively with OCL, but operationally via Java. The integration of an OCL/Java compiler is possible here, and will certainly happen in the near future in the context of the openArchitectureWare Open Source project.

In the same manner, we now proceed to create metaclasses for *Component*, *ProvidedPort* and *RequiredPort*. A few examples follow.

The metaclass for *Component* can e.g. look something like this:

```
public class Component extends Class {

  public ElementSet Port() {
    // return all ports of the component
  }

  public ElementSet RequiredPort() {
    return Util.filter( Port, RequiredPort.class );
  }

  public ElementSet ProvidedPort() {
    return Util.filter( Port, ProvidedPort.class );
  }

  public void CheckConstraints() {
       Util.assert( Operation().size() == 0,
         "Component must not define operations by itself" );
  }
}
```

The helper function *Util.filter()* filters a number of objects (here, the ports) for a specific metaclass. For example, the operation *ProvidedPort()* returns all ports that are actually provided ports. Note also the operation *CheckConstraints()*, which can be used for implementing invariants of the metamodel.

We can now look at the metaclass *Application*. This is a special kind of component that is not allowed to have *ProvidedPorts*.

```
public class Application extends Component {
  public void CheckConstraints() {
    Util.assert( ProvidedPort().size() == 0,
      "Application must not have any provided"+
      "ports, only required ports are allowed." );
  }
}
```

Here, too, *CheckContraints()* is used to guarantee that an application has no provided ports.

6.10 Metamodeling and Behavior

Behavior in the context of metamodeling is interesting in two respects. On one hand behavior can be hidden in the metamodel's meaning, while on the other one can use metamodeling to make behavior modeling explicitly accessible, for example in the form of activity or state diagrams.

Here, we will focus on the former scenario. We'll illustrate this using the familiar component example: let us assume we require each component to have an operation *init()*. This is realized most easily if we define an interface that contains this operation and that also require that all instances of the metaclass *Component* must implement the interface.

A simple calculator serves as an example, as is shown in Figure 6.25.

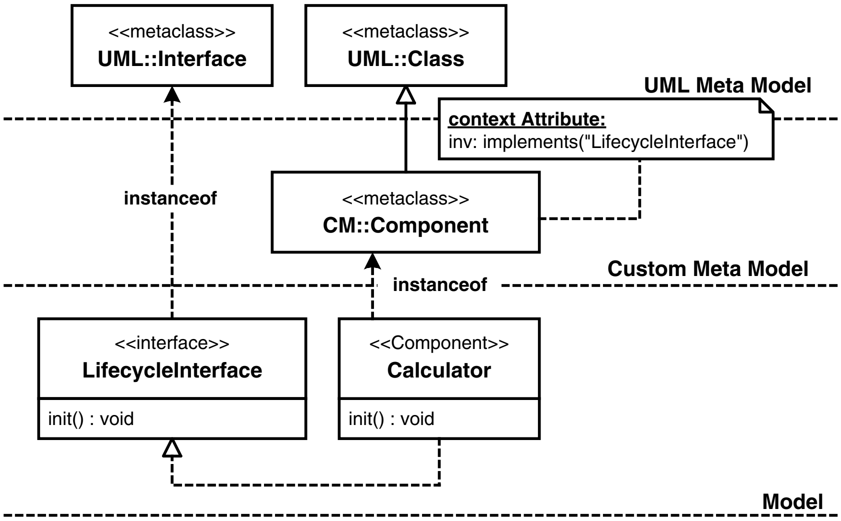


Figure 6.25 An example of 'All components must implement a specific interface'

The question is, what happens in the *init()* operation? For example, one can check whether links are available for all *RequiredPorts*. These links[10] are created by the *Container*. For this purpose, the component implementation offers a corresponding *set* operation for each *RequiredPort*, which is called by the container. The implementation of these operations saves the reference to the component that provides the *ProvidedPort* for the respective *RequiredPort* in an attribute. When *init()* is called by the container, the component instance expects these links to be present, that is, the corresponding attributes must no longer be *null*. The algorithm to verify this is as follows:

```
foreach r:RequiredPort {
  if (Attribute with the name of the port  == null ) {
    ERROR!
  }
}
```

[10] References are instances of associations.

This is a behavior that is not programmed by the user, but is instead implicitly determined by the architecture’s guidelines. This has the following effects:

- The programmer who creates an application doesn’t have to deal with it.
- In this case, the model validation is limited to making sure that each component implements the *LifecycleInterface*. This happens as explained above. The behavior within the method is not validated, because the implementation code (see below) can be generated automatically.
- If desired, one can also specify this behavior at the metamodel level, for example via sequence diagrams, action semantics (see Chapter 12), or – in this case – also using a constraint that states that all resource attributes must not be *null* once the operations have been executed.
- In the course of code generation, the implementation code for such operations can be generated directly. All information required for generation is present at generation time. The following section gives an example of the procedural realization of the constraint described above:

```
«DEFINE InitOperation FOR Component»
  public void init() throws IllegalConfiguration {
  «FOREACH Operation o IN RessourceInterface»
    if ( «o.NameWithoutSet» == null ) {
      throw ( new IllegalConfiguration(
        "Resource «o.NameWithoutSet» not set!” ) );
    }
  «ENDFOREACH»
«ENDDEFINE»
```

By the way, it is noteworthy that large parts of application’s behavior are often really behavior that is defined by the architecture. Among these are persistence, workflow, or remote proxys. All these aspects can easily be generated completely. For more details on modeling behavior in DSLs, see Section 8.1.3.

6.11 A More Complex Example

This section contains a more complex example of metamodeling. We are dealing with a part of the ALMA telescope here[11]. ALMA [ALMA] is an international astronomy project that pursues the goal of building an array of fifty radio antennas in the Atacama Desert in Chile. Several international organizations participate in this project: ESO, IRAM, MPI, NRAO. The fifty antennas are all connected via computer as a radio interferometer to achieve much higher resolution than is possible with a single antenna. To vary the telescope’s resolution, the positions of all fifty antennas can be physically changed using fork lift trucks.

[11] We thank the European Southern Observatory (ESO) for their kind permission to let us use this example here.

Naturally, such a project requires a fairly elaborate software infrastructure. This consists of:

- Real-time parts for steering the antennas, implemented in C++ and CORBA.
- Job definition scripts, implemented in Python.
- High-performance calculating modules for correlation and post-processing of digital images, implemented in C++.
- A 'classic' IT infrastructure, implemented in Java, for definition of the projects, data management, and remote access to the telescope infrastructure – the telescope is located at a height of 5,000 meters in Chile, while the scientists do their work from home over the Web.

Many of the system's data structures are needed by several of these subsystems. Due to the many non-functional requirements, the data structures must be available in different representations: XML for storage and remote transport, CORBA structures in the telescope control system, as well as some astronomy-specific formats for more efficient processing of raw data.

It was therefore decided to define the data structures with UML and to generate the various other artifacts from that[12]:

- XML schemata
- Wrapper classes for XML as well as (de-)marshalers in various languages (C++, Java, Python)
- Converters for the proprietary data formats
- HTML documentation for the data model

6.11.1 The Basics

We must first differentiate between *Entities* and *DependentObjects*. Entities have their own ID and can be searched based on several properties. An Entity can be subdivided. Its parts are DependentObjects. These do not have an identity of their own and cannot be searched – only the possessing entity knows them and references them. Parts can contain further parts. Figure 6.26 shows two examples:

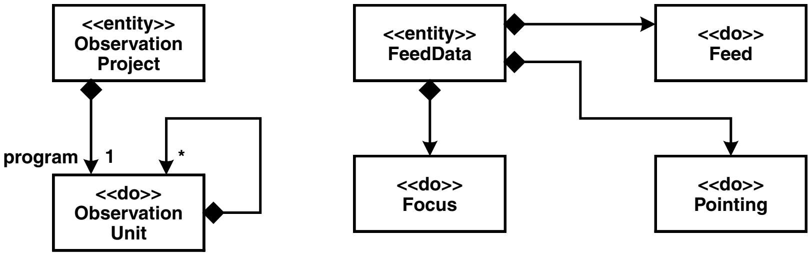


Figure 6.26 An example of *Entities* and *DependentObjects*

[12] For any astronomers amongst our readers: of course, the example is somewhat simplified.

An *ObservationProject* contains multiple *ObservationUnits*. These form a tree whose root is referenced with *program* by the *ObservationProject*. In the other example, observation data is shown. Without wishing to go into too much detail here, one can see that the *FeedData* consists of various substructures.

After coupling the metamodel to UML (that is, extending the UML metamodel), the metamodel for such models looks like Figure 6.27:

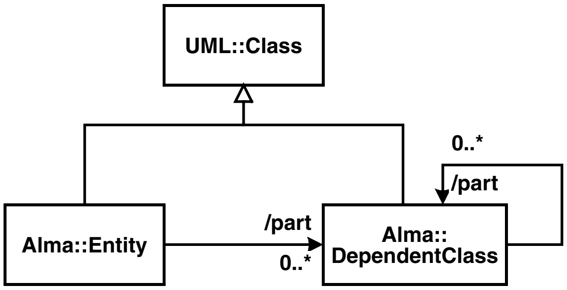


Figure 6.27 Metamodel for *Entities* and *DependentClasses*, coupled with the UML metamodel

6.11.2 Value Types

Other distinctions of the data exist in the data model. Specific information, such as a star's position in the sky, are neither *DependentObjects* nor are they primitive types. For this reason, we introduce *ValueTypes*. *ValueTypes* have no identity: they consist of only their value. Two *ValueType* instances of the same value are considered identical. As a convention, it is defined that the attributes of *Entities* or *DependentObjects* can only be primitive types or *ValueTypes*. The reason for this is that these values occur repeatedly all over the system. An example for the use of *ValueTypes* is shown in Figure 6.28:

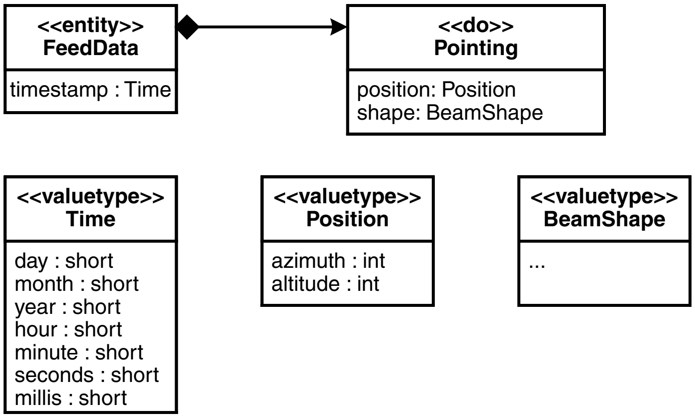


Figure 6.28 An example of the modeling and usage of *ValueTypes*

The metamodel is expanded accordingly. Since *Entities* as well as *DependentObjects* and *ValueTypes* have the same restrictions on their attributes, a corresponding abstract metaclass *AlmaAbstractClass* is introduced. This is common practice in object-oriented programming and is used here at the metalevel, as Figure 6.29 shows.

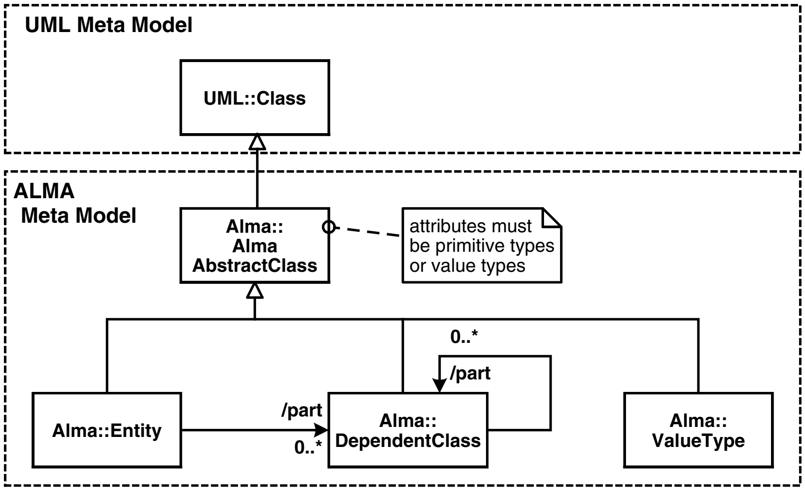


Figure 6.29 Metamodel with *AlmaAbstractClass* factored out

In this case we agreed to write the constraints in natural language rather than in OCL, because the generator requires manual programming of the constraints in Java anyway. The constraint for the attribute types could be described as an OCL constraint as follows:

```
context AlmaAbstractClass
inv: attribute->forAll( a |
          (a.oclIsKindOf(ValueType) ||
           a.oclIsKindOf(primitiveType) ) )
```

6.11.3 Physical Quantities

Since ALMA is, after all, a physical measurement instrument, the data it works with involves lots of physical quantities. It therefore makes sense to provide physical quantities explicitly as such in the metamodel. Physical quantities possess both a value and a unit, such as '10 arcsec' – *10* is the value, *arcsec* the unit. Various quantities have certain well-defined units and value ranges. For example, angles have the units *degree* or *arcsec* (arcsecond). Distances are measured in *mm, cm, km*, and *pc* (parsecs). All these aspects have to be reflected in the model. There are different options for visualizing this information in the model – we decided to use the one shown in Figure 6.30. Again, *angle* and *distance* serve as examples here.

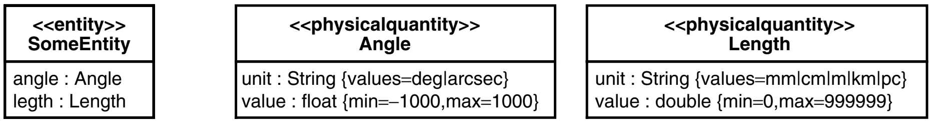

<<entity>>
SomeEntity
angle : Angle
legth : Length
<<physicalquantity>>
Angle
unit : String {values=deg|arcsec}
value : float {min=–1000,max=1000}
<<physicalquantity>>
Length
unit : String {values=mm|cm|m|km|pc}
value : double {min=0,max=999999}

Figure 6.30 Definition and use of physical quantities

The units are laid down in an attribute *unit*, which must be of type *String*. The value must be of type *int, long, float*. or *double*. The list of valid values for the *unit* attribute is given via tagged values: we use a list divided by '|'. Physical quantities can be used like *ValueTypes*, so they must also appear as attributes of an *AlmaAbstractClass*.

We next want to develop the respective metamodel. First, *PhysicalQuantity* is a subclass of *ValueType* (this should be clear after the discussion above) – *PhysicalQuantities* are a special kind of *ValueType*. Look at Figure 6.31 for the metamodel – for reasons of simplicity, we have again formulated the constraints in plain English.

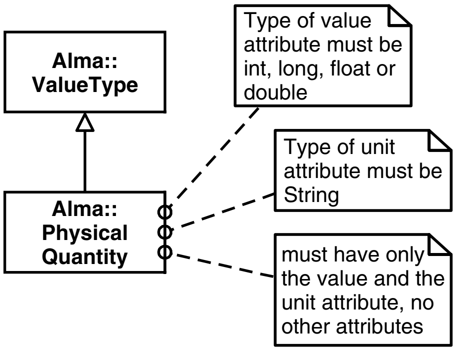

Alma::
ValueType
Alma::
Physical
Quantity
Type of value attribute must be int, long, float or double
Type of unit attribute must be String
must have only the value and the unit attribute, no other attributes

Figure 6.31 Metamodel for physical quantities

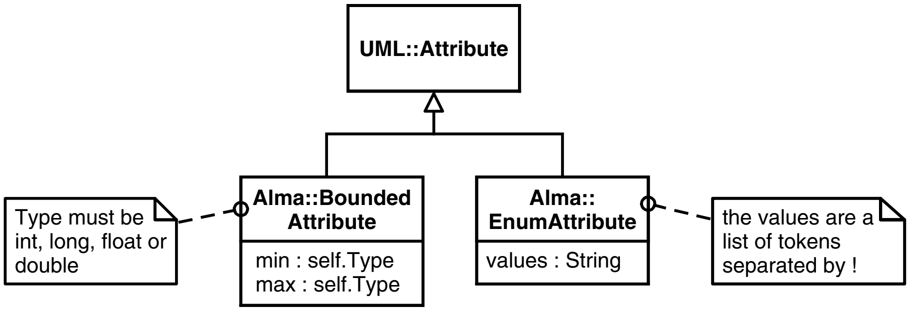


Figure 6.32 Special kinds of attributes

BoundedAttribute must be of the type *int, long, float*, or *double*. Two attributes *min* and *max* are also defined. These attributes of the metaclass appear in the model as tagged values, and are actually characteristics of the physical quantity defined in the model. The minimum and the maximum value are of the same type as the attribute itself. *EnumAttribute* can have any type. The tagged value *values* defines the valid values of the type to be defined.

However, the correlation between the physical quantity and the two new metatypes is still missing. Figure 6.33 shows this.

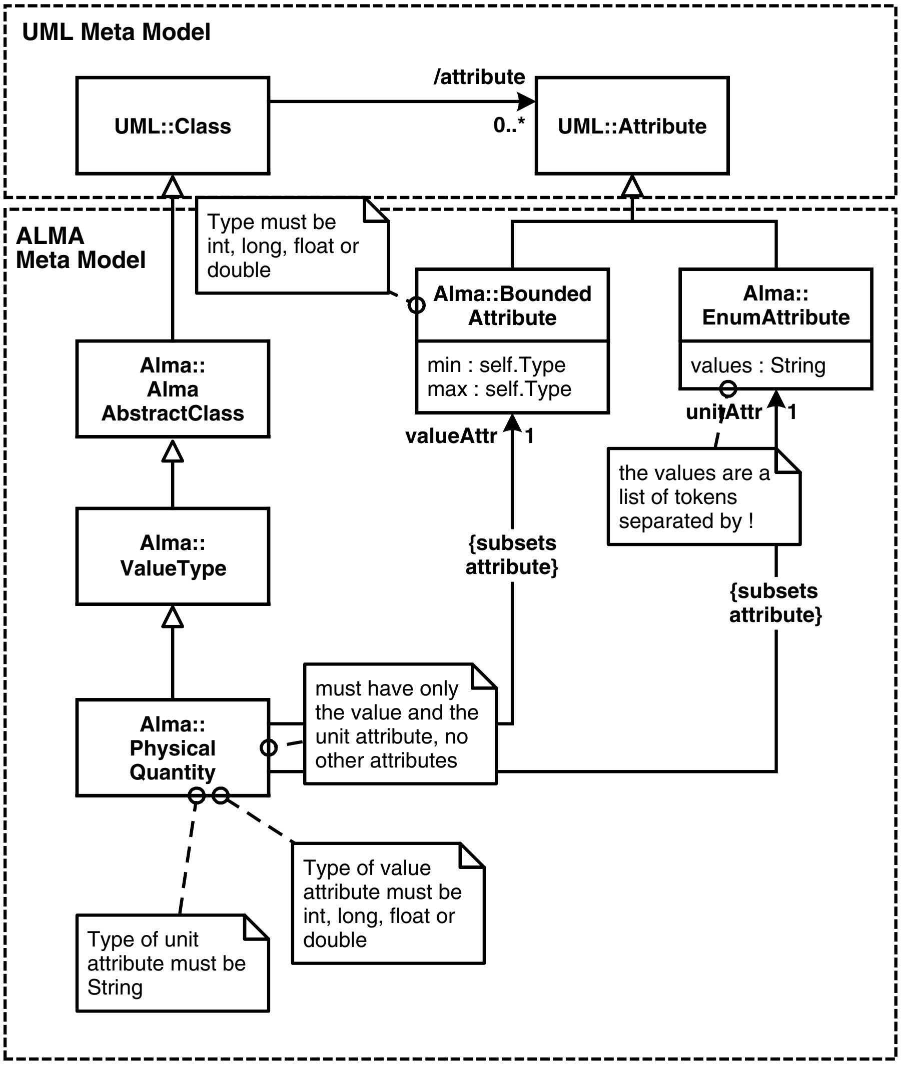

UML Meta Model
UML::Class
/attribute
0..*
UML::Attribute
ALMA
Meta Model
Type must be int, long, float or double
Alma::Bounded Attribute
min : self.Type
max : self.Type
Alma:: EnumAttribute
values : String
Alma:: Alma AbstractClass
valueAttr
1
unitAttr
1
the values are a list of tokens separated by !
Alma:: ValueType
{subsets attribute}
{subsets attribute}
must have only the value and the unit attribute, no other attributes
Alma:: Physical Quantity
Type of value attribute must be int, long, float or double
Type of unit attribute must be String

Figure 6.33 Excerpt from the complete ALMA metamodel

6.12 Pitfalls in Metamodeling

This section presents a few tips and tricks and reveals some of the pitfalls in metamodeling that particularly concern UML:

- One often reaches a point in metamodeling where it is no longer obvious which notation must be used.
- Accidentally finding oneself on the wrong metalevel.

In general, asking the central question of how the metamodel could be implemented in a programming language can prove useful. This view can, if reversed, give us hints for revealing which notation is the correct one.

6.12.1 Interfaces

Problem: You wish to express the fact that instances of a metaclass *Entity* (that is, all *Entities*) must implement a certain interface.

Correct solution: The set of an implemented interfaces of an *Entity* must contain *SomeInterface*. This can either be expressed via an OCL constraint, or by *subsetting* the respective meta-association (see Figure 6.34).

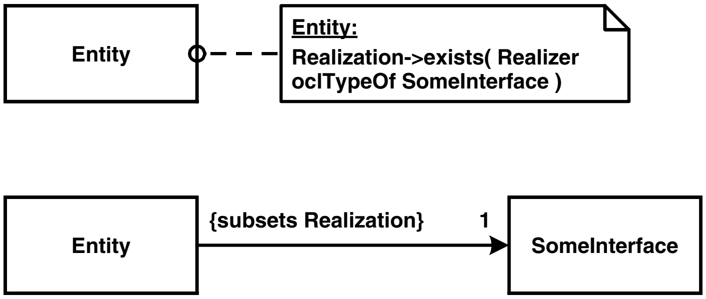


Figure 6.34 All *Entities* must implement a certain interface (correct)

Incorrect solution: Figure 6.35 shows that the metaclass *Entity* implements the interface *SomeInterface*. This is not the same statement as the original one.

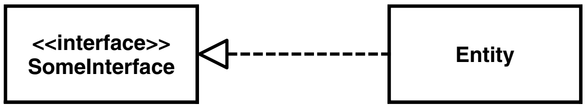


Figure 6.35 All *Entities* must implement a certain interface (incorrect)

Sometimes, the latter is required for other reasons. Assume there is a number of metamodel elements whose instances must all have names. It will possibly make sense to define an interface on the metalevel that contains the operation *Name()*. Figure 6.36 shows this.

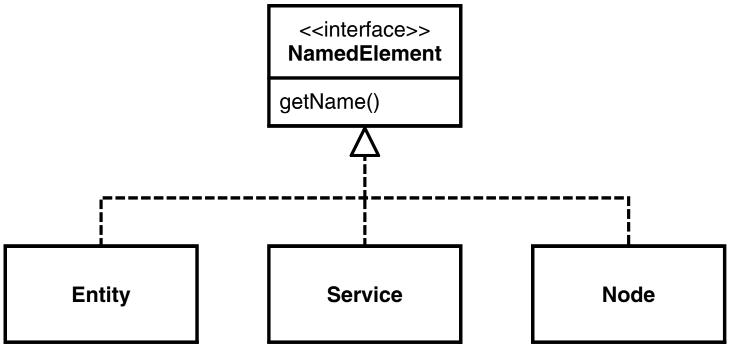


Figure 6.36 Use of interfaces and the *Implements* relation at the metamodel level

6.12.2 Dependencies

Problem: You want to express the fact that components can depend on interfaces because they invoke their operations.

Correct solution: You define an association between component and interface and call it *uses*. Figure 6.37 demonstrates that a component can use many interfaces and that an interface can be used by many components.

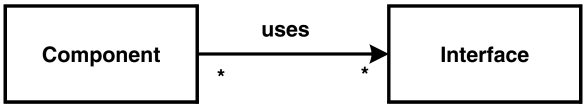


Figure 6.37 Dependencies (correct)

Incorrect solution: The model in Figure 6.38 states that the metaclass *Component* somehow depends on the metaclass *Interface*.

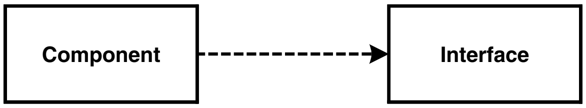


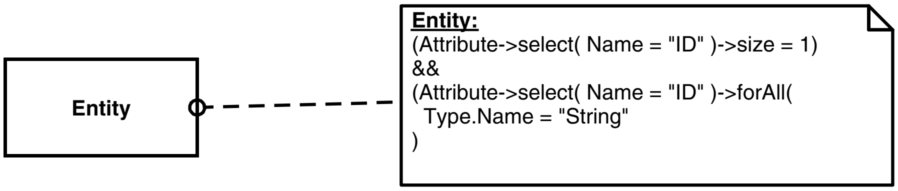
Figure 6.38 Mappings (incorrect)

Note that a *Dependency* like the one in Figure 6.38 can never have cardinalities. The statement ‘depends on several interfaces’ cannot therefore be mapped.

6.12.3 IDs

Problem: Entities must have exactly one attribute with the name *ID* of type *String*. This represents the identifying attribute or the primary key. We proceed on the premise that the metaclass *Entity* inherits from *UML::Class* and thus possesses the inherited association *Attribute*, which defines the attributes of the class.

Correct solution: The correct solution in Figure 6.39 uses an OCL constraint that states that among the attributes of the entities there must be one with the name *ID* and the type *String*.



```
Entity:
(Attribute->select( Name = "ID" )->size = 1)
&&
(Attribute->select( Name = "ID" )->forAll(
  Type.Name = "String"
)
```

Figure 6.39 Entities must have exactly one attribute with the name *ID* of the type *String* (correct)

Incorrect solution: The definition of an *Entity* attribute of the name *ID*, as shown in Figure 6.40, does not yield the correct result. Instead, it represents a definition of a tagged value for the metaclass *Entity.*

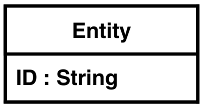


Figure 6.40 Entities must have exactly one attribute with the name *ID* of the type *String* (incorrect)

By the way, the following constraint in the correct model would also be incorrect:

```
context Entity inv:
  Attribute->select(
      (Name = "ID") && (Type.Name = "String")
    )->size = 1
```

This constraint would permit the existence of various attributes of the name *ID*, but only one of the type *String*.

6.12.4 Primary Keys

Problem: All instances of *Entity* must have among their attributes exactly one of the type *EntityPK*. Here, *EntityPK* is a specialization of the metaclass *Attribute*.

Correct solution: Figure 6.41 shows the correct metamodel:

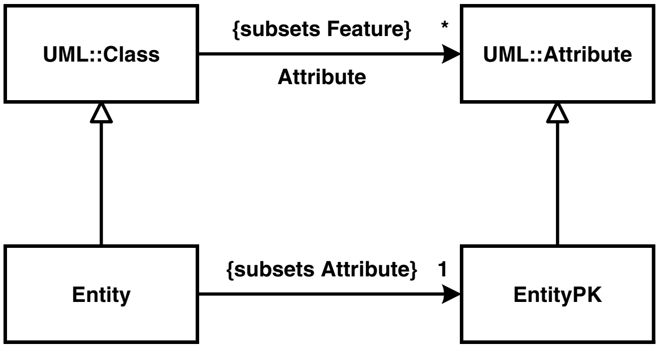


Figure 6.41 All instances of *Entity* must have among their attributes exactly one attribute of the type *EntityPK* (correct)

Incorrect solution: Figure 6.42 displays the same problem that we dealt with in the Section 6.12.3, the definition of a tagged value.

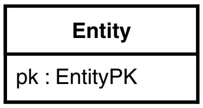


Figure 6.42 All instances of *Entity* must have among their attributes exactly one attribute of the type *EntityPK* (incorrect)

6.12.5 Metalevels and Instanceof

This example illustrates one of the pitfalls in the use of complex modeling languages using UML an example.

Figure 6.43 shows a UML class diagram (M1) and a UML object diagram (M0). Objects are instances of classes that are defined in the class diagram. So far, the object-class relationship is very clearly an *instanceof* relation, of which the fact that more than one object of the same class can exist is further proof. The same is true for the relationship between link and association.

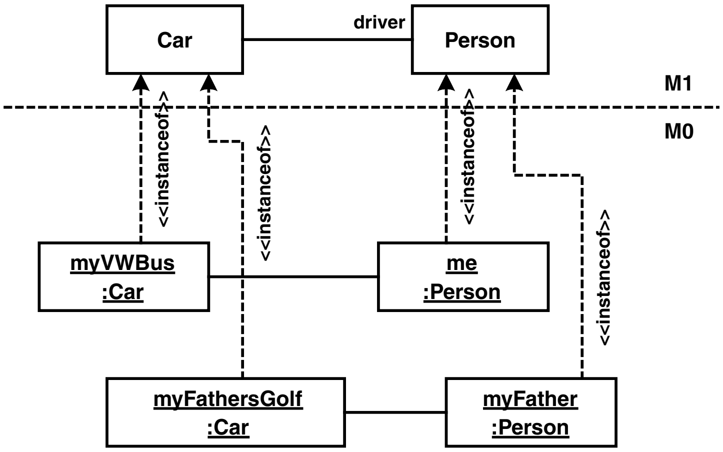


Figure 6.43 Objects as instances of classes

As can be seen in Figure 6.43, objects and classes are located on different metalevels. On the other hand, they are on the *same* metalevel in terms of UML. Classes as well as object models are instances of UML metamodel elements. Figure 6.44 shows this:

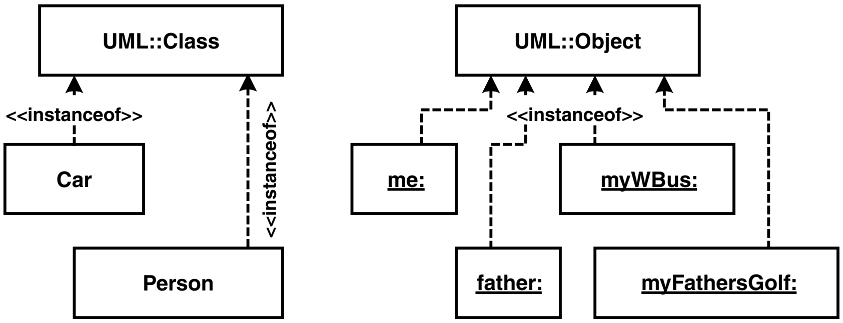


Figure 6.44 Model elements as instances of metamodel elements

On closer inspection, this apparent contradiction is easily resolved: the two *instanceof*s are not the same language construct. In the first example, *instanceof* is part of UML and is defined in that context (see Figure 6.45).

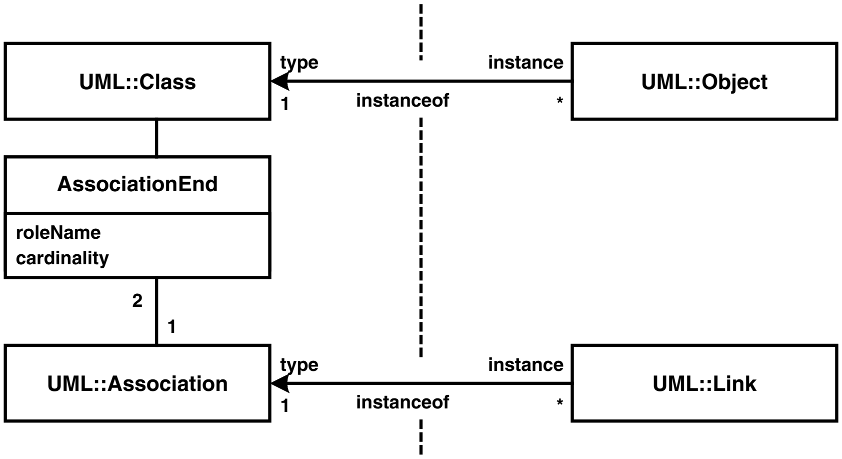


Figure 6.45 The *instanceof* relation defined in UML

The relation between *UML::Class* and *UML::Object* is an *MOF::Association*. It defines the *instanceof* relation between instances of *UML::Class* and *UML::Object* in instances of this (meta-)model – that is, in UML models. Nevertheless, all model elements (*me, my father, myVS-bus, myfathersGolf*) are of course instances of UML metaclasses, in this case *UML::Class* or *UML::Object*.

7 MDSD-Capable Target Architectures

7.1 Software Architecture in the Context of MDSD

As important as the term *software architecture* is, unfortunately it is also just as vaguely defined. We neither wish nor have to make the attempt to deliver a universally-valid and detailed definition – to this end, we recommend the appropriate literature, such as [BCK98], [POSA1], [JB00], [PBG04]). For further discussion, it is sufficient to carve out the relevant points of view and specifics. The following, simple 'definition' of the term software architecture will serve as a basis:

Software architecture describes to a certain level of detail the structure (layering, modularization etc.) and the systematics (patterns, conventions etc.) of a software system.

The topic of software architecture plays a role in various subcontexts of MDSD:

- First, a software architecture serves to structure the software systems to be generated or created at large. Here the reference implementation plays a central role: the software architecture of applications to be generated is already visible in its entirety – if only as an example. Yet each complete member of the software system family possesses the same software architecture. This view of the topic is therefore the classic and common one: How do you structure applications or, respectively, software systems? The answer to this question is independent of whether development is model-driven or not, which is why you can draw on the whole toolbox of software architecture to come up with answers. In the context of MDSD, additional requirements must also be considered. This MDSD perspective on the topic of software architecture leads us to the term *target architecture*. The target architecture contains the platform architecture (see below).
- An MDSD *domain architecture* (see Chapter 4) is also a software architecture. It defines the whole of the metamodel, DSL, and platform, as well as transformations. The domain architecture provides the basis for a software system family's products. Here, we are to a certain extent operating at the metalevel, because a domain architecture serves the creation of software – the domain architecture determines substantial parts of the target architecture.
- Software architecture is relevant in the context of the MDSD platform, where it describes the most important platform components, their interactions, as well as their non-functional characteristics, which is why we call it *platform architecture* in this context. The platform can be found at both the metalevel, because it is part of the domain architecture, and on the

concrete level, because it is also part of each software[1] that was generated with the help of the domain architecture – otherwise the software would not be complete: that is, runnable.
- Software architecture also plays a role in MDSD transformations, because it actually defines the software architecture of the generated code – which is part of the target architecture, as explained above – and, if necessary, concrete integration points for custom logic that must be programmed manually. Transformations are software too, and should therefore be structured by a software architecture. We call the latter a *transformation architecture*.
- Finally, generic MDSD tools must also meet certain architectural standards. We call their software architecture *tool architecture*.

This categorization provides us with a topical segmentation that we can use to further structure the next chapters of this part of the book:

- This chapter discusses target architecture, as well as platform architecture.
- Chapter 8 deals with transformation architecture, and also covers domain architecture.
- Chapter 9 looks in detail at code generation techniques.
- Chapter 10 provides insight into the state of the art of model transformations.
- Chapter 11 introduce the basics of and selection criteria for tool architectures.

In the context of MDSD, the target architecture is of extreme importance. The generation of parts of this architecture can be automated at all only if its concepts are well-defined. If the artifacts to be generated cannot be described systematically, the creation of generation rules (transformations) and thus of a domain architecture, is impossible. All of the recommendations in this chapter are therefore relevant for the creation of MDSD reference implementations.

7.2 What Is a Sound Architecture?

As we explained in Section 7.1, the application to be created must have a sound architecture. From our viewpoint, a sound architecture exhibits the following characteristics:

- First of all, the architecture must sufficiently support the functional requirements of the application for which it is created. Without this property, the architecture is useless.
- Furthermore, it must realize the expected non-functional requirements. Among these are dynamic aspects such as performance or scalability, but also factors like availability, testability, and maintainability.
- The architecture should comprise as small as possible a set of clearly defined constructs. Thus the architecture becomes simpler, easier to understand and, in consequence, practicable.
- The architecture should also allow specific growth/development paths for the application. This is not about creating an ideal solution that serves all purposes, but about creating a clearly-defined architecture that is potentially expandable.
- A sound architecture is also well-documented. This includes a brief and concise documentation of all the points listed above, a programming model that explains how one implements applications based on the architecture, as well as a rationale elaborating why the architecture was created the way it is and why possible alternatives were rejected.

[1] This is a *uses* relation: the platform is used by the (generated) software products, but it doesn't *belong* to them.

You know that you are dealing with a sound architecture if it can be implemented and used in everyday project business – even when time presses – in the way that its designer envisaged, and if it stands the test of time in daily practice.

In the context of MDSD, there are two important aspects to observe:

- First, the architecture must be able to support all products of the software system family. It should also be able to map new products in the family that were not known in detail at the time the domain architecture was defined (for further details, see Section 13.5).
- The architecture's concepts must be defined even more clearly, otherwise they cannot be generated automatically via transformation from models.

The second aspect is especially interesting: the concepts and constructs of an architecture that is to serve as a platform for MDSD must be defined very precisely. Although we have identified well-definedness as a sign of quality in a good architecture, one could say that the application of MDSD not only fosters a sound software architecture, but actually enforces it.

7.3 How Do You Arrive at a Sound Architecture?

The question of how one comes up with a sound architecture – generatively or not – fills whole books, and we do not wish to discuss it here in its entirety. Nevertheless, we want to address some of its aspects.

7.3.1 Architectural Patterns and Styles

In software technology only a limited number of architectural blueprints that work well are known. These have been described using various forms: among others, as patterns [POSA1], or styles [BCK98]. Figure 7.1 shows some typical architectural styles.

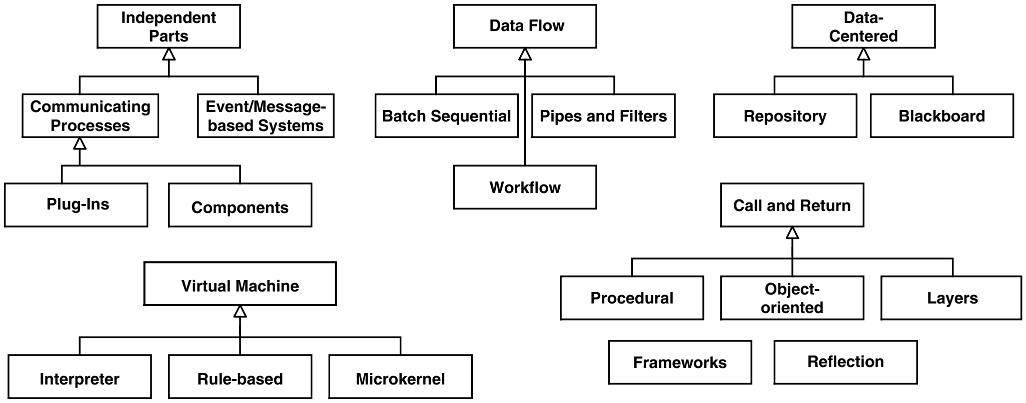


Figure 7.1 Some popular software architectural styles

A proven way of obtaining a good architecture is the use of a tried and tested architectural pattern or style as basis of one's own architecture. [POSA1] describes the basic architectural patterns quite well and extensively. Additionally, there are a number of books that describe the architecture of specific types of systems. Here, a few examples:

- *Patterns for Concurrent and Networked Objects* describes distributed, multi-threaded systems [POSA2].
- *Resource Management Patterns*, addresses the architecturally-significant aspect of resource management [POSA3].
- *Server Component Patterns* describes the internal architecture of component infrastructures such as EJB, CCM, or COM+ [VSW02].
- *Remoting Patterns* describes the internal architecture of remoting middleware such as CORBA, .NET Remoting, or Web Services [VKZ04].
- *Patterns of Enterprise Architecture* describes the architecture of big enterprise systems in general [Fow04].
- *Enterprise Integration Patterns* describes the architecture of EAI systems and messaging middleware [Hor04].

Today, reference architectures and platforms like J2EE or .NET are often used as a basis for architectures. The use of such a platform does not yield a solid architecture automatically, but it can serve as a solid foundation, specifically for non-functional aspects. One still has to decide which concepts offered by the platform one wishes to use and how.

A proven method for obtaining a good architecture is continuously to develop an architecture over the course of several applications (ideally of the same software family). The experience gathered working with the architecture can contribute to improving newer versions of the architecture or other members of the system family. In this respect, too, MDSD has a positive effect on the software architecture of the application created.

7.4 Building Blocks for Software Architecture

This section discusses some aspects of software architectures and their relevance in the context of MDSD.

7.4.1 Frameworks

We call *frameworks* anything that can be adapted or extended via systematic extension or configuration. For example, developers who use a framework must specify specific configuration parameters, extend superclasses, or implement callbacks. As a rule, more than one of these adaptations must be made to realize a specific functionality (that is, a specific feature). It is important that these adaptations are compatible with each other. For many frameworks, this is not always easily accomplished, which is one of the main reasons why frameworks are sometimes difficult to use and enjoy a dubious reputation.

MDSD can help insofar as it lets you specify the required features via a suitable DSL. You can then proceed to generate the various adaptations of the framework from the models built with the

DSL. Frameworks and DSLs are therefore an ideal combination: MDSD platforms can be very well implemented with the aid of frameworks.

7.4.2 Middleware

Middleware can be seen as a kind of framework. In most cases, it is specific to a technical domain such as distributed systems, messaging, or transactions, and provides the technical basis for a target architecture. Due to its focus on technical aspects, middleware is applicable in many functional and professional domains[2], and is thus often standardized. Well-known examples are CORBA, DCOM, MQSeries, and CICS.

7.4.3 Components

Component infrastructures are an especially powerful and very popular type of middleware. Without wanting to fully immerse ourselves in a discussion of how to define the term ‘component’, a brief explanation is in order here

A component is a self-contained piece of software with clearly-defined interfaces and explicitly-declared context dependencies.

Components therefore constitute the basis of tidily modularized and assemblable systems. Many domain architectures serve to define components or to put pre-fabricated components together to build an application.

Another important aspect of components is that they are the ‘smallest common denominator’ for the composition of systems that are specified via different DSLs, because of the various subdomains in a system. Ideally, this assembly should take place at the model level (see Chapter 15), but the tools needed for this purpose (model transformers) are not always available or applicable in practice. For this reason, the combination of different subsystems happens at the implementation level, as illustrated in Figure 7.2.

Container infrastructures such as EJB, COM+, or .NET Enterprise Services constitute an important foundation for MDSD. After all, they provide a technical platform for components that ideally only contain code that is related to the functional requirements of the system. In this case containers factor the technical aspects from components and make them available in a standardized and reusable form. Such containers are mostly not generated, but merely configured by MDSD through generation of configuration files from the model (deployment descriptors in EJB). The exception are component infrastructures for embedded systems, which we discuss in Chapter 16.

[2] We use the term ‘functional/professional’ throughout the book as an English version for the German word *fachlich*. The word does not have a direct equivalent in English: in German we speak of *technischen* domains and *fachlichen* domains as their opposite. *Technisch* clearly deals with technical issues such as scalability, persistence, transactions, load balancing, security. A functional/professional (*fachlichen)* domain is one that deals with application-orientated issues, for example insurance, radio astronomy, tax calculation, engine management and so on.

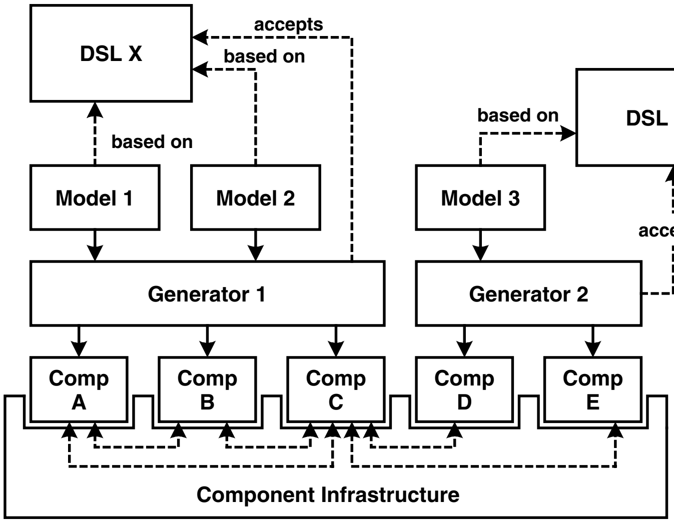


Figure 7.2 Integration on the implementation level

The integration of MDSD and component-based development is further illustrated in the remaining sections of this chapter and in Chapter 17.

7.5 Architecture Reference Model

In practice a layered model has proven to be most useful in software architectures. This structure can be found in some form in almost all well-structured software systems. Figure 7.3 shows this.

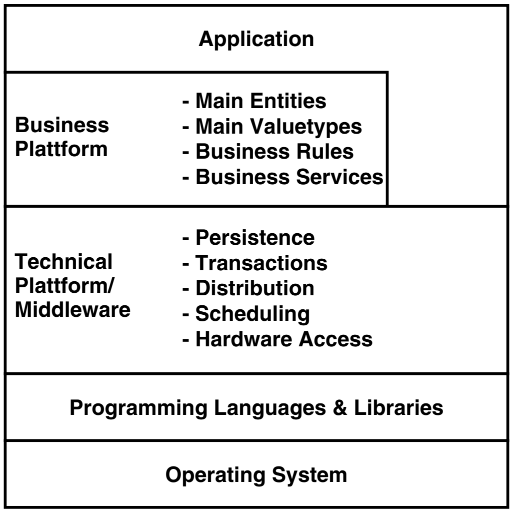


Figure 7.3 An architecture reference model

The operating system and the programming language form the basis of each architecture. Building on this foundation, there is usually a technical framework that provides essential technical services, often implemented using middleware. These can be persistence, transactions, distribution, workflow, GUIs, scheduling, or hardware access functions. The question of which services are actually offered by this layer depends on the technical domain, such as real-time embedded, business, or peer-to-peer domains. Typical examples of such frameworks are J2EE and .NET in the enterprise field, and Osek + Standard Core in the field of embedded systems, mostly in the automotive sector.

It is typical to find a framework based in this layer that provides the foundation for the functional/professional domain:

- *Entities* represent concepts that possess an identity and a lifecycle – for example, each customer has an identity that must be preserved as long as the object exists.
- *Value objects* represent values. Amounts of money in banking, for example, or coordinates sets in a GPS application are good examples of value objects. Value objects do not have an identity: only their value is relevant, and two objects with the same value are considered identical.
- *Business rules* and *Constraints*. Here, the domain's basic rules that cannot be assigned to an *Entity* are captured, for example that drivers of cars must always be older than eighteen, or that the amount of money in a transaction can never be negative.
- *Services*. Basic services are defined here that cannot be assigned to an *Entity*, for example the execution of a transaction, or the validation of a complex document structure in editorial systems.

Even though this list is based on Enterprise/Business systems, these statements are also valid for technical or embedded systems, although the terminology and the software/technical implementations are different in that context.The actual application builds on these frameworks.

7.6 Balancing the MDSD Platform

In the context of MDSD the reference model is very important – not only for structuring the target architecture, but specifically to define the boundaries of the MDSD platform, which is part of the target architecture. Finally, the application models must be mapped to the MDSD platform using transformations in order to make them executable. The larger the difference between the concepts of the MDSD domain and the concepts of the MDSD platform, the more complex the necessary transformations will be. This should be avoided, especially since complexity at the metalevel is harder to cope with than complexity at the concrete level of the target architecture. To decrease complexity, the MDSD domain and the MDSD platform should be as close to each other as possible – more precisely, the MDSD platform should 'meet the MDSD domain halfway'. We also call such a platform a *rich, domain-specific platform*.

As far as the reference model is concerned, we now have several basic options for reducing the conceptual distance between domain and platform:

- *MDSD domain and platform are located at the level of the reference model's technical platform*. This would for example lead to a choice of a UML profile for J2EE as DSL and

J2EE as an MDSD platform. Such a domain architecture is of course feasible, but due to its limited abstraction level it does not use the full potential for automation: the bulk of a modeled application must be programmed manually. On the other hand, the MDSD domain is quite versatile – the domain architecture allows for the production of very different applications.

- *MDSD domain and platform are at the level of the target architecture's concepts*. AC-MDSD is an example of this, as in Chapter 3's case study. Here the architectural realization patterns of the functional platform are derived from the reference model for the DSL definition, as well as for the MDSD platform. In this case, the domain is less versatile, but its abstraction level is increased and thus its potential for automation is higher.
- *MDSD domain and platform are at the level of the functional/professional platform of the reference architecture*. In this case the functional/professional platform of the reference architecture becomes the MDSD platform. The domain is significantly less versatile than in the architecture-centric case, but automation can reach 100% without a problem.

We recommend the last two options – or even a cascading of both – that is, the creation of a functional/professional MDSD platform on top of an architecture-centric domain architecture. This approach is illustrated in the case study in Chapter 16.

7.6.1 Examples

Where the boundary of the MDSD platform is drawn in practice, and where in each particular case, depends typically on how much flexibility is needed in the specific context. Here are some examples:

- Typical ingredients of an architecture-centric MDSD platform for e-business systems are flow control or workflow engine, persistence framework, superclasses for GUIs, activities and entities and so on, and technical standard infrastructures such as J2EE containers and relational databases. All these artifacts are identical for each software system of the architecture-centric MDSD domain.
- In the context of a system family for radio telescopes in the astronomy domain, stars, galaxies, or planets are relevant entities. Their properties usually don't change much, therefore these entities are part of the MDSD platform. Also, many 'business rules' are static in this case because they are based on the laws of physics.
- For insurance companies, insurance products are relevant entities. These are actually very different and change frequently. Such entities are conveniently described using the DSL, and their implementation is generated. Other core entities such as *Person* or *Account* can be part of the MDSD platform.
- In the context of a component infrastructure for distributed embedded systems, even the technical platform, the middleware, is generated based on predefined system constraints and topology definitions and is thus not part of the MDSD platform. Even if all basic entities or the technical infrastructure are generated, a domain-specific basic framework typically exists as part of the MDSD platform. In an embedded system, for example, we

might find bus drivers and marshalers to serialize a data structure for transport across the network.

We recommend that you expand the power of your MDSD platform incrementally in the course of your project, in keeping with your growing understanding of the domain. This will reduce the scope and complexity of the code developers have to write to customize the framework, which must either be generated or even partially be written manually.

The general rule is that generic and generalizable code segments should be part of the MDSD platform. Existing frameworks are usually also well-suited for integration in the MDSD platform.

7.6.2 Integration of Frameworks

The use of complex frameworks can be significantly simplified and sped up through the use of customized DSLs. This is often overlooked by framework purists. In many cases, only a DSL and a model-driven approach can guarantee that a framework is used correctly – that is, as intended by its inventor. Today's implementation languages do not possess any particularly powerful mechanisms for preventing faulty framework use. By definition frameworks entail a strong interlocking of framework implementation and framework use, which often contradicts the encapsulation principle. The domain can be clearly isolated from the applied implementation platform only via a DSL.

On the other hand, highly configurable, generic frameworks attract the danger of overburdening their implementation, making them difficult to maintain and hard to debug. This should be considered in MDSD platform construction.

The key to successful MDSD platform design is an iterative, incremental approach, as is described in Chapter 13. The development of powerful frameworks in large, independent projects that have no direct iterative connection with real-life application development projects will inevitably result in failure. Instead, small frameworks combined with code generation offer a solid basis for iterative development. When code generation – that is, writing templates – becomes too difficult, it is worth enhancing the frameworks and implementing more features with their help. Vice versa, if the implementation of the framework becomes overly complicated, or its runtime performance deteriorates, the use of generative techniques will often lead to elegant solutions.

7.7 Architecture Conformance

A good target architecture can exhibit its advantages only if it is not ignored or circumvented in the daily project routine. Traditional methods such as reviews and excessive documentation are not easily scalable when working with bigger teams. For generated code, MDSD per se offers the solution, particularly because the aspects of the architecture that are described using the models are laid down in the form of transformation rules.

The component example described in Chapters 6 and 16 serve as an example of how one can enforce or control the observation of specific characteristics of the target architecture in manually-programmed parts of systems as well. We can to demonstrate which effects the use of these options can have on the target architecture. Figure 7.4 helps to illustrate this example once more:

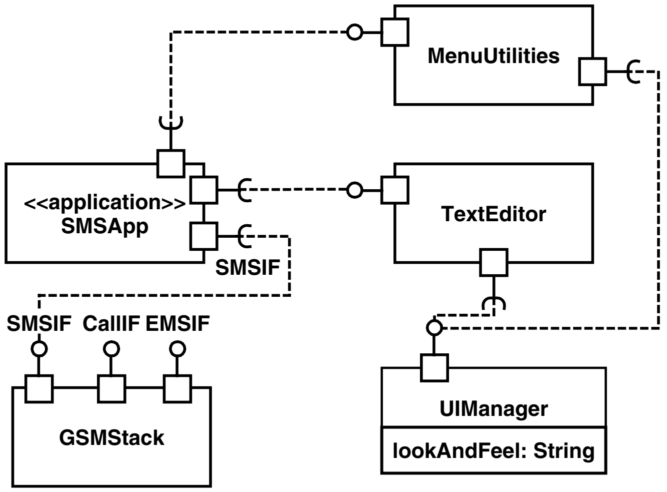


Figure 7.4 An example of dependencies between components

The figure shows, among other things, that the component *SMSApp* depends on the components *MenuUtilities, TextEditor*, and *GSMStack* via their interfaces. The explicit description of such dependencies and their management is an essential aspect of architecture in large projects. Therefore, it should be impossible for a component or its implementation code to access interfaces or components *for which no dependency is defined in the model*. This must actually be ensured in large projects, ideally by automation. It can be achieved with MDSD, particularly through its aspect of code generation. The following code segment shows a 'classic' implementation of a component: access to other components is obtained by querying the corresponding reference from a central component factory:

```
public class SMSAppImpl {
  public void tueWas() {
    TextEditor editor = (TextEditor)
           Factory.getComponent("TextEditor");
    editor.setText( someText );
    editor.show();
  }
}
```

It is not easy to ensure that the developer does not illegally obtain other references from the factory. This can only be accomplished through reviews or other tools such as AspectJ. However, if development is model-driven, there is another option: for each component, a *component context* [VSW02] can be generated, which exclusively permits access to those components or interfaces to which a dependency is given in the model. The following code illustrates this:

```
public interface SMSAppContext
         extends ComponentContext {
```

```
public TextEditorIF getTextEditorIF();
  public SMSIF getSMSIF();
  public MenuIF getMenuIF();
}
public class SMSAppImpl implements Component {
  private SMSAppContext context = null;
  public void init( ComponentContext ctx) {
    this.context = (SMSAppContext)ctx;
  }
  public void tueWas() {
    TextEditor editor =
           context.getTextEditorIF();
    editor.setText( someText );
    editor.show();
  }
}
```

Now the developer can no longer autonomously get arbitrary references – they can only access those for which accessor operations exist in the component context. These access operations are generated from the model. If the developer wishes to access other interfaces, they must include them in the model, otherwise the necessary accessor method will not be available in the code. This guarantees that a component only possesses those dependencies that it explicitly declares in the model. A more extensive infrastructure is required, of course: for example, ‘someone’ must call the operation *init()* and provide the correct context object. In component-based systems this is the task of a *container* – in this case the runtime system, which is in charge of the component’s lifecycle.

This approach has the pleasant side-effect that in modern IDEs one is conveniently informed, via code completion, of which operations are present in the component context – thus information on the legal dependencies can be found directly in the code, making it easier for the developer to observe the architecture’s rules!

The example given above also shows what effects the use of MDSD has on the architecture. It would never occur to a developer to provide a separate context class for each component if this had to be implemented manually. The use of MDSD therefore opens up additional architectural alternatives. It is pivotal to know about and use these alternatives when the target architecture is defined.

7.8 MDSD and CBD

Component-Based Development (CBD) is a popular metaphor for building complex systems, as is Service-Oriented Architecture, which is covered in Section 7.8. We have already covered some aspects of the interplay between MDSD and CBD in the previous sections: in this section we want to take it a step further. From our experience in development projects, we find that we almost always start by modeling the component structure of the system to be built. To do that, we start by defining what a component actually is — that is, by defining a metamodel for component-based development. Independent of the domain in which the development project resides, these metamodels are quite similar across application domains – insurance, e-commerce, radio astronomy, and so on (as opposed to technical domains such as persistence, transaction processing, security.) We therefore show parts of these metamodels here to give you a head start when defining your own component architecture. This ties in nicely with the architectural process proposed in Section 13.4.

7.8.1 Three Viewpoints

It is useful to look at a component-based system from three viewpoints, and idea that we enlarge on in the case study on enterprise systems in Chapter 17.

The Type Viewpoint

The *type* viewpoint describes component types, interfaces, and data structures. A component provides a number of interfaces and references a number of required interfaces. An interface owns a number of operations, each with a return type, parameters, and exceptions. Figure 7.5 shows this.

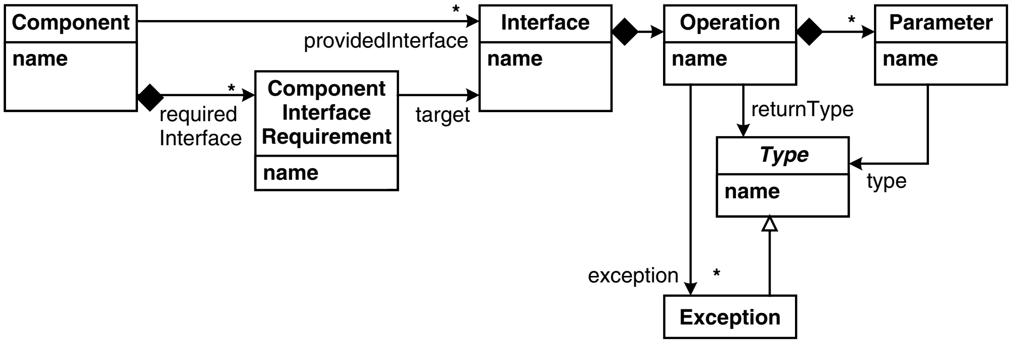


Figure 7.5 The metamodel for components, interfaces, and their dependencies

To describe the data structures with which the components work (Figure 7.6), we start out with the abstract type *Type*. We use primitive types as well as complex types. A complex type has a number of named and typed attributes. There are two kinds of complex types. *Data transfer objects* are simple structs that are used to exchange data among components. *Entities* have a unique ID and can be made persistent (this is not visible from the metamodel). Entities can reference each other and thus build more complex data graphs. Each reference has to specify whether it is navigable in only one or in both dimensions. A reference also specifies the cardinalities of the entities at the respective ends.

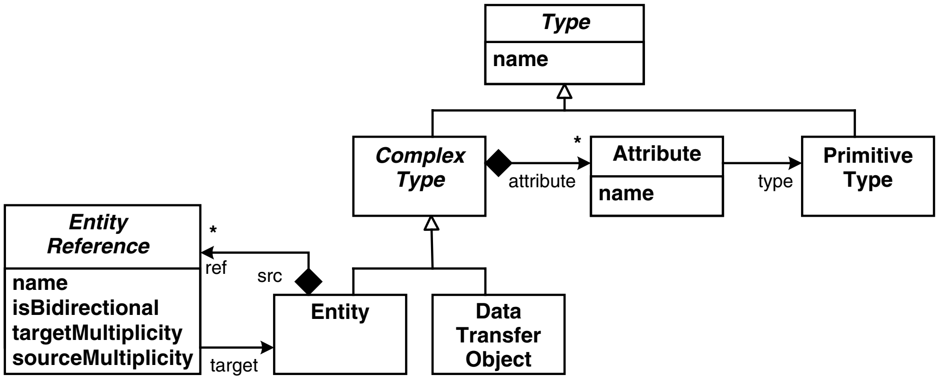


Figure 7.6 The metamodel for data structures

The Composition Viewpoint

This viewpoint, illustrated in Figure 7.7, describes component instances and how they are connected. A configuration consists of a number of component instances, each knowing their type. An instance has a number of *wires*: a wire is an instance of a component interface requirement. Note the constraints defined in the metamodel:

- For each component interface requirement defined in the instance's type, we need to supply a wire.
- The type of the component instance at the target end of a wire needs to provide the interface at which the wire's component interface requirement points.

Using the type and composition viewpoints, it is possible to define component types as well as their collaborations. Logical models of applications can be defined. You could, for example, use UML to render these two kinds of models, generate skeleton classes, and then implement the application logic in subclasses. From the composition viewpoint, you can generate or configure a container that instantiates the component instances. Unit tests that verify the application logic can be run here.

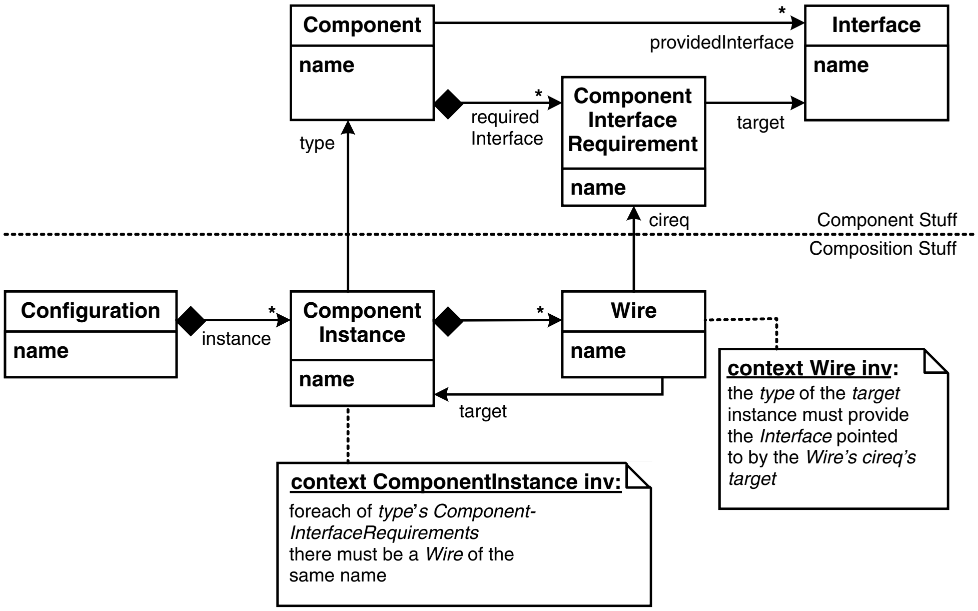

Component
name
providedInterface
*
Interface
name
*
required Interface
Component Interface Requirement
name
target
type
cireq
Component Stuff
Composition Stuff
Configuration
name
*
instance
Component Instance
name
*
Wire
name
target
context Wire inv:
the *type* of the *target* instance must provide the *Interface* pointed to by the *Wire's cireq's target*
context ComponentInstance inv:
foreach of *type's Component-InterfaceRequirements* there must be a *Wire* of the same name

Figure 7.7 Component instances and their connections in the composition metamodel

The System Viewpoint

This third viewpoint describes the system infrastructure onto which the logical system defined with the two previous viewpoints is deployed.

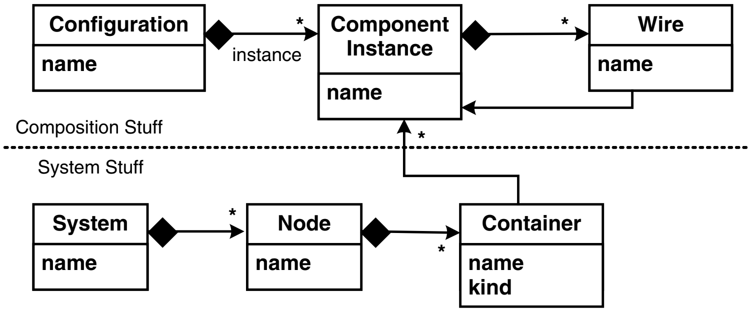


Figure 7.8 The metamodel for systems

A system consists of a number of nodes, each one hosting containers. A container hosts a number of component instances. Note that a container also defines its kind – this could be things like CCM, J2EE, Eclipse or Spring. Based on this information, you can generate the necessary 'glue' code to run the components in that kind of container.

The node information, together with the connections defined in the composition model, allows you to generate all kinds of things, from remote communication infrastructure code and configuration to build and packaging scripts.

7.8.2 Viewpoint Dependencies

You may have observed that the dependencies among the models are well-structured. Since you want to be able to define *several* compositions using the same components and interfaces, and since you want to be able to run the same compositions on *several* infrastructures, dependencies are only legal in the directions shown in Figure 7.9.

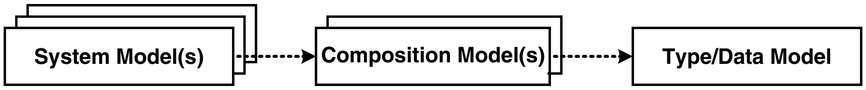


Figure 7.9 Dependencies among the viewpoint models

7.8.3 Aspect Models

The three viewpoints described above are a good starting point for modeling and building component-based systems. However, in most cases these three models are not enough. Additional aspects of the system have to be described using specific aspect models that are arranged around the three core viewpoint models, as illustrated in Figure 7.10.
The following aspects are typically handled in separate aspect models:

- Persistence
- Authorization and Authentication (important in enterprise systems)
- Forms, layout, pageflow (for Web applications)
- Timing, scheduling and other quality of service aspects (especially in embedded systems)
- Packaging and deployment
- Diagnostics and monitoring

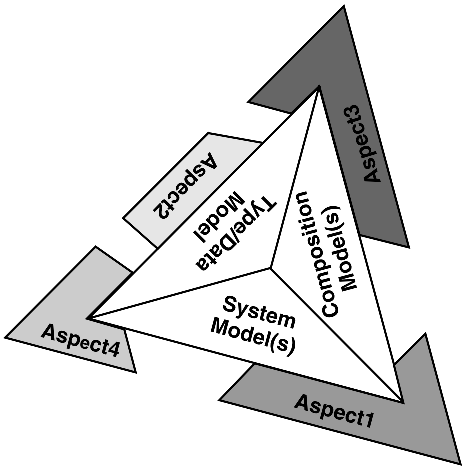


Figure 7.10 Arranging the aspect models around the three core viewpoint models

The idea of aspect models is that the information is not added to the three core viewpoints, but rather is described using a separate model with a suitable concrete syntax. Again, the metamodel dependencies are important: the aspects may depend on the core viewpoint models and maybe even on one another, but the core viewpoints must not depend on any of the aspect models. Figure 7.11 illustrates a simplified persistence aspect metamodel.

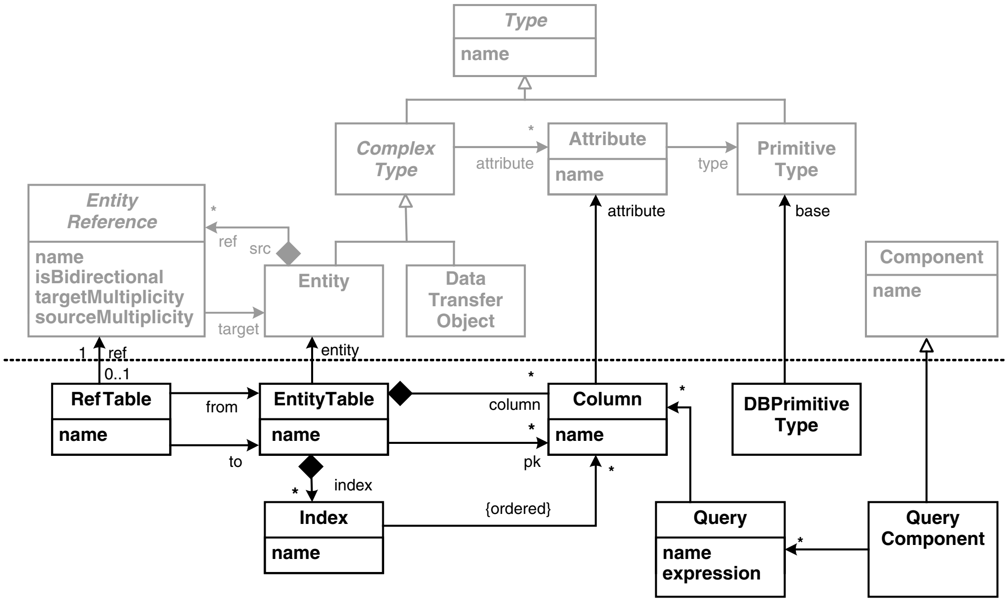
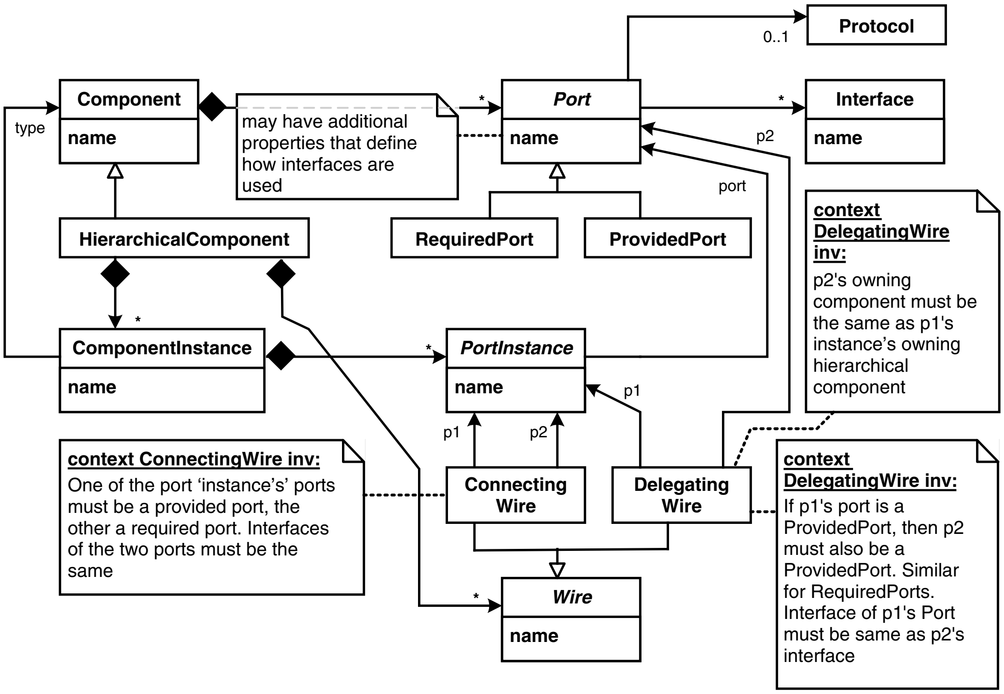


Figure 7.11 The metamodel for the (relational) persistence aspect

7.8.4 Variations

The metamodels we describe above cannot be used in exactly this way in every project. Also, in many cases the notion of what constitutes a component needs to be extended. So there are many variations of these metamodels. However, judging from practice, even these variations are limited. In this section we want to illustrate some of these variations.

- You might not need separate interfaces. Operations could be added directly to the components. As a consequence, of course, you cannot reuse the interface 'contracts' separately, independently of the supplier or consumer components.
- Often you'll need different kinds of components, such as domain components, data access (DAO) components, process components, or business rules components. Depending on this component classification you can come up with valid dependency structures between components. You will typically also use different ways of implementing component functionality, depending on the component types (see also Section 7.8.5).
- Another way of managing dependencies is to mark each component with a *layer* tag, such as *domain, service, gui*, or *facade*, and define constraints on how components in these layers may depend on each other.
- Hierarchical components, as illustrated in Figure 7.12, are a very powerful tool. Here a component is internally structured as a composition of other component instances. Ports



may have additional properties that define how interfaces are used

context DelegatingWire inv:
p2's owning component must be the same as p1's instance's owning hierarchical component

context ConnectingWire inv:
One of the port 'instance's' ports must be a provided port, the other a required port. Interfaces of the two ports must be the same

context DelegatingWire inv:
If p1's port is a ProvidedPort, then p2 must also be a ProvidedPort. Similar for RequiredPorts. Interface of p1's Port must be same as p2's interface

Figure 7.12 The metamodel for hierarchical components

define how components may be connected: a port has an optional protocol definition that allows for port compatibility checks that go beyond simple interface equality. While this approach is powerful, it is also non-trivial, since it blurs the formerly clear distinction between type and composition viewpoints.

- A component might have a number of configuration parameters – comparable to command line arguments in console programs – that help configure the behavior of components. The parameters and their types are defined in the type model, and values for the parameters can be specified later, for example in the composition or the system models.
- You might want to say something about whether the components are stateless or stateful, whether they are thread-safe or not, and what their lifecycle should look like (for example, whether they are passive or active, whether they want to be notified of lifecycle events such as activation, and so on).
- It is not always enough to use simple synchronous communication. Instead, one of the various asynchronous communication patterns, such as those described in [VKZ04], might be applicable. Because using these patterns affects the APIs of the components, the pattern to be used has to be marked up in the type model, as shown in Figure 7.13.

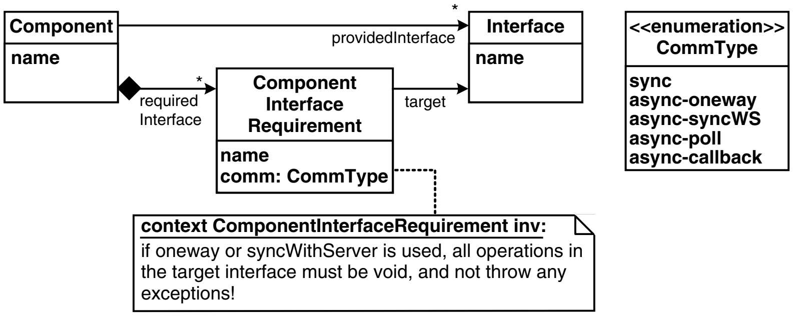

Component
name
providedInterface
*
Interface
name
required Interface
*
Component Interface Requirement
name
comm: CommType
target
<<enumeration>> CommType
sync
async-oneway
async-syncWS
async-poll
async-callback
context ComponentInterfaceRequirement inv:
if oneway or syncWithServer is used, all operations in the target interface must be void, and not throw any exceptions!

Figure 7.13 Asynchronous communication

- In addition to the communication through interfaces, you might need (asynchronous) events using a static or dynamic publisher/subscriber infrastructure. It is often useful that the 'direction of flow' of these events is the opposite of the is the opposite of the uses-dependencies discussed above.
- The composition model connects component instances statically. This is not always feasible. If dynamic wiring is necessary, the best way is to embed the information that determines which instance to connect to at runtime into the static wiring model. So, instead of specifying in the model that instance A must be wired to instance B, the model only specifies that A needs to connect to a component with the following properties: needs to provide a certain interface, and for example offer a certain reliability. At runtime, the wire is 'deferenced' to a suitable instance using an instance repository. This approach is similar to CORBA's trader service.
- Finally, it is often necessary to provide additional means of structuring complex systems. The terms *business component* or *subsystem* are often used. Such a higher-level structure

consists of a number of components. Optionally, constraints define which kinds of components may be contained in a specific kind of higher-level structure. For example, you might want to define that a business component always consists of exactly one facade component and any number of domain components.

7.8.5 Component Implementation

Component implementation typically happens manually. This means that developers add manually-written code to the component skeleton, either by adding the code directly to the generated class, or – a much better approach – by using other means of composition such as inheritance or partial classes. The main reason is that action languages that support the generic formulation of application logic at the model level are still not widely supported. However, using a generic action language to describe the behavior of structural artifacts (such as a class's operations or a state's actions) is only one alternative. There are other means of describing application logic, some of which we outline below. All have in common that instead of providing a *generic* way of modeling all kinds of behavior in models, they use notations that are specific to the *kind* of behavior that should be specified.

- Behavior that is very regular can be implemented using the generator, after parametrizing it in the model by setting a small number of well-defined variability points. Feature models are good at expressing the variabilities that need to be decided so that an implementation can be generated.
- For state-based behavior, state machines can be used.
- For things such as business rules, you can define a DSL that directly express these rules and use a rules engine to evaluate them. Several rule engines are available off-the-shelf.
- For domain-specific calculations, such as those common in the insurance domain, you might want to provide a specific textual notation that supports the mathematical operations required for the domain directly. Such languages are often interpreted: the component implementation technically consists of an interpreter that is parametrized with the program it should run.

Note that we are not generally arguing against Action Semantics Languages (ASLs), we just want to point out that they don't provide domain-specific abstractions, being generic in the same way as for example UML is a generic modeling language. However, even if you use more specific notations, there might still be a need to specify small snippets of behavior generically. A good example are actions in state machines.

To combine the various ways of specifying behavior with the notion of components, it is useful to define various kinds of components, using subtyping at the metalevel, that each have their own notation for specifying behavior. The case study in Chapter 17 illustrates this approach, and the idea is also illustrated in Figure 7.14.

Since component implementation is about behavior, technically, it is often useful to use an interpreter encapsulated inside the component. Such an 'interpreter component' is still a component just as any other. Their introduction however raises an issue that needs to be addressed: How does the interpreter know which script to execute?

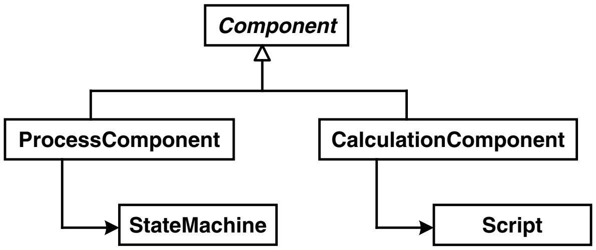


Figure 7.14 Subtypes of component to host various kinds of behavioral specifications

There are basically three different approaches to this. Either the component always executes the same script, the entire script is passed to the component as a parameter, or some sort of identifier for the script is passed to the component. This can be done either as part of the configuration process when the component is created, or it can be done with every method call. For more details on interpretation, see Section 8.4.

As a final note, be aware that the discussion in this section is only really relevant for *application-specific* behavior, not for all implementation code. Huge amounts of implementation code is related to the technical infrastructure – remoting, persistence, workflow and so on – of an application, and can be derived from the (structural) models.

7.9 SOA, BPM and MDSD

Service-Oriented Architectures (SOA) and Business Process Management (BPM) are two highly hyped topics in today's IT world. This section takes a MDSD-centric view of them and discusses their possible synergy.

7.9.1 SOA

There is no commonly agreed definition of what SOA actually means. Some people equate SOA merely to 'using Web Services'. In fact SOA is at least driven by Web Service technology, including BPEL (Business Process Execution Language). From our perspective, SOA has nothing to do with specific technologies (WSDL, SOAP, HTTP), but rather constitutes a set of architectural best practices for building large, scalable, and composable systems. A well-constructed component-based architecture with well-defined interfaces and clear-cut component responsibilities can quite justifiable be considered SOA. Components are a natural choice for the building blocks that provide and consume services. The industry realizes this and currently defines a standard for Service-Component Architectures [SCA].

However, looking at SOA a bit more closely, it is possible to identify a number of important properties that cannot readily be found in (most) component-based systems:

- Service interactions are message-oriented, or document centric. Instead of defining rigidly typed interfaces, document structures (schemas) are defined that serve as the basis for interactions. Done right, this can make evolution of message structures and versioning much simpler.

- The interaction patterns – valid sequences of messages – are explicitly defined. Interactions are often conversational – conversational session state is kept 'on both sides' of a service interaction. These features are the basis for orchestration among services. Usually, interactions are asynchronous.
- Quality of service aspects are explicitly addressed. Service providers do not just provide a certain services' functionality, they provide the functionality with a defined service level in terms of performance and reliability.
- Service descriptions and characteristics are available at runtime. Using service registries, systems can be assembled dynamically.
- Often, services are interoperable – they can be used by systems implemented on various platforms.

In addition to these characteristics, services should be designed to be coarse-grained and encapsulate functionality relevant from a business perspective – although nobody can say what this really means! Services are typically but by no means exclusively used by explicitly-modelled business processes. Finally, like any good IT system, they are secure, transactional and manageable.

Opinions differ over whether these characteristics are really so different from today's well-constructed enterprise systems. However, what is obvious in our view is that models play a central role in the definition and operation of service-oriented systems:

- Message schemas are data structure models that specify required and optional content, as well as how message formats change during the evolution of a service.
- Interaction patterns between services are defined using models: for example, communicating state machines are useful notations to describe valid sequences of messages as well as exceptional cases.
- The levels of quality provided by a service provider, and required by a service consumer, are basically models that are evaluated and checked for compatibility.
- The runtime repository basically makes the model information available to runtime queries.
- Finally, interoperability can be achieved by generating implementations and bindings for various platforms from the same authoritative model.

So the central idea to SOA in our view is to establish an interface contract first! The first thing you specify when developing systems is how the communication partners actually interact – which is *independent* of communication technology and *independent* of implementation platform. Rather, you specify message formats, interaction patterns, and quality of service contracts on an abstract and formal level. That basically means: use MDSD based on the architectural process described in Section 13.4.

Figure 7.15 shows a simplified metamodel for services. It also shows how to connect the SOA 'world' with the component 'world' described in the previous section.

In the context of SOA you often see two pictures showing a spaghetti-like system with all those interconnections labelled 'old' on one side, and a nicely-ordered set of components connected through a single bus to which all components are attached, labelled 'service-oriented' on the other. This leads people to think that in SOA all systems must be physically connected through a single technical infrastructure often Web Services). Nothing could be more wrong! If you connect all kinds of systems through the same infrastructure, you will have a hard time addressing non-technical requirements such as throughput, interoperability, or performance. The bus you often see on such PowerPoint slides must be interpreted as a 'logical bus' – that is, a common, model-based communications infrastructure through which messages can be mapped to various communication technologies, and service endpoint be implemented on various platforms.

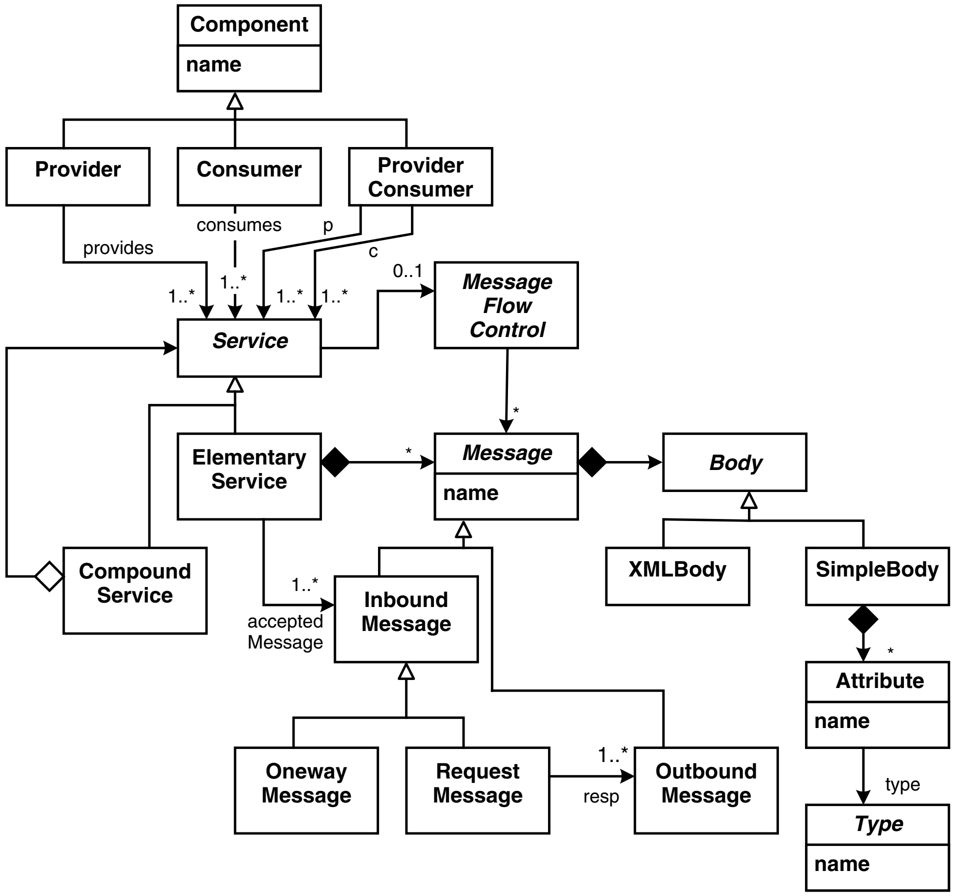


Figure 7.15 A simplified metamodel for services

7.9.2 BPM

As with SOA, there is no commonly-agreed definition of BPM, but there seems to be a kind of consensus among industry leaders:

- BPM deals with design and control of (rapidly changing) business processes, which leads to tasks of structuring, automation, and optimization of these artifacts.

- Business processes connect people with available information technology and material.
- Process definitions and implementations have to be flexible, so that you can change them to meet the value creation chains of your business.
- BPM respects the complete lifecycle of a business process, which consists of *definition* (standards-based graphical modeling, process simulation, business rules), *creation* (code generation), *execution* (integration, automation and workflow), *monitoring* (business activity monitoring, dashboards) and *optimization* (ability to adjust business rules dynamically).
- BPM is not a product and none of the following single product categories can be said to cover BPM completely: workflow, enterprise application integration (EAI), business activity monitoring (BAM), rules engines, process-simulations. Ideally they can be part of a system that supports BPM (BPMS).

Some standards exist, such as BPMN (Business Process Modeling Notation [BPMN] and BPML (Business Process Modeling Language [BPML]) from OASIS [OASIS], or XPDL (XML Process Definition Language, [XPDL]) from WfMC [WfMC]. In practice we observe quite a confusion, because respective tool suppliers often try to 'improve' their products with a BPM label. As a consequence BPM products essentially suffer from not being comparable at all.

An isolated MDSD-view of BPM should be obvious: models and transformations are already essential concepts of BPM in order to achieve it goals.

7.9.3 SOA and BPM

An intersection of SOA and BPM exists: modeling and specification of business processes on one hand, and respective infrastructure software (middleware) on the other. SOA covers business process modeling through BPEL (Business Process Execution Language, [BPEL]) which is based on and coupled to Web Service technology. BPM covers business process modeling through more abstract language concepts and (graphical) notations (BPML/N). So from a specific point of view we can say that SOA is a bottom-up evolution and BPM a top-down one. The middleware in the intersection placed by SOA is mainly referred to as ESB (Enterprise Service Bus), which should be viewed as a logical bus for messaging, service composition, and orchestration, as we pointed out in Section 7.9.1. BPM, on the other hand, comes with BPEs (Business Process Engine) or BPMS (see above).

It is not enough to say that the intersection between these disciplines is not empty: there is even conceptional mismatch in their intersection. For example, BPEL suffers (for now) from concepts of human resources involved in a workflow, so essentially BPMN/L cannot be mapped to BPEL yet.

Certainly we can hope and expect that these mismatches will be eliminated some day by a proper standardization process. At least until then a distinguished approach may serve as a way to find a useful synthesis for SOA and BPM – the principle of Separation of Concerns. BPM needs SOA in order to be as flexible as required, but not the other way round. If there is a decision in your company to have both architectural concepts combined in an enterprise architecture, it may be reasonable to use BPM concepts with respective middleware for the business process layer, and SOA concepts with respective middleware on a business service/component layer, which is placed logically below the business process layer. In that case, you do not use SOA (for example, BPEL) to model and maintain business processes.

To conclude this section we will try to place MDSD/MDA in the context of a synthesis of SOA and BPM:

- MDSD/MDA can provide standards-based modeling of business processes (that is, standardized, MOF-based metamodel definition for BPM) and respective 'generic' notations (for example BMPN as a UML-profile). The OMG has already set up corresponding activities.
- MDSD/MDA can provide a sound and complete architecture-centric MDSD-production line that supports all layers (business processes, service/components, entities, persistence) of an enterprise architecture in a consistent and defined way. It can take advantage from the knowledge of all these layers and their interaction in order to check comprehensive constraints and generate infrastructure code even between the layers. Hooks for version handling of services or business process definitions and support for transactions or compensations may be generated. Different model-to-code transformations may be implemented to realize a 'fan out' that is capable of adopting different platform technologies. But before you can automate an architecture you have to define and build it – at least partially (see Section 13.4).
- Some people claim that MDSD is useful for *service enabling* – that is, SOA-oriented re-engineering of monolithic legacy applications in to make them usable for BPM – since models derived from existing applications can serve as the essence of those applications. We think that this idea is quite ambitious but at least worth mentioning.

8 Building Domain Architectures

In this chapter we discuss how a suitable domain architecture can be constructed for an existent target and platform architecture, as described in the previous chapter. We look at building DSLs, general best practices for building the transformation architecture, as well as a couple of detailed technical aspects.

Please do not consider this chapter an outline for the process, because the best practices introduced here can affect the target architecture and/or the reference implementation. Questions about the development process are addressed in Part III of the book.

8.1 DSL Construction

This section contains some hints about constructing the DSL itself. Note that apart from the items listed in this section, you can find a great number of examples for DSLs throughout the book. An important ingredient for building DSLs is of course metamodeling, which is described extensively in Chapter 6.

8.1.1 Choose a Suitable DSL

When trying to find a DSL for a certain domain, you should always take into account the required amount of variability you need to express. So, before embarking on a fully-fledged, graph-like textual or graphical DSL, ask yourself whether or not simpler forms of DSLs are enough. Figure 8.1 shows the alternatives as adapted from Krzysztof Czarnecki’s work.

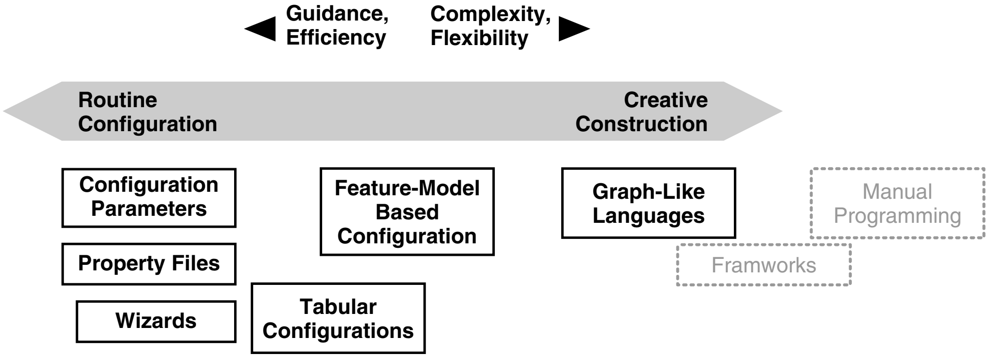


Figure 8.1 Various ways of building DSLs

If you only need to do routine configuration, then simple property files or wizards are probably enough to express the variability that is needed to describe a product. If you need more freedom – the variability space from which you have to select is larger – then tabular or tree-based configurations are more appropriate, feature models (Section 13.5.3) being the most powerful of the DSLs that follow a primarily configuration-oriented approach. Up to this point, using the DSL consists basically of selecting from a number of options – that is, configuration work.

If this is not enough, you have to adapt a creative construction approach to modeling, in which you can creatively instantiate, arrange, and connect the modeling elements available in a DSL using powerful graphical or textual languages. This is of course much more powerful, but also provides much less guidance to the developer. If you need to express structural variability, especially – for example, 'I need one of type A connected to this instance of type B, and then I need another three of type C' – then the configuration-oriented approach quickly become awkward and you have to revert to creative construction.

Often you will use combinations of such DSLs to describe a complete system. In the case study in Section 16.3.3, for example, we use graphical and XML-based graph-like languages to define the system structure, but use simple configuration parameters derived from feature models to characterize the communication.

Note how you can find frameworks and in general, manual programming, to the right of the graph-like languages in Figure 8.1. They provide the most freedom and flexibility, but also require the highest effort, provide the least guidance, and are thus the most complex approach.

8.1.2 Configuration and Construction – Variants

We can now see two 'kinds' of DSL: those that support creative construction and those that provide means for configuration. It is very interesting to combine the two. Consider the meta-model in Figure 8.2, which serves as the basis for a creative-construction DSL that describes data structures.

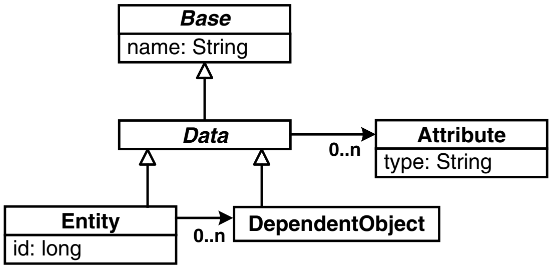


Figure 8.2 A simple metamodel for data structures

Using this metamodel you can define all kinds of data structures, such as those defined in Figure 8.3.

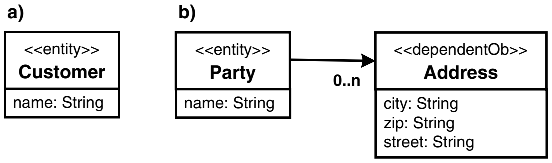


Figure 8.3 Two example models based on the metamodel defined in Figure 8.2

Now, assume that you want to describe variants of model. For example, in the context of people and addresses, the following variants might make sense:

- A party may have one or more address
- A party may or may not store telephone numbers
- In the case of telephone numbers, you may want to store the country code
- Addresses may have a *state* field, typically in the US

To express these variants you can overlay a feature model – a routine configuration DSL – over the model built with a creative construction DSL. Selecting or deselecting certain features in the feature model then results in the inclusion or removal of specific model elements. You have to associate specific model elements in the structural model with features in the configuration model: Figure 8.4 shows an example.

To make such an approach a practical reality, however, tool support must be available. The authors have built a prototype that uses the following components:

- The structural model is built with any UML tool.
- The feature model is built with pure::variants, a feature-model based variant management tool [PV].
- openArchitectureWare [OAW] is used to process and 'relate' both of these models.

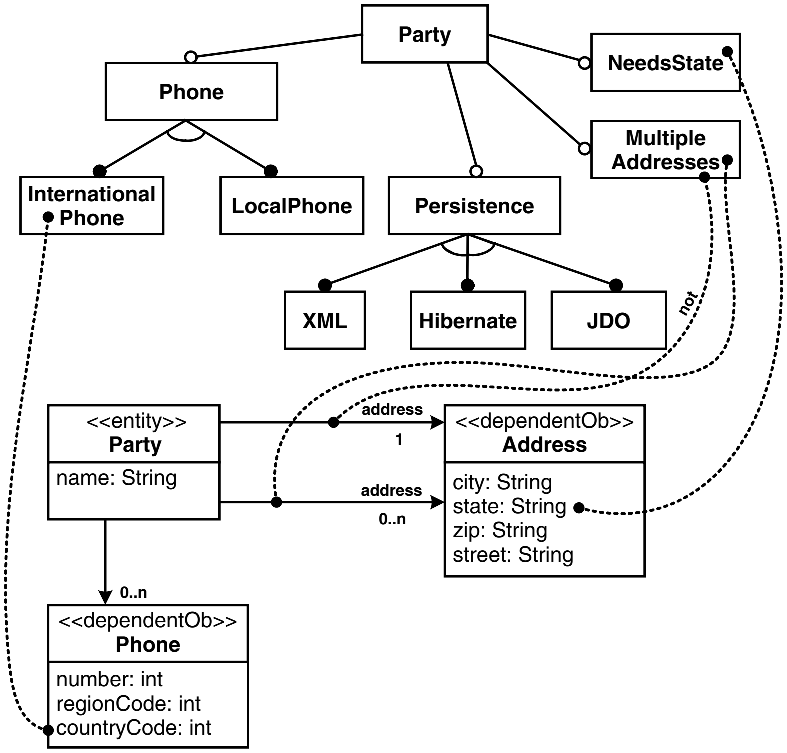


Figure 8.4 Attaching model elements to a configuration (feature) model

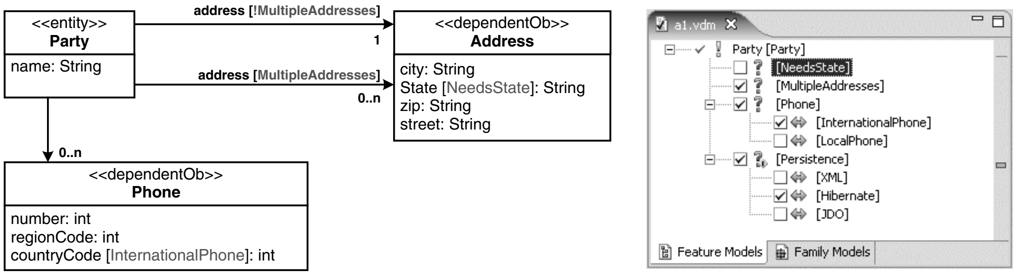
After reading both models, openArchitectureWare modifies the structural UML model based on the features present in the feature model. The association between UML model elements and the features is based on a special notation in the name (other approaches using tagged values or stereotypes could have been used, too). Figure 8.5 shows an example.

Krzysztof Czarnecki and his team at the University of Waterloo have built a similar (and much more sophisticated) tool based on their feature modeling plug-in [FMP] and the Rational Software Modeller UML tool. They provide good model integration in the same tool (Eclipse Platform) as well as various verifications of the correctness of model variants.

8.1.3 Modeling Behavior

Most examples of DSLs presented in this book, and also in most of the other publications on MDSD, describe structural aspects of a software system, things like component structures, persistence mapping, or input forms. However, most systems also have behavioral aspects that need to be modelled. In this section we want to provide a couple of hints on how to do that.

The easiest way to model behavior is to reduce the behavior to simple descriptive tags, if possible. For example, if you need to describe how the communication between components



- ✓ Party [Party]
  - □ [NeedsState]
  - ☑ [MultipleAddresses]
  - ☑ [Phone]
    - ☑ [InternationalPhone]
    - □ [LocalPhone]
  - ☑ [Persistence]
    - □ [XML]
    - ☑ [Hibernate]
    - □ [JDO]

Figure 8-5 Using special names to associate UML model elements with a pure-variants model

happens (for example synchronous, asynchronous), and if you are able to identify a limited number of well-defined behavioral alternatives, then the behavior can be described by just marking it with the respective alternatives. You don't have to actually *describe* the behavior, you just denote which alternative you need, and the transformation or the code generator can make sure the generated system does indeed behave as specified. Selecting a valid option can be as easy as specifying a specific property, or as complex as a sophisticated selection based on a feature diagram. This is an example of using routine configuration (see the previous sections) for behavior. An example of this approach can be found in Section 16.3.3.

Another relatively simple alternative for describing behavior is to use a well-known formalism for specifying specific kinds of behavior. The classical example of that approach is state charts, other examples include first-order predicate logic and business rule engines. Of course such approaches only work if the required behavior can actually be described in the selected formalism. If so, this has a number of advantages: the description and the semantics of the behavior are often quite clear, and editors and other tools are available. It is also usually well documented how to implement 'engines' for the particular formalism, to execute the specifications.

Within the constraints of the selected formalism, this approach already constitutes creative construction, rather than configuration. An example of this approach can be found in Section 17.4.3, where state charts are used to describe business processes.

In case a given formalism does not work, you might want to invent your own. For example, in the insurance domain, you might want to use textual languages that specify verification constraints for insurance contracts, or that can be used to define certain mathematical algorithms relevant in that domain. In that case you have to define the formalism (the language) yourself, and you have to build all the tooling. Writing engines might not always be easy, because it's not trivial to get the semantics of an 'invented' formalism right.

The last alternative you have is to use existing Turing-complete languages, such as a 3GL or UML action semantics languages (Section 12.2.6). Here you can specify any kind of behavior – albeit using a very general language that is *not* domain-specific for the kind of behavior at hand.

Which alternative should you use? The alternatives described above are ordered by increasing flexibility and by increasing complexity. If you can clearly identify behavioral alternatives, then using simple configuration to chose one of them is certainly best. If this is not possible and you need to revert to creative construction, you should still use a DSL that is as close as possible to the kind of behavior you need to specify. Only as a last resort should you fall back to action languages or 3GLs in general.

Note that it is always necessary to associate an individual behavior with a structural element. For example, a state machine can be implemented as part of a component, or an insurance-related mathematical algorithm defined to be the implementation of an operation or activity described structurally somewhere else. Structural 'behavior wrappers' provide a natural point of integration between structural models and behavioral models. Considering again the idea of using the most specific possible way to specify behavior, as explained in the preceding paragraph, you should then define certain subtypes of structural elements that implement their behavior with a certain formalism, rather than just allow developers to 'implement' the structural element.

To illustrate the idea, consider the example from Chapter 17. The main structural element is the *Component*. Instead of just saying 'components have operations, you can implement them with an action language', you might identify the following submetatypes:

- Process components represent business processes: behavior is modelled using state machines.
- Business rule components capture (often changing) business rules: behavior is modelled using predicate logic.
- Insurance contract calculation components are implemented with a specific textual DSL.
- 3GLs are used to implement the behavior for the rest of the components, which should be limited in number.

A final word on implementing behavior: while structural models almost always result in generated code that resembles the modelled structure in one way or another, behavior often lends itself to interpretation. So instead of generating state machine implementations, you might run an interpreter in the business process component that interprets state machines specified in XML: from the model, you just generate the XML representation. Another example could be an interpreter for insurance contract calculations. In this case you would embed the interpreter in the respective component. (Section 8.4 contains a more detailed look at interpreters in the context of MDSD.)

8.1.4 Concrete Syntax Matters!

From the perspective of model transformation or code generation, the concrete syntax of a DSL does not really matter as long as there is a way of transforming it into the abstract representation with which the generator works, and as long as the metamodel suitably represents the concepts that should be modelled with the DSL. However, the DSL is the 'user interface' for the metamodel — that is, application developers must be able to read, write and understand the models properly. This is a very important issue that should not be overlooked. After all, the idea of MDSD is to provide more efficient means of expressing domain concepts: this efficiency strongly depends on the concrete syntax. This is the main reasons why we think UML + profiles is not enough for MDSD.

Coming up with a suitable concrete syntax theoretically or on a flip chart is relatively easy. However, especially for graphical syntaxes, building the necessary editor can involve considerable work. When deciding for a specific form of concrete syntax, therefore, you should always consider the tooling you can use to build the respective editor. Section 11.3 provides a number of hints here.

When considering the concrete syntax, bear in mind that you typically don't 'draw' the models on a flip chart, but rather in an interactive editor. This means you might have features such as zooming, panning, context menus and buttons, or folding at your disposal. The way you interact with the diagram, and the means to adapt the diagram to specific views/parts are an important aspect of editor design, and can make DSLs that look crowded on paper perfectly feasible.

8.1.5 Continuous Validation of the Metamodel

The importance of continuous validation and evolution of a domain's formal metamodel is best demonstrated with a simple example (see Figure 8.6).

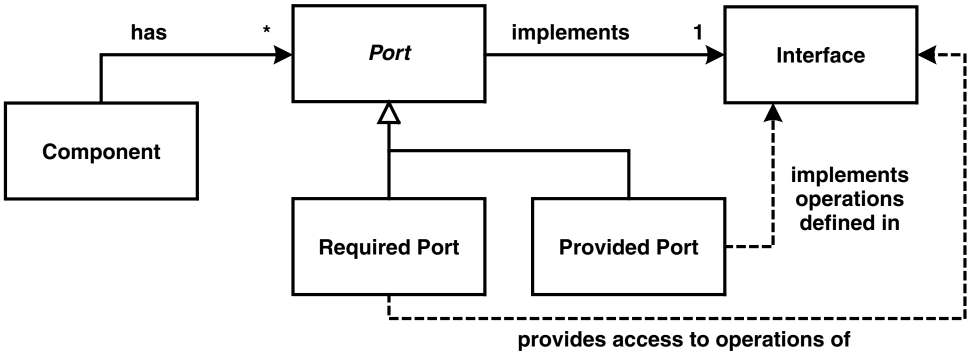


Figure 8.6 Simple components metamodel

The metamodel must be checked with the aid of domain experts. The development teams must be able to use it smoothly and without misunderstandings. To this end, it is useful to use the metamodel's terminology in all discussions with stakeholders – the metamodel can be considered a grammar for building valid sentences in the respective domain. In most cases, errors and inconsistencies that have sneaked into the metamodel are easily uncovered this way. In discussions with stakeholders, sentences such as the following can be used:

- A component has any number of ports.
- Each port implements exactly one interface.
- There are two kinds of ports: required ports and provided ports.
- A provided port implements the operation of its interface.
- A required port offers access to operations that are defined in its interface.

As soon as something cannot be expressed easily with the metamodel's vocabulary, this means that the formulation does not correspond with the metamodel, the metamodel has errors, or it is imprecise.

This technique is very popular in domain modeling. In his book on domain-driven design, Eric Evans calls this technique *ubiquitous language* [Eva00]. His book describes many useful techniques for the design of domain-specific frameworks.

8.2 General Transformation Architecture

8.2.1 Which Parts of the Target Architecture Should Be Generated?

We discussed the correlation and balance between the MDSD domain, the MDSD platform, and its target architecture, at length in the last chapter. Now, we must only draw the dividing line between code that is generated and code to be manually programmed.

In the context of a domain architecture, one will always generate those artifacts that on one hand are not covered by the platform, and on the other, cannot be described well and compactly using a DSL.

A reliable sign that generation has been taken too far is usually the introduction of typical programming language-based constructs such as loops into the DSL[1]. The DSL should be mostly declarative and not mutate into a classic programming language – if this occurs, one should actually use a programming language and integrate the manually-written code in the generated code.

8.2.2 Believe in Reincarnation

The final, implemented application should be created using a fully-automated build process that includes the regeneration of all generated/transformed artifacts. Once a single manual step is required as part of the build process, or a single line of source code must be adapted manually after regeneration, it is only a question of time until MDSD is given up in favour of traditional, non-generative development, since manual adaptation of generated artifacts is tedious and error-prone.

This does not mean that you should – or must – generate 100% of an application. It is absolutely okay to continue to implement parts of it in a 3GL, as long as the 3GL is suited for this purpose. This best practice only states that those parts that *are* generated must be *completely* generated, so that the complete system can be recreated in one go. Subsequent adaptation of generated artifacts is off-limits!

8.2.3 Exploit the Model

The information contained in a model should be exploited as much as possible, to avoid duplication and minimize the amount of manual work. This means that you will usually generate more than source code: build and deployment scripts, skeletons/fixtures for component tests, maybe even test implementations, test data and mock objects, database generation scripts, data migration and filling scripts, simple interfaces for master and test data maintenance or parts of the documentation. All this can save you a lot of trouble and effort in routine project work. The right

[1] Executable UML is an exception here – see Chapter 12.

balance between the effort of automation and that of repeatedly executing the same steps manually should be found based on ‘sustainability’ considerations – that is, based on what is reasonable and economic over longer periods in the sense of extreme programming (»three strikes and you automate«). A rule of thumb that applies here is: compare a rough estimate of the required key strokes and user gestures to reach a specific goal, then select the method that requires the smallest amount of work. Keep in mind that manually-created artifacts must often be recreated or adapted.

Please do not take this advice the wrong way and generate things that are already available on the market. Choose between buying, creating, and the use of Open Source software. Often others have already done the necessary work, and you can save yourself a lot of the thinking and typing that would be needed if you reinvented the proverbial wheel.

Generation of Component Configurations

In the last chapter we explained that frameworks and DSLs suit each other insofar as DSLs can be used as a ‘configuration language’ for frameworks. For complex platforms, these configurations are often not source code, but configuration files in the broadest sense, which these days are often rendered in XML. In most cases, such configurations can be easily generated from the model, because the necessary information is often already contained in it. Here are some examples:

- The EJB deployment descriptors can be created from the model if some additional annotations are defined that, for example, define the transactional behavior of operations.
- The definitions for the Struts page flow framework can be generated from state diagrams or activity diagrams. These configurations, too, are specified in XML.
- Hibernate, a Java persistence framework, also works with XML-based configuration files that describe the classes and attributes to be persisted. These, too, can be easily generated.
- CORBA IDL is yet another candidate. The creation of IDL definitions for model elements can also be easily automated. From these, the CORBA artifacts are derived in a further generation step.

The CORBA example is interesting insofar as it demonstrates that the generated artifacts themselves can serve as the input (model) for further generators – cascading. Of course we could also generate the CORBA stubs and skeletons directly from our model, but the idea of reusability forbids this approach. This shows that the sequencing of several transformations – with well-defined intermediate formats, here CORBA IDL – is an extremely useful approach. (Cascading is explained in Section 8.2.8, and an example can be found at the end of Chapter 17.)

Support of the System Architecture

No system can be built with an exclusive focus on *software* architecture. A system architecture that makes statements about existing machines, processes, and the assignment of software to them is always required. In spite of the progress in the field of MDSD, we are not yet able to generate hardware, but we can significantly support software deployment.

Assume we wish to run a complex system with many components on a server farm. To this end, the following things must be done:

- The components must be installed on the respective machines.
- If necessary, the correct tables must be created and initialized on the correct database instances.
- If necessary, the infrastructure must be configured. A typical example are load balancers.

The respective system structures can simply be modeled, for example with UML deployment diagrams. Based on these models, one can then generate the required artifacts, such as installation and configuration scripts.

Here is an example: in the infrastructure of distributed, embedded systems (see Chapter 16), the topology of the distributed system is defined in a model. Moreover, the network connections between the single nodes are described. The generator then creates the complete images for the required nodes, which can be deployed directly and run on the respective system node.

8.2.4 Generate Good-Looking Code – Whenever Possible

It is unrealistic to assume that developers will never see the generated code. Even if developers don’t have to change generated code, for example by inserting manually-written sections, they will be confronted with it if they debug the generated applications with conventional tools, or if they have to check the generator configuration. How can one make sure that application developers understand the generated code and are not afraid of working with it?

The prejudice that generated code cannot be properly read, understood, or debugged is deeply rooted. In some cases, this prejudice is used as an excuse for not applying Model-Driven Software Development at all. This can be avoided if you try to generate ‘good’ code. Consider the following points:

- You can generate comments. In the templates, you have most, if not all, necessary information at hand to produce sensible comments. Typically, generated comments are not static text, but are based on information taken from the model like the rest of the templates.
- Because the options for working with whitespace in code generation templates are often quite limited, you must decide whether the templates or the generated code will be indented correctly: the generator often cannot distinguish if an indention serves to structure the templates or to structure the generated code. In many cases, it is best to focus on the templates’ readability, and use a downstream pretty-printer/beautifier that can format the code correctly. Pretty-printers are available for the majority of all programming languages, as well as for XML.

- A third and very useful technique is the application of *location strings* that identify the transformation or the template used, as well as the underlying model elements in the generated code. A location string might look like this:

```
[2003-10-04 17:05:36]
GENERATED FROM TEMPLATE SomeTemplate
MODEL ELEMENT aPackage::aClass::SomeOperation().
```

The use of this best practice is important in increasing the acceptance of code generation among developers. In essence, it aims at applying the same quality standards and style guidelines to generated code as to non-generated code. Logically-correct indentation regarding control structures is of particular importance. If you convert a high-quality application prototype into templates, you should have all necessary comments at hand to integrate them into the templates.

The only restriction/exception to this best practice concerns the generation of performance-optimized code. In such cases, one must often resort to constructs that impair the code's readability. These cases should be explicitly identified and described, and the generated code should be managed separately from all other code. Manual implementation too, of course, can make it necessary to sacrifice structure to save performance.

The fundamental statement behind this best practice applies not only to generated code, but also to hand-written code: source code is no longer primarily written for machines, but instead for reading by other source code users – humans.

8.2.5 Model-Driven Integration

In many cases you will have to integrate your system developed using MDSD with existing systems and infrastructures. Software projects that take place in isolation are rare in practice. In most cases, software is developed in the context of existing system environments that have remaining lifetime ahead of them. In addition, developers often wish to replace legacy systems incrementally with newer and better suitable functionality that should be realized with modern technology over time.

Integration typically includes the systematic mapping of various APIs, as well as of the required data and protocol transformations between these APIs. Depending on the integration strategy, integration code must be added either to the application to be developed, or into the existing legacy application that is to be integrated. The required artifacts often also include suitable data conversion scripts for one-time use.

Data and interface mappings between systems are most valuable if they are mapped in a model. You therefore need to approach the integration issue as a part of MDSD. Do not ignore it! Define a technical subdomain (a specific DSL) for model-driven integration. If things get more complicated, consider using a technical subdomain for each of the systems to be integrated. Define DSLs in those subdomains that support an exact description of the mapping rules between model elements and existing legacy applications. Use automation to simplify the shutting down of legacy systems to the extent that no expert knowledge is needed for this step.

Integration with existing systems is a strength and not – as some people argue – a weakness of MDSD.

In the case of the integration of two different, model-driven systems, it might prove reasonable to structure the integration code in such a manner that the required knowledge about implementation technologies is not needlessly duplicated in the template source code of both systems.

When dealing with simple integration tasks, a technical subdomain can be unnecessarily complicated, and it may suffice to use UML-tagged values or the like in the models that are used for modeling the actual application. However, you should use this approach only if the integration information does not unnecessarily impair the model's clarity, and if the integration is of a permanent nature and not applied to a legacy system with a limited life span. In the latter case, you should make sure that the integration code can be removed easily as soon as it is no longer needed. Otherwise, 'dead code' will only contribute to the architecture's pollution. Use an anti-corruption layer [Eva03], and specify the mappings via external model markings – overlaid models that add a specific aspect to the model.

Try to automate the step-wise shutting-down of legacy systems to the extent that you can label the parts to be shut down in a DSL. You will make the lives of future (developer) generations much easier, because the people who will shut down the last parts of a legacy system three years from now might know very little about the details of the integration code.

8.2.6 Separation of Generated and Non-Generated Code

When only parts of an application are generated – often the case in many of today's scenarios – the gaps must be filled with manually-written code. However, the modification of generated code harbors a wealth of problems in terms of consistency, build management, versioning, and consistency between the model and generated source code, particularly due to repeated regeneration, even though the latter aspect is – from an exclusively *technical* point of view – mastered by modern generators.

If the files with the generated code are never modified, the generated code can simply be deleted if necessary and a complete regeneration can take place. If generated code must be changed manually, this may only occur in specially-designated areas that will not be overwritten during regeneration (often called *protected regions*).

Therefore, keep generated and non-generated code in separate files: in most cases, it is even sensible to use separate directories. Never modify generated code! Design an architecture that defines clearly which artifacts are generated and which must be created manually. In the context of this architecture, you should also define how generated and non-generated code is combined in terms of the target architecture. Interfaces, abstract classes, delegation, and design patterns such as Factory, Strategy, Bridge or Template Method [GHJ+94] are suitable means in the realm of object-orientation. In languages that are not object-oriented, one can for example work with includes.

The separation of generated and non-generated code forces architects to choose a design that very cleanly separates various aspects – in our view, a highly desirable side-effect. As a consequence of this best practice, generated source code can be seen as a disposable product that need not even be versioned, which can reduce consistency problems. In all cases in which manually-implemented code and generated code are used together, however, inconsistencies can occur if the model is altered in such a way that it is no longer structurally or semantically compatible with the handwritten code, even if the handwritten code has been completely delegated to separate classes.

If, for performance reasons, or because the target language doesn't offer any options for consolidating different artifacts, handwritten code must be inserted into generated code directly, the introduction of protected areas is inevitable. Please do this only if such exceptional conditions require this approach!

The best practice described here can be generalized in the sense that different generators can be used to generate different parts of the system, such as various technical subdomains. In such cases, handwritten code can be seen as the result of a very special 'generator' – the human programmer. The system architecture must of course make allowances for all these different aspects.

8.2.7 Modular Transformations

To enable the reuse of (parts of) transformations, it is advisable to modularize transformations. Depending on the transformation language used, this can (and should) be done using the concepts of structured or object-oriented programming, just as in classical programming. Among these are subroutines/procedures, classes, or loosely-coupled components (often called *cartridges*) that are then responsible for generating different layers or aspects of the target architecture. They can also be exchanged separately. Such means for structuring are usually tool-specific (see Chapter 11), yet very important – use them!

Figure 8.7 shows how this approach might look in the context of a template-based approach. This example is about extending the standard UML-Java mapping in such a way that JavaBeans can be generated. For each attribute in the model, both an attribute in the Java class and also the respective getters and setters should be generated.

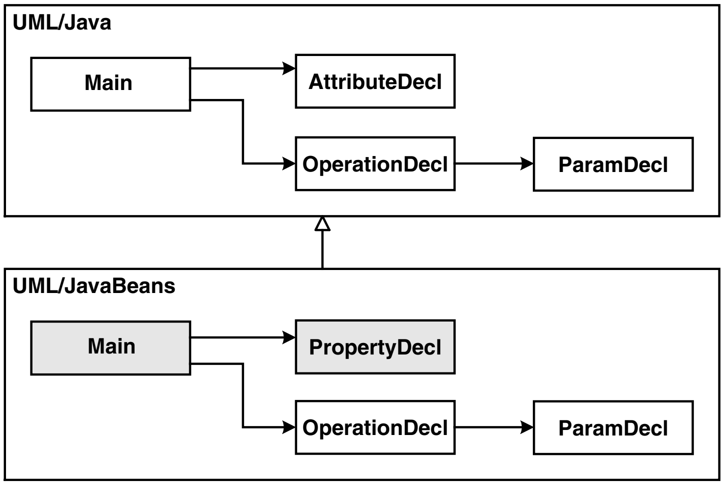


Figure 8.7 Template modularization

In this example, the *Main* template for the Bean is newly-defined, so that it uses the newly-defined *PropertyDecl* instead of the 'old' *AttributeDecl* for each attribute. The other templates remain in use as they are, that is, unchanged.

A form of modularization that goes beyond the scope of this example is the breaking up of a transformation into several transformations that are then carried out sequentially. Consider the example in Figure 8.8:

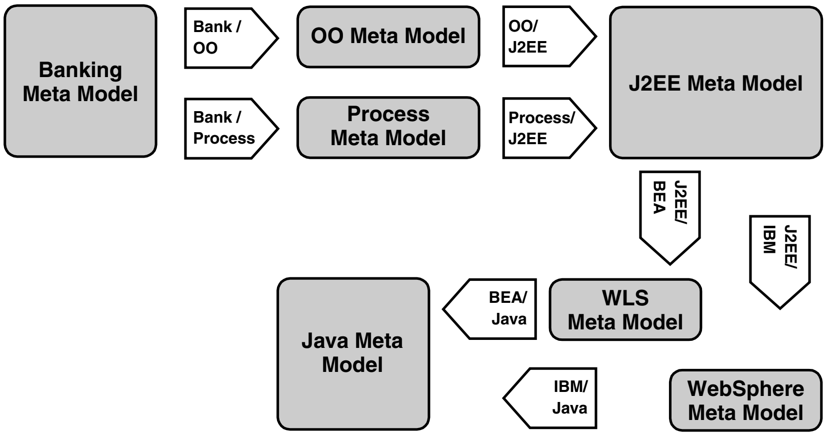


Figure 8.8 Modular transformation

In this figure, the model of a banking application is transformed into an application that runs on J2EE, specifically on two different application servers, *BEA WLS* and *IBM Websphere*. However, the transformation does not take place in a single step, but in several phases:

- First, the process aspects of the banking domain are mapped to a process model and the remaining aspects to an OO-model.
- In the next step, the two models are mapped to J2EE.
- Afterwards, this J2EE model is mapped to an application server-specific model.
- In the last step, the two models are converted into code.

Now imagine that we wish to create J2EE code for call-center applications. Since the transformations are modular, we need only exchange the first part of the transformation. The subsequent transformations can be reused without changes, which saves us a lot of work (see Figure 8.9).

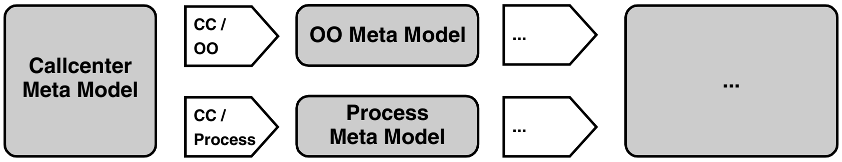


Figure 8.9 Adaptation of the transformation process to another professional domain – from banking to call centers

If we now wish to port both software families to .NET, we only have to exchange the second part of the transformation, as shown in Figure 8.10

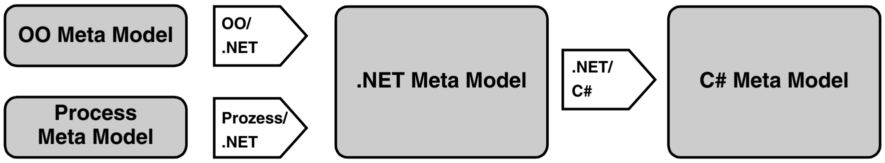


Figure 8.10 Adaptation of the transformation process to another technology

If we used only a single, direct transformation, this kind of reuse would be unthinkable.

In contrast to the OMG's opinion, we recommend not to modify the intermediate models manually or add further markings. They are merely needed as an 'interface' between the various transformation steps, an agreed data model.

If we need additional information to control or configure subsequent transformations, we should implement those markings in the form of external model markings. This approach is not only clean in terms of separation of concerns (»aspect-oriented modeling«), but also much more practicable with the currently-available tools. Figure 8.11 shows why: transformations are used only inside the generator tool – that is, the transformation result is not editable in the modeling tool. The (multi-step) transformations are used exclusively to modularize the transformation process. Generator-internal JUnit tests are used to verify that the transformation works as expected.

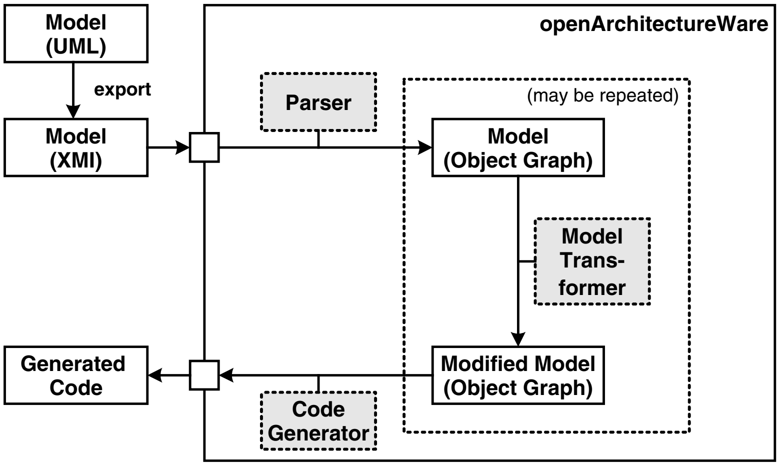


Figure 8.11 Using model transformation inside the transformation framework illustrated by openArchitectureWare

This discussion also makes it obvious that one person's target model is another's source model. Thus multi-step transformations constitute the basis for the reuse of transformation steps. Cascading, model-driven development points in the same direction.

8.2.8 Cascaded Model-Driven Development

We saw in the previous section that modularizing a transformation process using model-to-model transformation is useful. However, the question remains of into which steps a potentially complex transformation process should be broken down. The MDA uses the metaphor of PIM and PSM to guide these decisions: you should model your business logic in the PIM and then transform it into the PSM, from which you finally generate code. In practice we find that this guideline is often not specific enough. Based on our own experience, we recommend a different approach.

We start with architecture-centric MDSD (see Section 4.3). This means that you provide MDSD-support for your (technical) software architecture. The metamodel contains the architectural building blocks, and models describe systems from a technical point of view. Once you have built this infrastructure, you can cascade additional layers of MDSD-infrastructures onto it, as shown in Figure 8.12.

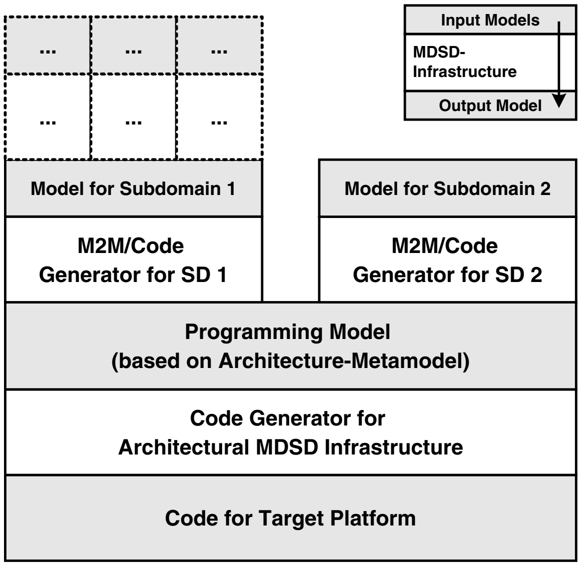

Input Models
MDSD-Infrastructure
Output Model
...
...
...
...
...
...
Model for Subdomain 1
Model for Subdomain 2
M2M/Code Generator for SD 1
M2M/Code Generator for SD 2
Programming Model (based on Architecture-Metamodel)
Code Generator for Architectural MDSD Infrastructure
Code for Target Platform

Figure 8.12 Cascaded Model-Driven Software Development.

These additional layers typically address a more limited (sub)domain and become less technical, and thus more functional, with every layer. The idea is that the more specific higher layers map the more specific concepts to the already-defined more general (architectural) concepts of the lower layers. This automatically leads you to the 'right' modularization steps, as you add more specific layers *as you need them*. Eventually you will arrive at completely non-technical, functional model that your domain experts are able to specify. You can see this approach in action in the case study in Section 17.4.3.

Cascaded MDSD can also help you out of another dilemma. If you are working with a mainstream platform such as J2EE or Spring+Hibernate, you often have the chance to use third-party off-the-shelf cartridges (a cartridge is a ‘piece of generator’ for a certain architectural aspect). The problem then often becomes how to combine these different cartridges, especially if they have been developed independently and thus use different metamodels – different stereotypes, tagged values, and so on. You certainly don’t want to model things several times merely to be able to use various incompatible cartridges.

We recommend the approach shown in Figure 8.13. As usual, you start by defining your own project-specific architecture-centric metamodel. Applications are modelled using the concepts defined in that layer. Your project-specific cartridge uses model-to-model transformations to create the models required as the input for the third-party cartridges. These in turn generate the code for their specific architectural aspects. Your cartridge generates the code to glue together the artifacts generated by the various third-party cartridges.

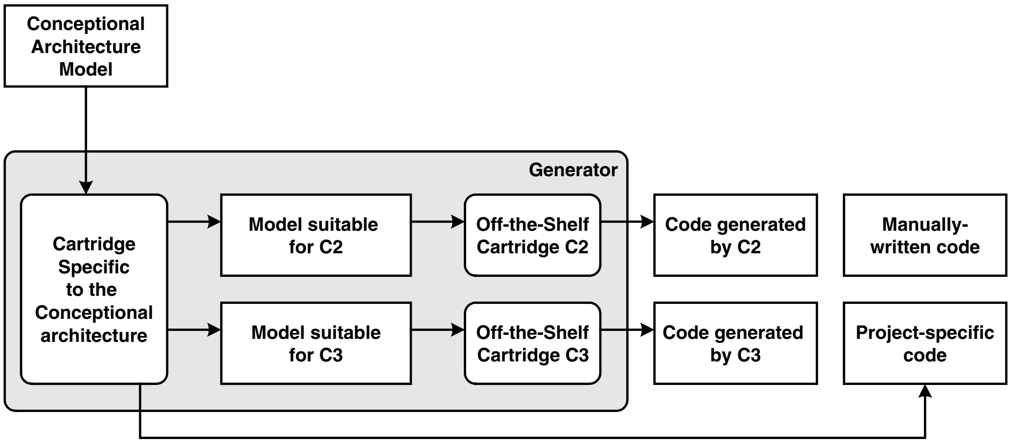

Conceptional Architecture Model
Generator
Cartridge Specific to the Conceptional architecture
Model suitable for C2
Off-the-Shelf Cartridge C2
Code generated by C2
Manually-written code
Model suitable for C3
Off-the-Shelf Cartridge C3
Code generated by C3
Project-specific code

Figure 8.13 Using cascaded MDSD to shield technology-specific cartridges from the modelling layer

8.3 Technical Aspects of Building Transformations

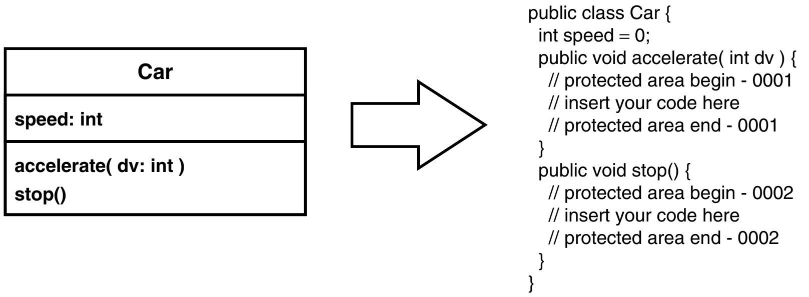
In addition to the best practices described in this section, we describe a number of details of code generation and model-to-model transformations in the two following chapters.

8.3.1 Explicit Integration of Generated Code and Manual Parts

Explicit integration means that in the beginning, the generated code is totally independent of handwritten code. It is up to the developer to integrate the artifacts. There are only very few, rare cases in which both kinds of code are factually completely independent – non-generated code often depends on generated code, since they are commonly used together in a system’s context.

The simplest integration method is to create protected areas in the generated code in which the developer can insert handwritten code. These areas are designated in such a way that they can be read by the generator, so that the handwritten code will not be overwritten during subsequent generator runs.

UML tools typically work in this manner. Here, class stubs are generated from model data, into which the developer then integrates the behavior. Figure 8.14 shows this:



```
public class Car {
  int speed = 0;
  public void accelerate( int dv ) {
    // protected area begin - 0001
    // insert your code here
    // protected area end - 0001
  }
  public void stop() {
    // protected area begin - 0002
    // insert your code here
    // protected area end - 0002
  }
}
```

Figure 8.14 Generated code with protected areas

This approach has a number of disadvantages:

- The generator is more complex, because it must recognize the management, recognition, and preservation of protected areas.
- Preserving the contents of the protected areas is not always easily accomplished. In practice, code sometimes gets lost.
- The separation of generated and handwritten code dissolves, because both are in the same file/class.

The last point is the most problematic one because the developer must work in the generated code. For this purpose, they must first understand it, which is not always easy. Other aspects of software development such as versioning are also made more complicated.

Other mechanisms for integration should therefore be considered. A solution that is often applied (and one that also handles integration with the platform) is *3-tier implementation*. In many cases, the system components that must be generated consists of three kinds of functionality:

- Functionality that is identical for all components of a certain type.
- Functionality that is different for each component, but that can be generated from the model.
- Functionality that must by implemented manually by the developer.

The approach illustrated in Figure 8.15 is a proven technique for implementing this in object-oriented languages:

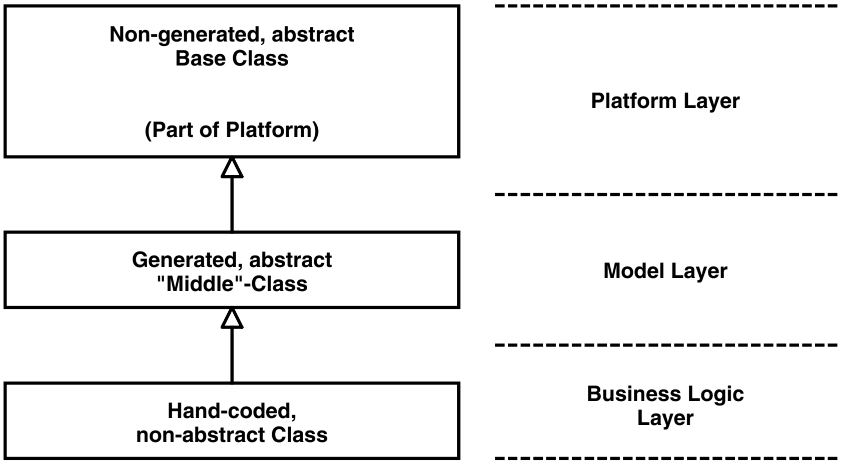


Figure 8.15 The three layers of a 3-tier implementation

An abstract superclass for all components of a certain type is implemented as part of the platform. For each component, the generator generates an abstract intermediate class that inherits from the superclass and realizes all aspects that can be deduced (and thus generated) from the model. Last but not least, the developer creates a non-abstract implementation class that inherits from the generated class. This class is later used by the system by being instantiated. This manually-implemented class ‘fills the holes’ in the generated intermediate class. This can be done quite elegantly with the aid of the Gang-of-Four patterns [GHJ+94]. The rest of this section, as well as Figure 8-16, explains how.

In Figure 8-16, case a), generated code calls non-generated code. This is almost trivial. In generated code, one will always access non-generated code – after all, one continues to use tried and test libraries. In this case, interpretation should be taken a bit further: one should always generate as little code as possible and, whenever it’s feasible, resort to using existing, tested code that has its place in the domain architecture in the shape of the platform.

Case b) is a bit less obvious. Here, manually-implemented code calls generated code. For this purpose, the manually-implemented code must ‘know’ the generated code, which can lead to unpleasant dependencies in the build process. Case c) can help here. Here the generated code can inherit from a manually-created class, or respectively implement a handwritten interface. The handwritten code can then be programmed against that interface. At runtime, the instance of a generated class will then be instantiated, for example with a factory operation.

Let’s return to the example with protected areas: one option for avoiding protected areas is to use inheritance, as shown in case d). Here, an implementation class inherits from the generated superclass. The implementation class overwrites the generated operations and in this way provides the behavior. A factory can again help with instantiation. Of course – as shown in case e) – generated code can inherit from a non-generated class and, if necessary, invoke its operations.

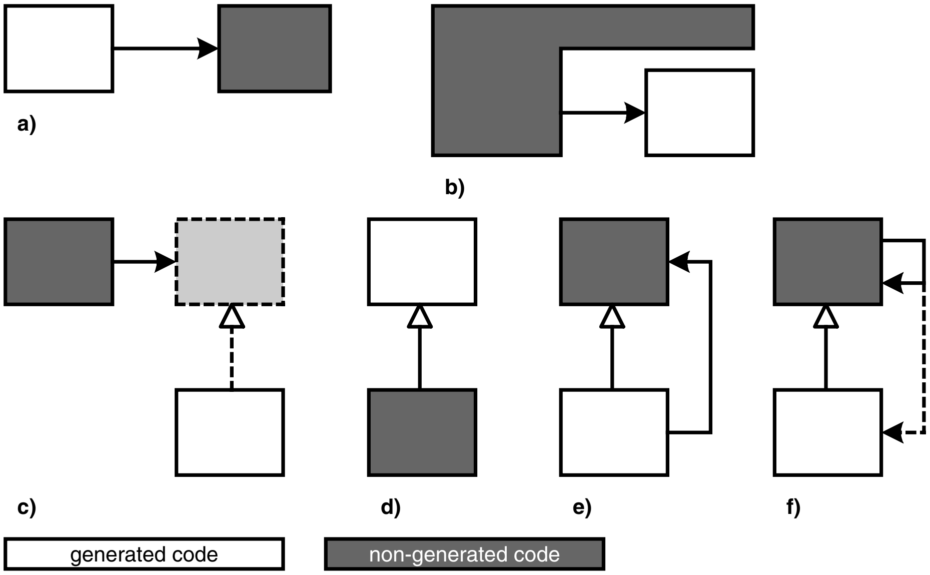


Figure 8-16 Pattern-based integration of generated and non-generated code

Case f) is also interesting, where a manually-implemented class or its operations call the operations of a generated subclass. This is basically a use of the Template Method pattern. The non-generated superclass defines a number of abstract operations that are overwritten by the generated class. Other operations of the superclass call the abstract operations. Once more, a factory helps with instantiation.

Another way of integrating handwritten code and generated code (or, more generally, code created through various tools) is to use partial classes – if your language supports them. For example, on the .NET platform you can split a class declaration into several files by marking the class definitions in each file with the additional keyword *partial* (*public partial class XYZ...*). When compiling the code, the compiler considers all the partial declarations of a class together. This allows you to generate one part of a class and manually implement the other.

The approach of integrating generated and non-generated code described in this section often requires generation that comprises several steps. Of course, the application developer can implement their application logic in the lowest layer only if the middle layer has been generated. On the other hand, in many cases an additional generation, compilation, and a further build step will be required that use the generated artifacts as well as the platform's artifacts, plus those that were manually implemented, to merge them into a complete application.

Thus, often two generator runs must be executed: the first reads specific model elements and generates a number of superclasses – that is, a kind of API – to which the developer then implements their application logic, which must be written manually.

The second run of the generator generates all model elements and uses them, as well as those written manually by the developer, to create the complete application. Depending on the platform, this step also encompasses processes such as compilation, linking, packing and so on. To enforce the developer's code to really inherit from the generated class, one can use dummy code (see the next section) or a recipe framework (Section 11.1.4).

If the right features of the targeted programming language are applied, this approach can also serve to enforce certain architectural constraints. Consider the diagram in Figure 8.17:

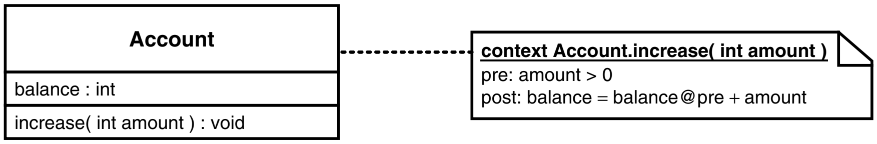

Account
balance : int
increase(int amount) : void
context Account.increase(int amount)
pre: amount > 0
post: balance = balance@pre + amount

Figure 8.17 A simple UML model with a constraint

In this case the goal is to anchor the pre- and post-condition defined in OCL in the model in the generated code, and in such a manner that it is not possible for the programmer to circumvent the check at runtime. Using protected areas, the following code could be generated:

```
// generated
class Account {
  int balance;
  public void increase( int amount ) {
    assert( amount > 0 ); // precondition
    int balance_atPre = balance; // postcondition
    // --- protected area begin ---

    // --- protected area end ---
    assert( balance = balance_atPre + bamount );
                  // postcond.
  }
}
```

Of course, it doesn't really work like this. The developer can delete the pre- and post-condition code at any time. The variant using simple inheritance is not practicable either. The developer would overwrite the operation *increase()* and thus avoid the checking of the constraints. Alternatively, one could define additional operations, *increase_pre()* and *increase_post()*, that check the pre- and post-condition. The implementation operation in the subclass would have to invoke this operation. However, in this approach the developer could forget to call these operations. There is a rather elegant solution to this problem that uses the Template Method pattern:

```
// generated
public abstract class Account {

  int balance;

  public final void increase( int amount ) {
    assert( amount > 0 ); // precondition
    int balance_atPre = balance; // postcondition
    increase_internal(amount)
    assert( balance = balance_atPre + amount );
                   // postcond.
  }

  protected abstract void increase_internal(int amount);
}
```

Here the operation *increase()* is completely generated. It contains pre- and post-condition code and internally invokes an operation *increase_internal()* for the execution of the actual behavior. This is defined as an abstract method, and the subclass to be written manually by the developer must implement the operation.

```
// manually written code
class AccountImpl extends Account {
  protected void increase_internal(int amount) {
    balance += amount;
  }
}
```

Since the external interface (the *public* method) is still *increase()*, a client of the class must always invoke this operation, and therefore cannot circumvent the checks, because *increase_internal()* is protected. A developer is also unable to overwrite the *increase()* operation in the subclass to circumvent the checks, because it is *final*. Thus the problem's solution is both bulletproof and elegant.

8.3.2 Dummy Code

One often has to coerce a developer to do certain things to obtain a consistent system. In the context of 3-tier implementation, it is for example necessary that the developer implements a class of their own that meets the following requirements:

- It must follow a particular naming convention (has the same name as the superclass, only with the suffix ‘Impl«).
- It must inherit from a certain (often generated) class and, if necessary, overwrite specific operations.
- It must perhaps implement a specific interface, or otherwise provide specific predefined operations.

Since we are dealing with manually-developed classes here, these things cannot be enforced by simply enforcing them through code generation templates. However, what can be done is to generate code that checks the required characteristics with the compiler's help. Let's assume that we have the following generated class:

```
public abstract SomeGeneratedBaseClass
       extends SomePlatformClass {
  protected abstract void someOperation();
  public void someOtherOp() {
    // stuff
    someOperation();
  }
}
```

Let's further assume that the developer will inherit from this class, overwrite the operation *someOperation()*, call the class *…Impl*, and implement an interface *IExampleInterface*. The correct implementation then looks as follows:

```
public abstract SomeGeneratedBaseClassImpl
       extends SomeGeneratedBaseClass
       implements IExampleInterface {
  protected void someOperation() {
    // do something
  }
  public void anOperationFromExampleInterface() {
    // stuff
  }
}
```

How can one enforce the developer to correctly implement this class, even if the framework (as regularly happens) only instantiates the class dynamically in the context of a factory, thus preventing the option of a compiler check?

The solution is to generate dummy code that only serves to verify the guidelines described here:

```
public abstract SomeGeneratedBaseClass
       extends SomePlatformClass {
  // as before
  static {
    if ( false ) {
      new SomeGeneratedBaseClassImpl();
        // verifies that the class is present,
        // and that it is not abstract
      SomeGeneratedBaseClass a =
         (SomeGeneratedBaseClass)
            new SomeGeneratedBaseClassImpl();
        // verifies that the implemented class is
        // actually a subclass of SomeGeneratedBaseClass
      IExampleInterface x = new
        SomeGeneratedBaseClassImpl();
        // verifies that it implements the
        // IExampleInterface
```

```
 new SomeGeneratedBaseClassImpl().xyzOperation();
        // this would verify that the operation
        // xyzOperation is implemented in the class
    }
  }
}
```

This mechanism ensures that the compiler issues corresponding error messages *as soon as the base class has been generated*. This forces the developer to implement the subclass correctly. Mistakes that might be hard to find later on are thus avoided. Note the *if (false)*, which ensures that the code is never executed at runtime[2].

Depending on the level of support you get from your tool chain, using a recipe framework (Section 11.1.4) is obviously a better approach than dummy code.

8.3.3 Technical Subdomains

Big systems typically encompass a variety of (technical) aspects. The description of all these aspects in a single model (and consequently with a single DSL) is complicated and impractical. The model runs the danger of being overburdened with details of different aspects and so drowning in its own complexity. Furthermore, in most cases a certain DSL (or rather its notation) is well-suited for the description of *one* specific aspect, but not for that of other aspects in the system.

Let's for example examine a UML-based DSL for the description of business processes. In addition, the persistence of model elements and the GUI design/layout must be described.

You must mark persistent model elements accordingly in the DSL, so that the required code and RDBMS schemas (SQL DDL) can be generated. It is difficult to accommodate all this information in a single model. A UML-based language is often not capable of covering all these aspects.

Theoretically, the modeling of a detailed GUI layout with UML is conceivable, but it is not practicable, especially because today's GUI design tools provide very good graphical, WYSIWYG-style DSLs and accompanying wizards. GUI design can only be further automated via real abstraction, for example if the platform strongly standardizes layout and design and makes many decisions for the developer. Extensive standardization makes sense in some application domains (business applications, administration interfaces), whereas in others too strong a standardization restricts the necessary leeway for design decisions (such as Web sites).

If you try to cover too many aspects in a single model, maintainability will be impaired, and efficient distribution of modeling tasks for various (partial) teams also becomes more difficult. To avoid such problems, introduce a structure of technical subdomains. Each subdomain should be modeled using a DSL suited for this purpose. Model the various subdomains with separate models, and bring these different models together in the generator. To enable this, define a small number of gateway metaclasses – that is, metamodel elements that are used in the metamodels of various DSLs. These serve to link the models of the different DSLs, as can be seen in Figure 8.18.

[2] Some IDEs are clever enough to notice that the code in the *if (false)* branch is never executed and thus complain about the unreachable code. In this case one has to formulate the condition a bit less obviously.

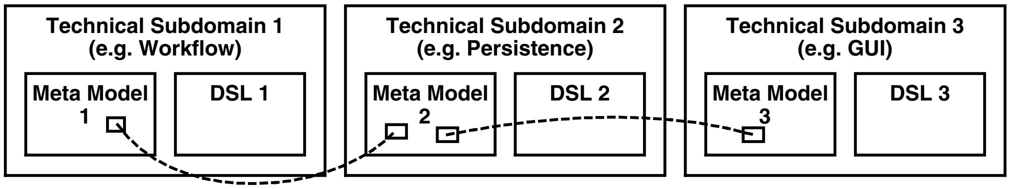


Figure 8.18 Linking of different technical subdomains via gateway-metaclasses

This approach is particularly useful if you ignore the concrete syntax in the transformer/generator (see Chapter 11), because then the gateway-metamodel elements can be rendered in different concrete syntaxes in the different subdomains, while the transformer/generator alone works with the abstract syntax. Thus a simple and natural integration of the subdomains is possible.

Note that the approach described here partitions the system into technical subdomains instead of structuring the system into functional subsystems. The latter is important and necessary too, however, but is independent of the structure of the subdomains described here.

Model-driven integration is a very special technical subdomain. Mapping and wrapping rules can be very elegantly defined in a customized DSL. Generator-based AOP is an alternative for handling cross-cutting subdomains.

8.3.4 Proxy Elements

In principle the integration of different subdomains via gateway-metaclasses works well, but it causes specific model elements to appear in a number of models, regardless of whether the models are models of different subdomains or shared elements such as example interfaces in various partitions (partial models). To avoid duplication of information, it is often advisable to work with proxys or references. Take a look at the example in Figure 8.19:

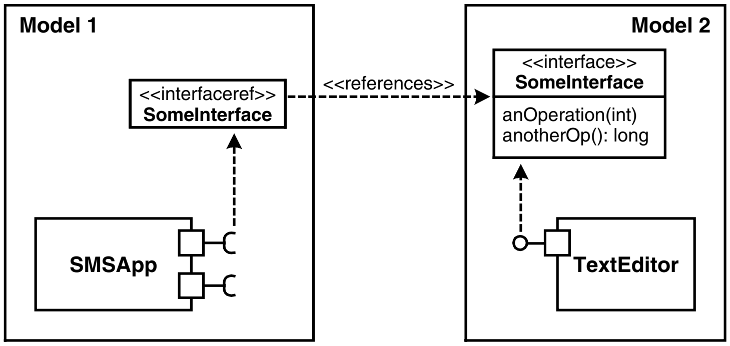


Figure 8.19 Examples for the use of proxy elements in the model

Here the interface reference in Model 1 refers to the interface of the same name in Model 2 – matching is based on identical names. Of course, the metamodel must be extended accordingly. You can see the respective excerpt in Figure 8.20.

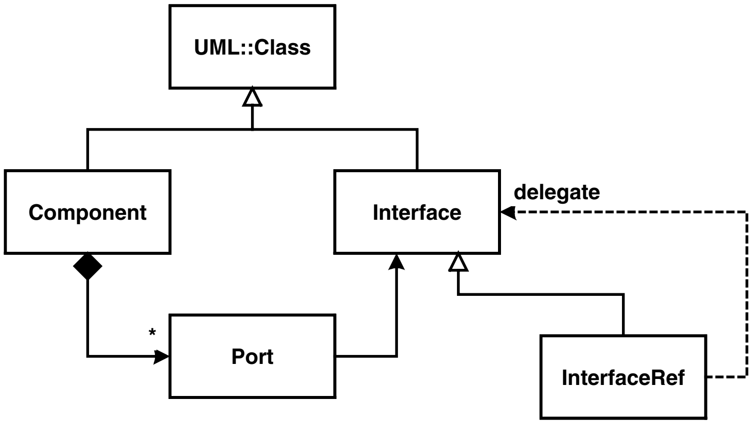


Figure 8.20 Metamodel with interface-proxy

A port implements an interface: an interface reference is a subclass of *Interface*, as in the Proxy pattern in [GHJ+94], and references the *delegate* – that is, the object for which the proxy stands. Note that this reference to the delegate is actually realized by an association in the context of the metamodel — that is, in the generator, once all partial models are loaded. In the models the delegate reference references the actual object via other properties: in this example, it uses the name.

This approach presupposes that the generator has Model 2 at its disposal when generating Model 1 in the example. The generator must dereference the reference to the actual object. Ideally, it replaces the port's association to the reference directly with an association to the actual object. Alternatively, the proxy can forward any operation invocations to the delegate. Due to the subtype relationship, this is no problem from a technical standpoint, via polymorphism.

Note that the application of this principle lets you merge any partial models (be they partitions or technical subdomains) independently of the abilities of the modeling tool or the concrete syntax. In practice, this is an extremely useful technique.

8.3.5 External Model Markings

To enable transformations of a source model into a target model (or to generate code), it is sometimes necessary to provide additional information at generation time that is tailored to the specific target metamodel. Adding this information to the source model would pollute it unnecessarily with target model concepts. The OMG suggests the use of model markings, but only very few tools support this concept sufficiently. Instead, we recommend that this information is described *outside* the model, for example in an XML file, and this external information provided to the generator/transformer at generation time.

If this is done correctly, no inconsistencies will emerge, because the generator can issue error messages when information in the external model markings are missing for a model element, or if information is given for nonexistent model elements. In principle, this procedure is a special case of technical subdomains.

8.3.6 Aspect Orientation and MDSD

This section requires knowledge of aspect-oriented programming (AOP). Introductory material can be found in [Lad04] and [AOSD].

Aspects in terms of AOP are cross-cutting concerns that traverse an application's code 'horizontally«. Typical examples are often of a technical nature[3], such as transactions, persistence, or logging. Aspect orientation has the goal of localizing such cross-cutting concerns in the context of a software system family in a *single* module, and thus making it more easily changeable or configurable. Cross-cutting concerns can be addressed at different levels in the MDSD context:

- The generator can read different models that represent different aspects of the complete system (technical subdomains), weaving them at the model level and generating code from them that addresses all the aspects.
- The generator per se intrinsically localizes specific cross-cutting concerns. Since a number of artifacts are generated using a *single* transformation rule or a *single* code generation template, a change in this one place in the template affects all the generated artifacts.
- Specific cross-cutting concerns can be addressed using architectural constructs. The classic examples of this approach are proxys and interceptors. In this case, the generator can create the required proxys automatically while the required interceptors can be installed at runtime after the necessary configuration has been defined.
- Finally, one can of course also integrate aspect-oriented programming by generating aspects – for example in the shape of AspectJ – or at least the *pointcuts*[4], which define where the aspect influences the base system. An example of this can be found at the end of Chapter 16.

Several concrete aspects can easily be realized with a generator:

- *Thread synchronization*. When the generator generates the implementation of a queue, it can automatically insert the synchronization code if the queue is used from several threads. (This must be stated in the model for the generator to know.)
- *Resource allocation*. The generator can generate the implementation of a factory that supports various resource allocation strategies (eager acquisition, lazy acquisition, and pooling – see [POSA3]).
- *Security.* The generator can create proxys for all components that carry out authorization checks. The instantiation of components can take place via a factory that inserts these proxys. It is important to make sure that the generator has introspective access to the components at generation time: It know their interfaces, methods, parameters, and so on, because these aspects are explicitly present in the model, or at least in the generator.

[3] There are also functional cross-cutting concerns, but for simplicity we ignore them here.

[4] A *pointcut* is a location in the execution of a program where an aspect can contribute additional behavior.

Another way of addressing cross-cutting concerns is the use of a suitable (technical) platform. The generator will then only generate the necessary configuration files for that platform.

- Some platforms allow only some specific predefined technical aspects to be addressed. EJB containers, for example, conduct security checks and manage transactions and resource allocation. In this case the generator creates the corresponding deployment descriptors.
- Other platforms only allow the configuration of specific aspects at predefined join points[5]. CORBA's portable interceptors are an example of this. Here, the generator would generate the code for instantiation of an suitably configured POA or ORB Core.

We are faced with the question of which of these options we should use in practice. There is no general answer. However, there are various factors that can influence a decision:

- At which granularity level must aspects be addressed? When corresponding hooks exist at the platform level, using these is certainly the easiest approach.
- If the join points influenced by the aspect are generated by the generator, then the generator can take care of adding the respective aspects at these join points. In this case, a cross-cutting concern in the generated code becomes – practically automatically – a 'module' in the generator. The Strategies pattern [GHJ+94] is definitely applicable here!
- When aspects influence join points in non-generated code and the platform framework provides no access to these join points, an aspect language such as AspectJ is certainly the best choice. In this case, the generator would only decide – based on the model – which aspect applies to which code sections. In consequence, the generator creates the configuration for the aspect weaver[6] and the pointcuts. An example of this can be found towards the end of the second case study in Chapter 16.

The general rule is to avoid additional tools if possible. A more detailed elaboration of how MDSD and AOSD/AOP relate to each other and how they can be used together can be found in [Voe04].

8.3.7 Descriptive Meta Objects

When a rich domain-specific platform is used for MDSD, the application often needs information about model elements at runtime to be able to control the platform dynamically. How can information from the generated application be made available at runtime and associated with generated artifacts? How can a bridge between generated code and framework parts be built?

Let's assume, for example, that you want to equip an application with a logging mechanism for domain issues. The application needs to log the values of attributes from instances of generated classes in a file. To this end, both the attributes' values and the attributes' names are needed. In languages that do not feature a reflection facility, the implementation of such functionality is non-trivial – unless you really want to endure the pain of implementing it manually for each class.

Another case is the annotation of object attributes in the model with useful additional information, such as multi-lingual descriptions for the use in GUIs, or regular expressions for

[5] A *joinpoint* is a specific location in the execution of a program where an aspect's advice can be woven in.

[6] The tool that combines (*weaves*) aspect code with the base program.

checking simple value-range restrictions for attribute values. At runtime, you must be able to access this information conveniently, for example to construct GUIs from it dynamically. Often you cannot simply embed this information in existing classes, either for performance reasons, to avoid dependencies, or due to other architectural restrictions.

To solve this problem:

- Use the information available in the model to generate meta objects that describe the generated artifacts at runtime.
- Provide a mechanism that lets the generated artifacts access their respective meta objects.
- Make sure that the meta objects have a generic interface that is accessible for the domain-specific platform.

Figure 8.21 illustrates this approach, which makes selected parts of the model accessible to the running application in an efficient manner. In theory, storage of parts of the model with the application would be an alternative, but access to model information typically would be slow and cumbersome, which is why in most cases this approach is useless in practice. An exception are small textual model fragments (for example value restrictions) that can be evaluated by an interpreter at runtime.

There are different ways of associating meta objects with the artifacts they describe. If the artifact is completely generated, often a *getMetaObject()* operation can be generated directly into the artifact.

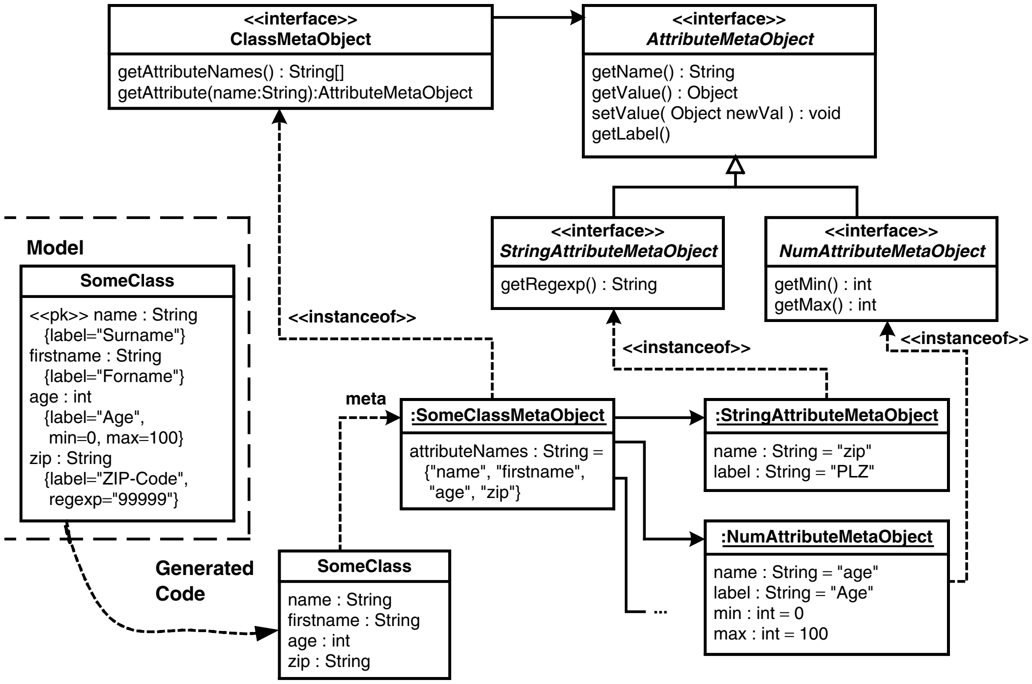


Figure 8.21 The principle of descriptive meta objects

If this is not feasible, a central registry can be used instead, which provides access to meta objects using a lookup function like *MetaRegistry.getMetaObjectFor(an Artifact)*. The implementation of this operation is generated, of course.

Meta objects can be used not only to describe program elements, but also for working with program elements, which results in a generated reflection layer, which we describe in Section 8.3.8.

In languages that support Annotations, such as .NET or Java 5, you can use this feature to achieve a similar effect. The following fragment of code illustrates how to attach the information to the source code. Reflection can be used at runtime to query the information.

```
public class SomeClass {

  @StringAttributeMeta( name=“name“, label=“Name“ )
  private String name;

  @StringAttributeMeta( name=“firstname“, label=“First Name“ )
  private String firstname;

  @IntAttributeMeta( name=“age“, label=“Age“, min=0, max=0 )
  private int age;

  @StringAttributeMeta( name=“zip“, label=“ZIP“ )
  private String zip;

  // getters and setters ...
}
```

8.3.8 Generated Reflection Layers

Meta object protocols such as those described in [KRB91] are a method for inspecting, modifying, or ‘re-interpreting’ programming language objects. This typically happens dynamically, for example in languages such as CLOS [Kos90]. In the context of MDSD, we can at least provide one *readonly* meta object protocol, so that classes can be introspected or operations invoked dynamically. This works independently of whether the underlying programming language supports reflection or other similar mechanisms: you can also implement this approach in C/C++! A generic interface allows clients to access all kinds of generated classes. To simplify matters, we illustrate this here with Java:

```
public interface RClass {
    // initializer - associates with base-level object
  public setObject( Object o );
    // retrieve information about the object
  public ROperation[] getOperations();
  public RAttribute[] getAttributes();
    // create new instance
  public Object newInstance();
}
```

```
public interface ROperation {
    // retrieve information about op
  public RParameter[] getParams();
  public String getReturnType();
    // invoke
  public Object invoke(Object[] params)
}
public interface RAttribute {
    // retrieve information about op
  public String getName();
  public String getType();
    // set / get
  public Object get();
  public void set( Object data );
}
```

The implementation of this interface for the respective classes is generated. During generation, you have access to all of the relevant information in the model. Since the interfaces are generic, platforms or other dynamic tools can work with this data using the reflective interface.

8.4 The Use of Interpreters

by Arno Haase

For the most part this book discusses MDSD with code generation in mind. This is arguably the most widely used way of doing MDSD, but interpreters are a different approach that shares the same underlying principles and fits the definition of 'automatically transforming formal models into executable code', albeit in a different fashion.

Interpreters and generators are functionally equivalent. Every model can serve as input for either, at least in principle. It is however more common to use generators for structural aspects of a system and interpreters for behavior. The rationale behind this is that structural aspects are – well, structural. Many of the established mainstream libraries that handle structural aspects such as persistence, remoting, or any kind of component interaction or integration, require data structures to be present as explicit source code, and therefore code needs to be generated to use them as part of the platform. This is mostly a matter of taste, and it is easy to conceive frameworks for structural aspects based on generic data structures that would allow the use of interpreters. Widely-used libraries, however, currently follow a different path, one of the main reasons being a traditional focus on performance.

Behavioral aspects, on the other hand, have traditionally been perceived as less critical to the overall performance of a system[7]. In addition, the abstractions of behavioral models are often large compared to the amount of code that glues them together (for example steps in a workflow), further reducing the performance impact of an interpreter. Interpreters for expressions are more convenient to build and test than generators. In addition to that, changing the model for an

[7] Obviously there are exceptions to this. But even in the domain of numerical simulations, it is common for a system to read and interpret a model of the problem to be solved. Good performance is achieved by using big building blocks, such as entire solvers for differential equations, as part of the platform.

interpreter requires nothing more than perhaps a restart of the system, adding to the convenience of this approach. Behavioral aspects of systems are therefore often implemented using an interpreter rather than a generator.

The following section takes a closer look at interpreters in the context of MDSD.

8.4.1 Interpreters

An interpreter is a piece of software that reads and evaluates a model at *runtime*, performing whatever operation is specified in the model. Just as does a generator, an interpreter works on formal models with precisely-defined semantics, and both techniques result in the execution of the operations specified in the model. The term 'execution of operations' is used in a very loose fashion: it is meant to include code that a compiler generates to handle a generated structural definition.

Both approaches require a parser for the model, that is, a piece of software that transforms a model from its concrete syntax into an abstract syntax, typically an object graph in memory. This parser is identical for both approaches in the degree to which it can be shared by a generator and an interpreter for the same models.

An interpreter differs from a generator (or *compiler*, using programming language lingo) in two respects: the point in time at which the model is analyzed, and the way the operations are executed. Let's look at these two differences in more detail:

- *Analysis time*. For the system to be built, a generator typically analyzes the model at build time, whereas an interpretative approach makes it possible to read and analyze the model at runtime, allowing late binding and changes to the running system.
- *Mode of execution*. It is understood that an interpreter consecutively looks at chunks of the model, distinguishing between the different kinds of model elements and executing different pieces of code based on what it sees. This is different than a generative approach, in which the generator looks at chunks of the model and combines different sections of code into source files that are then compiled. Interpreters use one more level of indirection at execution time.

These distinctions are however less strict than they might appear (or, for the nostalgic among us, than they used to be). In some domains it has for example become common practice to generate Java byte code at execution time, giving a generator the benefits of very late binding. There is also a growing trend for interpreters to come with a preprocessing component that checks and validates a model at build time, ensuring that some types of errors are found at build time rather than during execution.

An increasing number of interpreters go a step further by following a two-step approach. The original models are read and validated at build time, and are then transformed into an optimized intermediate format usually referred to as *byte code* [8]. The actual interpreter (or *virtual machine*) then reads and interprets only the byte code. The advantage of using such an intermediate format

[8] This byte code can make use of an existing virtual machine such as the JVM, but it is perfectly possible to define a specific kind of byte code that fits the domain in question particularly well. One example of this is the Eclipse GMT ATL model-to-model transformation engine.

is the same as that of transforming models in several steps: it simplifies each step by introducing a well-chosen intermediate abstraction.

Another particularly common practice that combines the two approaches is to generate models that are interpreted at execution time. This is very commonly done for configuration files, and often for platform libraries such as a persistence framework.

So while it is useful to distinguish between interpreters and generators, it is good to keep in mind that the decision of one or the other is not a black and white question. Instead, it is possible to combine the approaches, or move gradually from one to the other as the need for specific non-functional properties arises.

Another approach to this duality has been adopted by many rule engines. It is common for these to offer both an interpreter and a generator for the same model, leaving the choice between them, with its non-functional implications, to the developer using the engine.

8.4.2 MDSD Terminology Revisited

Interpreters and generators are both technologies for implementing MDSD, and therefore obviously share the core abstractions of MDSD. But interpreters come from a different background of programming culture and involve different trade-offs than do generators, so it is beneficial to take another look at the key concepts of MDSD from an interpretative angle.

Domain

We defined a domain as a bounded field of interest of knowledge, intentionally making the term widely applicable. This definition clearly makes no assumption about whether an interpreter or a generator is used for a given domain.

Some domains however clearly lend themselves more to a generative approach – for structural aspects in particular – whereas it is easier to envision others based on interpreters. Therefore it is helpful during domain analysis to be consciously open-minded with regard to a later implementation technology, postponing the decision of whether to build a generator or an interpreter for a given domain, or code it manually instead, until a solid understanding of the domain has been reached.

The interpreter–generator decision also influences how we draw the boundaries of the domains. The scope of a generator and an interpreter for a given domain are typically different to some degree: it is for example more natural to include constraint checks in an entity metamodel if the models are to be interpreted than if they serve as input for a generator. It is therefore good practice to vary the domain boundaries a little to find the approach that best fits a given system context.

Metamodel

Any kind of metamodel can serve as the basis for either a generator or an interpreter. It is however more common to use generators for structural metamodels and interpreters for those that describe behavior.

Meta Meta Model

The parser is identical for interpreters and generators, and therefore the meta meta model is the same for both – at least in principle. It is for example perfectly possible to define the abstract syntax for an interpreted model in terms of MOF, even if it is a behavioral metamodel organized around the key abstraction of 'expression'. It is however far less common to do so for behavioral models than it is for structural models.

Formal Model

Any model with any concrete syntax can serve as input for both a generator and an interpreter. But behaviorally rich textual models are a domain in which interpreters particularly can show their strengths, because they make it easy to use primitive building blocks with slightly different semantics from those of the underlying programming language. As these textual models look a lot like well-known 3GL source code, they are also often referred to as 'source code'.

Platform

Both generators and interpreters are based on platform code – the code that already exists on the target platform – and it is usually a good idea to use as much platform code as possible. The interaction with the platform however looks different for interpreters than it does for generators.

Firstly, platform libraries and frameworks typically demand exactly one kind of interaction, and whatever MDSD approach is used must cater to their needs. If for instance a persistence framework requires business objects to be present in struct-like classes, then there may be no way for an interpreter to provide them directly: they need to be generated. API calls, on the other hand, can be performed both from generated code and from an interpreter.

Secondly and on a more conceptual note, interpreters themselves can be viewed as part of the platform. The interpreters are written in the underlying programming language and can be viewed as providing specific services. But even if we do not take quite so radical a view, the distinction between the interpreter and its supporting framework and library code is more blurred than it is for generators, where a given piece of code is either generated or not.

8.4.3 Non-Functional Properties of Interpreters

So when should one use a generative approach, and when is an interpreter preferable? Most of the strengths listed in Section 9.1 for generators hold for interpreters as well. That said, the following list discusses the different trade-offs of the two approaches.

Performance

This is the number one concern whenever interpreters are involved. Interpreters are inherently slower than generated code, mainly because the compiler of the underlying programming language can perform fewer optimizations and there are usually more indirections.

Obviously, there are domains in which performance is of paramount importance, such as embedded real-time systems, which rules out an interpretative approach. For other systems however a performance overhead of 30 – 50% for a specific part of the operations might not even be noticeable, because I/O operations take up the bulk of the time.

So while the performance impact sometimes rules out the use of an interpreter, it by no means renders interpreters universally useless. The performance difference must be measured for every system individually, and often optimizations inside the interpreter can provide sufficient performance.

Code Volume

Unless great care is taken, generators create more code than the original model contained, giving them a bigger footprint than interpreters. On the other hand, that is not an issue for most systems. With embedded systems, where memory *is* an issue, analyzability and predictability are good reasons to prefer generators over interpreters.

Binding Time

Interpreters allow late binding at runtime, whereas generated code is usually bound at compile time. This is the single biggest advantage of interpreters compared to generators, opening the door to a number of beneficial practices.

Firstly, it becomes possible to change a model at runtime and directly affect the behavior of the system without the need for a a redeployment or rebuild, or even a restart. While not terribly useful in a production setting, it makes for very short debugging cycles, making it possible to hunt down and fix bugs while sitting in front of a machine with a requirements engineer.

Secondly, it makes it very simple to run several versions of the business logic in parallel. This is useful mainly in two situations: in mandator-based systems (that is, systems that consist of separate logical ‘partitions’ for separate groups of users – the mandators), every mandator potentially has a slightly different version of the logic. The other situation occurs in cases in which you need to run older versions of the business logic with older data, while newer data must be processed using new business logic. In such systems, every mandator or data set can have a reference to the business logic model that is applicable to it, allowing all data to be processed using the same interpreter.

8.4.4 Integrating Interpreters into a System

To be useful, interpreters must be integrated with the rest of a system. There are three points at which an interpreter touches the rest of the world, namely the interface through which it is called, extension points at which a system can make specific functionality callable from a model, and the mechanism through which the models are provided to the interpreter. We will look at these three points in turn.

Calling the Interpreter

Interpreters are pieces of code written in the platform's programming language, and can therefore be called just like any other piece of code. An interpreter typically provides a simple interface with basically just one method, taking objects passed as parameters to the interpreted model – plus optionally an identifier describing which model or part of the model should be executed – and returning the results of the execution of the model. This simple call interface lends itself naturally to encapsulation in a component.

Like any other component, an interpreter can operate on other components or resources, such as files or a database. In such cases the resources must be made known to the interpreter, just as they would have to be made known to any other component, for example by using dependency injection, parameter passing, a look-up in a centralized registry, or even global variables.

Extension Points

It is often not feasible to define a comprehensive metamodel that handles all facets of a domain. To do that would often introduce significant additional complexity, and simplicity is one of the reasons we do MDSD at all. Therefore it is a best practice to define extension points in the metamodel, allowing a model to reference code that is written in the underlying programming language. A good example of this would be the algorithm for creating the value of a primary key in a persistence language: common strategies should be explicit in the metamodel, but there should also be a way to handle the rare cases in which an exotic strategy is needed.

The mechanics of how such an extension is done best depend on the programming language: for Java, for example, the usual way is to provide a fully-qualified class name in the model and use reflection.

Providing the Models

An interpreter needs a model to execute at runtime, so the application must provide it. Models for interpreters are often text documents, which opens a wide range of possibilities for their storage and provision.

The simplest approach is to deploy them as part of the system. How that can be done depends on the underlying language – in Java they could for example be packaged inside one of the JAR files that make up the system. This approach effectively removes much of the benefit of late binding, but on the other hand it makes for a simple deployment model.

An alternative is to store the models as external resources and have the system retrieve them at runtime. This could be achieved using files, possibly on a shared file server for clustered servers, but it is also perfectly possible to store them in a database, on FTP servers, or anywhere the system can access. This makes them first-class resources, increasing flexibility, but adding complexity to the deployment and system management.

8.4.5 Interpreters and Testing

Interpreters need to be tested just like any other piece of software. And as with any kind of MDSD, testing comes in two flavors.

Testing the Interpreter

Interpreters tend to be easier to test than generators because they do not require code to be generated. In fact, interpreters can be tested with plain vanilla unit tests in a straightforward fashion, especially if they process textual models.

The most universal approach is to use black-box tests that provide the interpreter as a whole with a model, then check the results. Many interpreters are naturally modular, however, which allows tests at a finer granularity. This can for example take the form of providing just an expression as input to the parser and feeding the resulting abstract model to the expression part of the execution part of the interpreter.

It is possible in principle to test at an even finer granularity by passing manually-assembled abstract syntax trees to the execution engine of the interpreter. Bypassing the parser in this fashion rarely yields any additional benefit for textual models, however, because it is usually straightforward to find a piece of concrete syntax that yields a given fragment of abstract model. It can however be useful for interpreters that are based on a very complex concrete syntax.

Testing the Models

The models can be tested using normal unit tests, calling the interpreter through its official interface and checking the results or the side effects.

An added benefit of interpreters however is that models can be changed without restarting the system, making the system expose the modified behavior immediately. That makes debugging sessions in association with requirements engineer possible in which both parties iteratively modify the model until the system behaves as desired.

9 Code Generation Techniques

In keeping with the structure introduced in Chapter 1, we now want to address proven techniques that are the foundation for the selection or construction of MDSD tools – that is, those aspects that can be factored from domain architectures because they are of a more general nature. Yet a domain architecture cannot work without them: code generation techniques are an important foundation.

9.1 Code Generation – Why?

We have repeatedly mentioned that there is a close connection in MDSD between modeling, code generation and framework development – for example, framework completion code for the MDSD platform can be generated from a DSL. A rich, domain-specific platform, as described in Chapter 7, simplifies code generation and lessens the need for it. On the other hand, code generation offers advantages over purely generic approaches, or at least supplements them.

9.1.1 Performance

In many cases, code generation is used because one wishes to achieve a specific level of performance while maintaining a degree of flexibility. Traditional object-oriented techniques such as frameworks, polymorphism, or reflection are not always sufficient regarding their achievable performance. Using code generation, the configuration is stated abstractly – which also where its flexibility lies – and efficient code is generated.

9.1.2 Code Volume

Another reason for code generation is code size. If you know at compile time which features will be needed at runtime, the generator only needs to place those parts in the code. This can help to make the image smaller. Vice versa, the excessive expansion of constructs at the source code level can significantly enlarge the image. One example of this is C++ template instantiations.

9.1.3 Analyzability

Complex, generic frameworks or runtime systems tend to relocate programming language-related complexity to a proprietary configuration level. They usually make heavy use of interpretation, which hampers the possibilities for static analysis of program properties, and occasionally impairs error detection. In contrast, generated programming language source code possesses the analyzability of manually-programmed code. As we have pointed out, a sensible balancing of both approaches is ideal.

9.1.4 Early Error Detection

Flexible systems often use weak typing to allow decision-making at runtime (*Object* in Java, *void** in C/C++). Thus error detection is deferred to program runtime, which is often undesirable, and which is one of the reasons why this kind of programming is not popular in embedded systems. Some of these disadvantages can be cured through the use of 'static frameworks«. Configurations can be recognized as being flawed before compilation, and the compiler, too, usually has more information, so that it can too can report error messages.

9.1.5 Platform Compatibility

The classical case of using code generation in the context of MDA is that application logic can usually be programmed independently of the implementation platform. This enables an easier transition to newer and potentially better platforms.

9.1.6 Restrictions of the (Programming) Language

Most programming languages possess inconvenient restrictions in their expressiveness, which can be circumvented using code generation. Examples are type genericity in Java (at least before version 1.5), or the downcast to a variable class. Another example is the introduction of object-oriented concepts into a non-object-oriented language.

9.1.7 Aspects

Cross-cutting properties of a system such as logging or persistence can typically be implemented locally – that is, not scattered through the application – via code generation. We will further elaborate on this issue in the course of this chapter.

9.1.8 Introspection

The issue of introspection should not go unmentioned. Introspection describes a program's (read-only) access to itself. This allows the program to obtain information about itself, for example about classes, their attributes and operations. In certain programming languages, such as

Java, this mechanism is supported dynamically. Other languages such as C++ do not offer introspection. In such cases, code generation can create a substitute statically: instead of analyzing the program structure at runtime, the structure is analyzed at generation time – that is, before runtime – and code is generated that provides access to the respective structures.

9.2 Categorization

This section looks at various questions when dealing with metaprograms – programs that generate other programs. This includes the mixing/separation of the metaprogram and the *base program* (the generated program), as well as ways of integrating generated and non-generated code so that the metaprogram and the program are not mixed.

9.2.1 Metaprogramming

Code generators are metaprograms that process specifications (or models) as input parameters, and which generate source code as output.

Metaprograms can be run at different times in relation to the generated program:

- Completely independently of the base program – that is, before it.
- During compilation of the base program.
- While the base program runs.

Typical MDA/MDSD generators adhere to the first approach. Here, the metaprogram and the part of the base program to be created manually are usually specified separately. The generated code is also separated from the manually-created code, and both must be integrated by the developer (see Section 8.3.1).

Systems such as the C++ preprocessor or the C++ template mechanism can also be used for metaprogramming. Here, base program and metaprogram are mixed, and similarly the result of the generation process already contains manually-created as well as generated code, so is also mixed. However, the created program no longer knows anything about the metaprogram. We refer to that as *static metaprogramming*.

Lisp and CLOS [Kos90] allow the execution of metaprograms at runtime via a *meta object protocol*. This works because in Lisp programs are represented as data (lists). Metaprograms can modify these lists and thus create or modify base programs at runtime. Changes of the metaprogram made from the base program enable the modification of the base program's semantics.

9.2.2 Separation/Mixing of Program and Metaprogram

In the case in which metaprograms and base programs are mixed, a common (or at least integrated) language exists for programming and metaprogramming, and the source code components are not separated, but mixed. This can lead to invocation relationships between the two levels, in principle in both directions. C++ template metaprogramming can fall into this category, as well as Lisp and CLOS.

Mixing programs and metaprograms is a very powerful approach. However, the resulting system can easily become extremely complex, so its relevance to mainstream software development is very limited. A further implication of the mixed approach is that the target language is no longer a parameter of the code-generation process.

If program and metaprogram are separated, the system's creation takes place in two distinct phases. The metaprogram is run and creates the base program (or parts of it) as output, then terminates. The program does not know that the metaprogram exists[1]. The separation is maintained throughout the (meta-)programming process, including the build process.

The question of whether program and metaprogram are written in the same language is irrelevant here. For example, one can easily write a metaprogram in Java that generates Java, C++, or C# as output.

The approach in which metaprogram and base program are separated does support meta object protocols, but due to its lower complexity it is better suited for typical architecture-centric, Model-Driven Software Development. The generator used in our book, openArchitectureWare [OAW], is a hybrid with respect to the metaprogramming language: parts of it are implemented in Java (metamodel of the application family), and parts in the template language *Xpand*. Thus one can combine the advantages of an expressive template language with the power of a 'real' programming language without overloading the templates with too much metaprogramming logic.

9.2.3 Implicit or Explicit Integration of Generated with Non-generated Code

Implicit integration of both program types results in code that already constitutes a mix of generated and non-generated code. As a result, one no longer has to worry about the integration of the two categories.

In the other case, the generated code is initially independent of handwritten code sections. The two kinds of code must be integrated in an appropriate way: this was described in the previous chapter.

9.2.4 Relationships

In general we can say that a relationship exists between the two aspects of generator categorization:

- Generators in which program and metaprogram are separated usually also create generated code that is separated from the manually-created code, and which must therefore be integrated manually.
- Generators with a mix of program and metaprogram do not require this manual integration – the generator already creates the combined system.

Separation of program and metaprogram, as well as the explicit integration of generated code and manually-created parts, is recommendable in practice. We will introduce the corresponding techniques in detail later on, but first we give a few examples for the second category.

[1] The program can partially learn about the metaprogram's existence through descriptive meta objects.

9.2.5 Examples of the Blending of Program and Metaprogram

The C++ preprocessor is a system that blends program and metaprogram. The languages applied here are independent of each other: one can also use the C++ preprocessor with other programming languages, as it works purely textually on the source code. Since the system is based on macro expansion, the preprocessor already produces source code that integrates both generated and manually-created code. The following is a simple C++ macro:

```
#define MAX(x,y) (x<y ? y : x)
```

If this macro is used in the source code, it is expanded by the preprocessor according to the rule cited above:

```
int a,b;
int greaterValue = MAX(a,b);
```

becomes the following expression:

```
int greaterValue = (a<b ? a : b);
```

The generated code has been directly inserted where the macro to be expanded was originally located.

Another example, again from C++, are templates. Templates are a way first and foremost to implement type genericity in C++. Functions or classes can be parameterized with types. Due to the fact that this feature is realized in C++ using static code generation, this method can also be used for *template metaprogramming* [Ale01]. Here, generation takes place through the evaluation of templates[2]. This approach is primarily used for performance optimization, optimization of code volume, static program optimization, and in certain special cases, for the adaptation of interfaces, for generative programming or other interesting purposes described in [Ale01].

Here too program and metaprogram are blended. Integration is clearly closer than in the case of the preprocessor, because the template mechanism knows and uses C++'s type system. Similarly, the resulting system consists of already-generated and manually-created code.

The adaptation of code volume can serve as an example of template metaprogramming here. Specifically, we want to use the smallest possible data type (*short int)* adequate for the maximum value of a variable defined here so that we can decrease the memory footprint at runtime. The following code fragment illustrates this:

```
#define MAXVALUE 200
IF<(MAXVALUE<255), short, int>::RET i; // type of i is short
```

The IF statement in the example above is evaluated at compile time. It is implemented using templates, clearly visible from the use of the '<' and '>' brackets. Specifically, we apply *partial*

[2] Because this step takes place in the course of the C++ compilation process, source code is not the inevitable result, but rather direct machine code or corresponding intermediate representations in the compiler.

template specialization. We first define a template that has three parameters: a Boolean value, as well as classes (types) for the *true* case and the *false* case.

```
template<bool condition, class Then, class Else>
struct IF {
  typedef Then RET;
};
```

In the context of the C++ template definition, we now define a new type using *typedef*. Based on a commonly-accepted convention, this type is called *RET.* It serves as the template evaluation's return value. In the default case, we define the return type – the value of the template instance – as the type that has been defined as the *true* case above, that is, the template parameter of the name *Then*.

Next, the template is partially specialized for the case in which the Boolean expression is *false*. Now the template has only two parameters, since the Boolean expression has been set to *false*.

```
//specialization for condition==false
template<class Then, class Else>
struct IF<false, Then, Else> {
  typedef Else RET;
};
```

The return value *RET* is now the type parameter that was specified for the *false* case (the parameter *Else*). If the compiler finds an instance of this template, like in the *short/int* example given above, it will use the template that is the most specific – the *false* case in this context. *RET* is then defined with the type *Else* and is thus *short*.

9.3 Generation Techniques

In this section we introduce proven code generation techniques, including some code examples. We have organized the generation techniques into various categories:

- Templates + filtering
- Template + metamodel
- Frame processors
- API-based generators
- In-line generation
- Code attributes
- Code weaving

We briefly introduce all these techniques and illustrate them with examples. Except for in-line generation (and to some degree, code attributes), all these approaches require explicit integration.

Independent of the – in some cases important – differences between the various generation techniques, they all have the following in common:

- A metamodel or respectively an abstract syntax always exists, at least implicitly.
- Transformations that build on the metamodel always exist.

- Some kind of front-end that reads the model (the specification) and makes it available to the transformations exists.

9.3.1 Templates and Filtering

This generation technique describes the simplest case of code generation. As shown in Figure 9.1, we use templates to iterate over the relevant parts of a textually-represented model, for example using XSLT via XML.

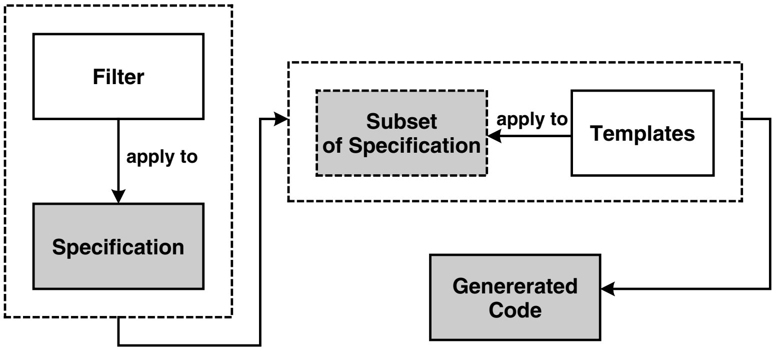


Figure 9.1 Templates and filtering

The code to be generated is found in the templates. Variables in the templates can be bound to values from the model. Below, we will present a simple example in which a Java Bean *Person* is generated from an XML specification (for simplification purposes, we do not use XMI as the model representation, instead we use a custom schema). This is the specification:

```
<class name="Person" package="com.mycompany">
  <attribute name="name" type="String"/>
  <attribute name="age" type="int"/>
</class>
```

The generated code should look like the following:

```
package com.mycompany;
public class Person {
  private String name;
  private int age;
  public String get_name() {return name;}
  public void set_name( String name) {this.name = name;}
```

```
 public int get_age() {return age;}
  public void set_age( int age ) {this.age = age;}
}
```

The XSLT stylesheet that performs this transformation looks roughly like the following:

```
<xsl:template match="/class">
  package <xsl:value-of select="@package"/>;
  public class <xsl:value-of select="@name"/> {
    <xsl:apply-templates select="attribute"/>
  }
</xsl:template>

<xsl:template match="attribute">

  private <xsl:value-of select="@type"/>
    <xsl:value-of select="@name"/>;

  public <xsl:value-of select="@type"/>
    get_<xsl:value-of select="@name"/>() {
       return <xsl:value-of select="@name"/>;
  }

  public void set_<xsl:value-of select="@name"/> (
    <xsl:value-of select="@type"/>
     <xsl:value-of select="@name"/>) {
       this.<xsl:value-of select="@name"/> =
         <xsl:value-of select="@name"/>;
  }
</xsl:template>
```

The generation using templates and filtering is fairly straightforward and portable, but the stylesheets soon become very complex. For this reason, this approach is totally unsuitable for larger systems, particularly if the specification is based on XMI.

The XMI problem can be somewhat alleviated if one works in several steps: an initial transformation transforms the XMI into a concrete, domain-specific XML schema. Further transformation steps can now generate code based on this schema. One gains a certain decoupling of the templates from the concrete XMI syntax, and the actual code generation – the second step – becomes much easier. However, one still works on the abstraction level of the XML metamodel – a problem that can clearly be solved using the approach presented next.

9.3.2 Templates and Metamodel

To avoid the problems of direct code generation from (XML) models, one can implement a multi-level generator that first parses the XML, then instantiates a metamodel (which is adaptable by the user), and finally uses it together with the templates for generation. Figure 9.2 demonstrates the principle.

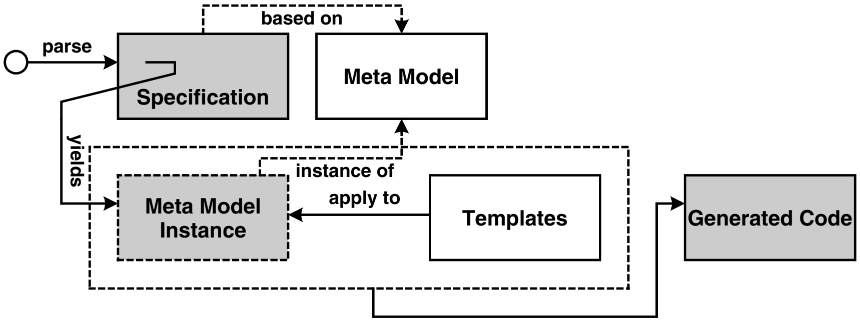


Figure 9.2 Templates and metamodel

The advantage of this approach is that on the one hand one gains greater independence from the model's concrete syntax, for example UML and its different XMI versions. On the other hand, one can integrate more powerful logic for the verification of the model – constraints – into the metamodel. In contrast to the templates, this can be implemented in a real programming language, such as Java. This kind of code generation is of special significance in the context of MDSD, as we pointed out in Chapter 6.

One interesting implementation aspect of the openArchitectureWare generator that falls in this tool category should not go unmentioned: from the perspective of compiler construction, metamodel implementation (for example in Java) *and* templates are part of the transformation. The metamodel assumes the role of the abstract syntax. Since the abstract syntax and the transformation are parameters of the compiler, we are in fact dealing with an open compiler framework. What is remarkable is that the constructs of the syntax – that is, the metamodel elements – compile themselves. Put another way, the compiler is object-oriented, which helps to avoid, among other things, tedious type switches. The templates are – from a conceptual point of view – compiler methods of the metamodel, just like the help methods implemented in Java. You can see this from the template definitions («*DEFINE method FOR metaclass*»). Just like Java, the template language also supports polymorphism and overwriting – both are necessary to build an object-oriented compiler – and only the definition of classes is delegated to the Java part. This is why we consider the template language to be object-oriented, even though no classes can be defined directly in the template language.

9.3.3 Frame Processors

Frames, the central element of frame processors, are basically specifications of code that should be generated. Like classes in object-oriented languages, frames can be instantiated, multiple times. During this instantiation, the variables (which are called *slots*) are bound to concrete values. Each instance can possess its own values for the slots, just like classes. In a subsequent step, the frame instances can be generated, so that the actual source code is generated.

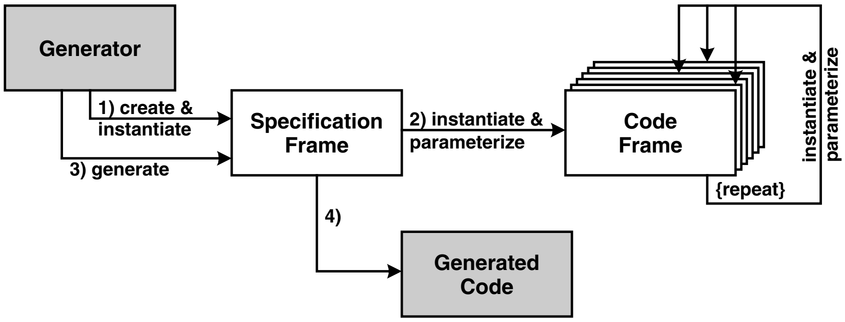


Figure 9.3 Frame processors

The values that are assigned to slots can be everything from strings to other frame instances. At runtime, this results in a tree structure of frame instances that finally represents the structure of the program to be generated. Figure 9.4 shows an example.

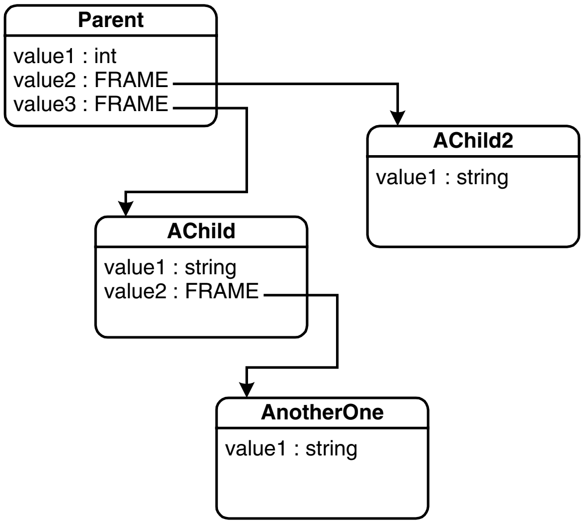


Figure 9.4 An example frame hierarchy

The following example uses the ANGIE processor [DSTG]. First, we show the generation of a simple member declaration that looks like the following:

```
short int aShortNumber = 100;
```

This code fragment already contains quite a number of variable aspects: the variable's name, its type, as well as an optional initialization parameter. The following frame generates this piece of code:

```
.Frame GenNumberElement(Name, MaxValue)
  .Dim vIntQual = (MaxValue > 32767) ? "long" : "short"
  .Dim sNumbersInitVal
  <!vIntQual!> int <!Name!> <? = <!sNumbersInitVal!>?>;
.End Frame
```

The first code line declares the frame, which is basically a constructor with two parameters: the name of the *NumberElement* and the maximum value. Based on this maximum value, the second line decides whether we need a *short int*, a *long int*, or simply an ordinary *int*. Line four defines the *host code* that is eventually generated in the course of code generation. The *<!...!>* syntax accesses the value of a slot of the frame instance. The code between *<?...?>* is only generated if the value of the slot contained in it is not undefined. The following piece of code instantiates the frame:

```
.myNumbElm = CreateFrame("GenNumberElement","aShortNumber",100)
```

It should be noted that this instantiation does not yet generate any code – only a frame instance is created and assigned to the variable *.myNumbElm*. The instance is kept in the generator-internal instance repository. If one finally executes:

```
.Export(myNumbElm)
```

the instance is 'executed' and the code generated. Instead of exporting the instance directly (and thus immediately generating the code), it can also be assigned to another frame instance's slot as a value, to construct more complex structures. The next frame, which generates a simple Java class, serves as an example:

```
.Frame ClassGenerator(fvClassName)
  .Dim fvMembers = CreateCollection()
  public class <!fvClassName!> {
    <!fvMembers!>
  }
.End Frame
```

This frame accepts the names of the class to be generated as a parameter. Moreover, a multi-value slot (a *collection*) is created. An external script (or another frame) can now set values, such as other frame instances. For example, the *NumberElements* from above can be set.

```
.Static myGeneratedClass As ClassGenerator
.Function Main(className)
  .myGeneratedClass =
        CreateFrame("ClassGenerator",className)
     .Add(myGeneratedClass.fvMembers,
        CreateFrame("GenNumberElement","i", 1000))
     .Add(myGeneratedClass.fvMembers,
        CreateFrame("GenNumberElement","j", 1000000))
.End Function
```

During the export of *myGeneratedClass*, a simple Java class is generated that contains the two members *i* and *j*.

9.3.4 API-based Generators

Probably the most popular type of code generators are the API-based ones. These simply provide an API with which the elements of the target platform or language can be generated. Conceptually, these generators are based on the abstract syntax (the metamodel) of the target language, and are therefore always specific to one language, or more precisely to the target language's abstract syntax.

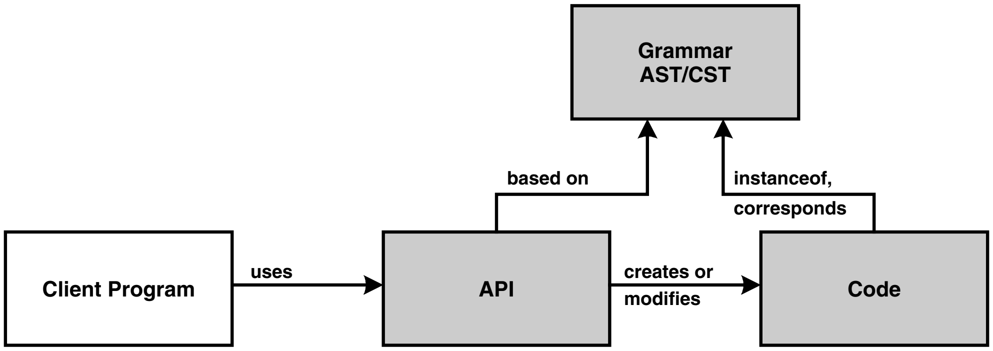


Figure 9.5 Functional principle of API-based generators

For a change, the following example is taken from the .NET world. The following code should be generated:

```
public class Vehicle : object {
}
```

The following fragment of C# code generates it:

```
CodeNamespace n = ...
CodeTypeDeclaration c = new CodeTypeDeclaration ("Vehicle");
c.IsClass = true;
c.BaseTypes.Add (typeof (System.Object) );
c.TypeAttributes  = TypeAttributes.Public;
n.Types.Add(c);
```

The code shown above builds an internal representation of the code, typically in the shape of an abstract syntax tree (AST). A call to a helper function initiates the actual code generation, which is not shown here.

This kind of generator is fairly intuitive and easy to use. Furthermore, it is also easy to enforce that only syntactically correct code can be generated, guaranteed by the compiler of the generator code in combination with the API. However, the problem with this kind of generator is that the potentially large amounts of constant code – the code that does not depend on the model – must be programmed instead of simply being copied into the templates.

The use of such generators becomes efficient when you build domain-specific generator classes by using well-known OO concepts *on the generator level*. In the following example, a Java class as well as an (empty) main method are defined using the tool *Jenerator*:

```
public class HelloJenerator {
  public static void main( String[] args ) {
    CClass createdClass =
           new CClass("demo", "HelloWorld" );
    CMethod mainMethod =
            new CMethod( CVisibility.PUBLIC,
                         CType.VOID, "main" );
    mainMethod.addParameter(
      new CParameter( CType.user( "String[]" ), "args" )
    );
    mainMethod.setOwnership( Cownership.STATIC );
    createdClass.addMethod( mainMethod );
  }
}
```

This program is very long. A useful refactoring would consist of a generator class *MainMethod* – Java's main methods by definition always have the same signature:

```
public class MainMethod extends CMethod {
  public MainMethod() {
    super(CVisibility.PUBLIC, CType.VOID, "main" );
    setOwnership( Cownership.STATIC );
    addParameter(
      new CParameter( CType.user( "String[]" ),
          "args" ) );
  }
}
```

As you can see, the *HelloJenerator* example from above is simplified considerably:

```
public class HelloJenerator {
  public static void main( String[] args ) {
    CClass createdClass =
       new CClass( "demo", "HelloWorld" );
    createdClass.addMethod( new MainMethod() );
  }
}
```

You can imagine that efficient, domain-specific generators can be built with these generator classes. Flexibility is achieved through suitable parametrization of the generator classes.

Such generators are clearly specific to the abstract syntax of the target language, not necessarily to their concrete syntax. If one had different languages with the same abstract syntax, one could generate different target languages simply by exchanging the code generator backend. This is possible, for example in the context of the .NET framework. Using CodeDM, one defines an abstract syntax tree based on the abstract syntax that is predefined for .NET languages in the context of the common-language runtime (CLS). One can generate the concrete syntax for any .NET language (C#, VB, C++) by selecting a suitable implementation of *ICodeGenerator*.

Byte code modifiers, a type of tool that is especially popular in the Java universe, are usually also API-based generators. They usually operate on the abstraction level of the JVM byte code, although some provide higher-level abstractions while technically still manipulating the byte code. .NET IL code can also be generated directly with .NET's CodeDOM.

9.3.5 In-line Generation

In-line generation refers to the case in which 'regular' source code contains constructs that generate more source code or byte/machine code during compilation or some kind of preprocessing. Examples are C++ preprocessor instructions or C++ templates.

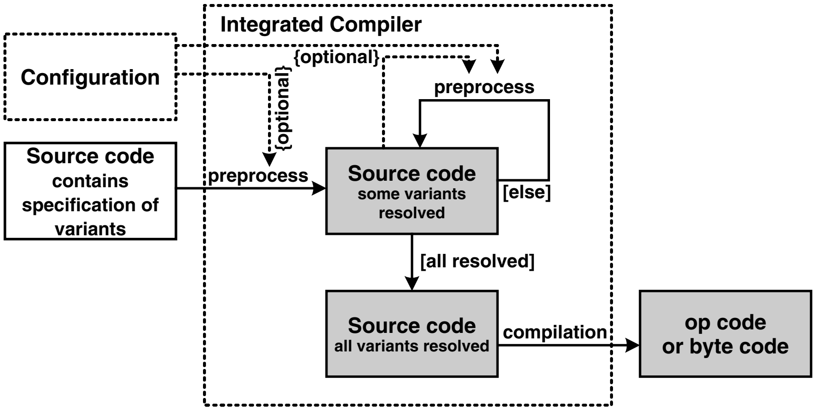


Figure 9.6 In-line generation

A trivial example based on the C++ preprocessor could look as follows:

```
#if defined (ACE_HAS_TLI)
static ssize_t t_snd_n (ACE_HANDLE handle,
  const void *buf, size_t len, int flags,
  const ACE_Time_Value *timeout = 0,
  size_t *bytes_transferred = 0);
#endif /* ACE_HAS_TLI */
```

Here, the code between the *#if* and the *#endif* is only compiled if the flag *ACE_HAS_TLI* has been defined. More complex expressions with parameter passing are also possible:

```
#define MAX(x,y) (x<y ? y : x)
#define square(x) x*x
```

If application code contains the statement *MAX (v1, v2)*, this is textually replaced according to the rule defined before. Thus, *MAX (v1, v2)* is replaced with *(v1<v2 ? v1 : v2)* by the preprocessor. All of this is purely based on text substitution and no type constraints or precedence rules are observed. As a consequence, this approach is only useful for simple cases.

In comparison, template metaprogramming allows a more structured approach, because the processing of templates by the compiler provide a Turing-complete functional programming language that operates on C++ types and literals. One can therefore write entire programs that run during the compilation process. The following calculates the factorial of an integer at compile time:

```
#include <iostream>
using namespace std;

#include "../meta/meta.h"
using namespace meta;

struct Stop
{ enum { RET = 1 };
};

template<int n>
struct Factorial
{ typedef IF<n==0, Stop, Factorial<n-1> >::RET
  PreviousFactorial;
  enum { RET = (n==0) ? PreviousFactorial::RET :
  PreviousFactorial::RET * n };
};

void main()
{ cout << Factorial<3>::RET << endl;
}
```

To find out how this works is an exercise we leave to our readers – if you want to cheat, you can look it up in [EC00]. Due to the clumsy syntax and the occasionally very strange error messages, using this approach is only advisable in exceptional cases, and is not suitable for more complex model-driven projects. This is mainly because the compilers that actually run the metaprograms were neither created nor optimized for these purposes.

9.3.6 Code Attributes

We continue with another mechanism that is very popular in the Java field: code attributes. In the Java world these were first used by *JavaDoc*, where special comments were used to enable automatic generation of HTML documentation. The extensible architecture of *JavaDoc*, makes it possible to plug in custom tags and code generators. Probably the most popular example is *XDoclet* [XDOC]. XDoclet serves, among other purposes, to generate EJB *Remote/Local Interfaces* as well as deployment descriptors. The developer writes the implementation class manually and adds the required XDoclet comments to the class, which are then read by the XDoclet code generator. Furthermore, the generator has access to the source code's syntax tree, to which the comments are added. In this way, the generator can derive information from the comments as well as from the code itself.

The following is an example of a Java class that has been supplemented with XDoclet comments.

```
/**
 * @ejb:bean type="Stateless"
 *           name="vvm/VVMQuery"
 *           local-jndi-name="/ejb/vvm/VVMQueryLocal"
 *           jndi-name="/ejb/vvm/VVMQueryRemote"
 *           view-type="both"
 */
public abstract class VVMQueryBean
    /**
     * @ejb:interface-method view-type="both"
     */
    public List getPartsForVehicle( VIN theVehicle ) {
        return super.getPartsForVehicle( theVehicle );
    }
}
```

The central idea behind this method is that much of the information needed by the generator is already present in the code. The developer only has to add a few special comments. The generator has the AST of the code as well as the additional comments at its disposal.

An often-heard criticism in this context is that such tools are necessary only because J2EE (and primarily EJB) require so much redundancy in the code that it can no longer be handled manually. This is certainly true – nevertheless, generation via code attributes is not restricted to the generation of EJB infrastructure code. One can quite elegantly create persistence mappings for *Hibernate* [HIBE], or similar frameworks with *XDoclet*.

.NET offers the attribute mechanism as an integral concept of the .NET platform. Various source code elements – methods, attributes, classes – can be assigned attributes, as shown in the following example:

```
[QoSServicePriority(Prio.HIGH)]
class SomeService : ServiceBase {
    [QoSTimeLimit(100, Quantity.MS)]
    public void processRequest( Request r ) {
      ….
    }
}
```

Here we specify in a purely declarative way that the instances of this class have a service priority *HIGH* and that the execution of the operation *processRequest()* may take a maximum of 100 ms. The idea behind this is that the service is executed in a framework that measures parameters such as the execution time of the service operation. It can then make, for example, a log entry if the limits set by the attributes are exceeded, or alternatively stop accepting client requests, to lower the system load, and throw an exception to the client.

The framework's access to the attribute (and thus to the defined time limit) is achieved using reflection. Technically this is realized as follows: attributes are simply serializable .NET classes that are instantiated by the compiler during compilation and serialized into the respective *Assembly.* These objects can then be accessed via reflection. Code generators can read this information from the compiled .NET Assembly and – as with XDoclet – use it as a basis for generation.

Note that such features are also available in Java, starting with Version 5. Annotations can be added to many source elements, such as classes, attributes, or operations. Technically, this works the same way as in .NET, in that the compiler instantiates data that is stored with the byte code and can subsequently be queried using reflection.

9.3.7 Code Weaving

Code weaving describes the intermixing of separate but syntactically complete, and therefore independent, pieces of code. To this end, one must define how these various parts can be put together: such locations are called *join points* or *hooks*. *AspectJ* [ASPJ] is a well-known example of this kind of generator: regular OO program code and aspect code are interwoven, either on the source code or byte code level. Aspects describe cross-cutting concerns – that is, functionality that cannot be adequately described and localized using the available constructs of object-oriented programming.

The following example illustrates an aspect that inserts log statements at all code locations at which methods on instances of the *Account* class are invoked For each method call, the log states *from which method* the respective *Account* method has been called[3]:

```
aspect Logger {
 public void log( String className, String methodName ) {
```

[3] The syntax of AspectJ is evolving constantly. It is therefore quite likely that the syntax shown here will not work with the latest version.

```
 System.out.println( className+"."+methodName );
 }
 pointcut accountCall():  call(* Account.*(*));
 before() calling: accountCall() {
   log( thisClass.getName(), thisMethod.getName() );
 }
}
```

After this aspect has been applied to a system – that is, interwoven with the regular code via the weaver – code along the following lines is created, assuming the weaving happens on source level:

```
public class SomeClass {
  private Account someAccount = ...;
  public someMethod( Account account2, int d ) {
    // aspect Logger
    System.out.println("SomeClass.someMethod");
    someAccount.add( d );
    // aspect Logger
    System.out.println("SomeClass.someMethod");
    account2.subtract( d );
  }
  public void anotherMethod() {
    //aspect Logger
    System.out.println("SomeClass.anotherMethod");
    int bal = someAccount.getBalance();
  }
}
```

Another example of this *invasive composition* of source code is the *Compost Library* [COMP]. This ultimately provides an API to change the structure of programs based on their AST. Compost is implemented in Java and also operates on Java source code. An additional library, the *Boxology Framework*, allows systematic modification of source code via hooks. We have to distinguish here between explicit hooks declared by the original developer of the source code to be modified, and implicit hooks. Implicit hooks are specific, well-defined locations in the AST of a program, such as the *implements* hook. Through the extension of that hook, further interfaces can be implemented. This framework can therefore serve as a basis for a wide variety of source code modification tools.

9.3.8 Combining Different Techniques

Combinations of the different code generation techniques are also possible. The Open Source tool *AndroMDA* [ANDR] creates source code via templates. This source code again contains code attributes. The template-based generation of this code takes place using *Velocity* [VELO], and further processing with XDoclet. The following is an example of a Velocity template with XDoc comments:

```
// -------------- attributes --------------------
#foreach ( $att in $class.attributes )
#set ($atttypename =         transform.findFullyQualifiedName($att.type))
    private $atttypename ${att.name};

   /**
#generateDocumentation ($att "    ")
    *
#set ($attcolname = $str.toDatabaseAttriName(${att.name}, "_"))
#set ($attsqltype = $transform.findAttributeSQLType($att))
#if ($transform.getStereotype($att.id) == "PrimaryKey")
    * @hibernate.id
    *     generator-class="uuid.string"
#else
    * @hibernate.property
#end
    *     column="$attcolname"
    *
    * @hibernate.column
    *     sql-type="$attsqltype"
```

Another popular combination are API-based generators that can optionally read templates to simplify handling of the API.

9.3.9 Commonalities and Differences Between the Different Approaches

First, we classify the different approaches based on the criteria listed in Section 9.2:

	Time of Compilation	Program/Metaprogram	Generated/Manually-created code
Templates and filtering	before	separate	separate
Template and metamodel	before	separate	separate
Frame processors	before	separate	separate
API-based generators	before/during/after	separate	separate
In-line generation	before/during	mixed	integrated
Code attributes	before/during	separate/mixed	separate
Code weaving	before/during/after	separate	integrated

This table needs to be explained, particularly those parts where more than one option is listed:

- API-based generation can occur either prior to actual compilation via an integrated preprocessor, or during compilation using compile-time meta object protocols, as well as at runtime using runtime meta object protocols.
- In-line generation can take place either before actual compilation via an integrated preprocessor, or during compilation, for example via Lisp's quoting mechanisms, C++ templates and so on.
- Code attributes are either evaluated during compilation, as for example in .NET, or beforehand using a preprocessor such as XDoclet. Depending on the generator, the generated code can be embedded directly in the source code or written to separate artifacts.
- Code weaving can also take place during a separate run prior to compilation, during compilation (with fully aspect-oriented languages) and at runtime (with dynamic aspect-weaving). In Java, interweaving at load time via special class loaders is common.

At this point we need to address a number of additional characteristics, differences, and commonalities. In principle, frame processors and API-based generators build an AST of the system to be created. For both approaches, the alignment with the problem domain – and thus the abstraction level and efficiency – can be increased by inserting domain-specific generator constructs, for example a frame that generates a JavaBeans property. As a rule, one always starts with the AST of the target language/platform.

If the 'templates and metamodel' approach is used, the generator builds an AST of the model at runtime – a representation of the problem space, depending on the metamodel, thus starting on a higher abstraction level. The templates do the translation work toward the target platform. This kind of generator is particularly suited for application in cases where the problem space's metamodel is already complex.

A question that often comes up is whether API- or template-based generators are better in this context. In our opinion, template-based systems are better suited when large amounts of identical code are created. API-based generators are more efficient when finely granular code is to be created, for example state machines or algorithms. In the context of architecture-centric MDSD, template-based generators are preferable.

Code attributes can be considered as a form of in-line generation. The code can be generated directly into the location where the specification (the attribute) in the base program source code is located. However, in almost all cases this does not happen: the code generated from the base program and the attributes is in most case external and complete and does not have to be integrated with handwritten code, because it usually covers technical aspects such as persistence or EJB 'glue' code. Code attributes are recommended if you *do not* work completely model-driven, but you still want to generate specific artifacts. The approach is limited in that it only works if the necessary input for the generator can be reasonably specified using source code structures plus the additional information in the attributes.

The main difference between in-line generation and code weaving is that the latter approach can be used to localize cross-cutting concerns in a non-invasive way: the code to be modified need not be changed manually: it is modified from the outside by the weaver instead. Both approaches are specifically useful if you work exclusively with code and not with models.

9.3.10 Other Systems

A number of other systems exist in the field of code generation. However, they are not very relevant for MDSD. A short overview of such systems can be found in [Voe03]. We will list only two examples here:

- *Meta object protocols* (MOPs) allow access to compiler structures – compile-time MOPs such as *OpenC++* [OC++] change the compiled program, while runtime system MOPs such as CLOS [KRB91] change the running program.
- A number of tools exist primarily in the context of Java that allow the modification of generated byte code. For example, transparent accesses to an OO database can be generated into the relevant locations in the byte code. Examples of this include *BCEL* [BCEL] and *Javassist* [JASS].

10 Model Transformation Techniques

by Simon Helsen

Model-to-model transformations are a contentious topic, partly because they are not very well understood, and partly because their merit in practical model-driven development scenarios is not very clear. This is something of a ‘chicken and egg’ problem, of course, because the lack of understanding of the underlying problems and mechanisms to solve them is feeding the lack of understanding about where and how to apply these kinds of transformations.

Nevertheless, model-to-model transformations – referred to as *M2M* transformations – could become an important mechanism to bridge some of the abstraction gaps that occur in MSDS. Not surprisingly, countless attempts to develop M2M transformation languages have been made in academia, Open Source communities, and commercial companies.

In this chapter we focus on model-to-model transformations with the *Query / View / Transformations* standard of the OMG, also known as QVT. However, since the QVT standard has turned out to be a rather voluminous and complex specification, we only give a high-level overview of its architecture and features. An elaborated example is used to give you a sense of what QVT transformations look like. We also give a brief overview of its history and future, as well as a critical assessment.

A more comprehensive account is outside the scope of this book, but the adopted QVT specification document [QVT] is now publicly available from the OMG [OMG] Web site[1]. A very good discussion of the various model transformation techniques (beyond QVT) can be found in a paper by Czarnecki and Helsen [CH05].

10.1 History

The MDA guide as it is defined by the OMG [MDAG] often talks about transformations between different models at different levels of abstraction. For example, it assumes that a typical MDA-based development scenario entails the transformation of a platform-independent model (PIM) to a platform-specific model (PSM) before generating code from the latter. The MDA guide intentionally does not say how this is supposed to happen, and this is where QVT comes into play.

[1] The adopted specification is not yet finalized, which means minor changes may be made to it. See also Section 10.5.

The OMG originally issued a *Request for Proposal* (usually referred to as an RFP) for model-to-model transformations in April 2002 [QVTR]. It took until November 2005 before the adopted specification was eventually released. The time required to come up with the QVT specification is relatively long even by OMG standards, and the standardization process is still not entirely finished (see Section 10.5). To address code generation from MOF models, the OMG issued an RFP in April 2004 [M2T], but that standardization process is currently on-going and outside the scope of this book.

The rather long standardization time can partly be explained by the intrinsic complexity of the problem. Although lots of people had good ideas on how to write programmatic model-to-model transformations in, say Java, it quickly became clear that realistic M2M scenarios required more sophisticated techniques. The QVT RFP nevertheless explicitly asked for proposals that addressed this level of sophistication, even though such requirements were largely unexplored and not understood at the time.

Another problem was that no or little prior experience existed with model-to-model transformations in the first place. This was not a very good starting position from which to come up with a standard, which ideally is a consolidation of existing technology. The situation was worsened by the fact that eight different groups submitted initial responses. Most of these RFP responses were so different that there was no clear basis for consolidation. As a result, it took a considerable amount of time to find common ground, and the result today still specifies three different QVT languages that are only loosely connected.

10.2 M2M Language Requirements

Before we delve into the details of QVT itself, it is helpful to discuss some of the more important and perhaps less obvious requirements for model-to-model transformation languages and their implementations:

- Most realistic model-to-model transformation scenarios require transformation rules to have the ability to look up what other transformation rules have calculated, because each rule usually addresses only one small aspect of the entire transformation. A look-up is only possible if the engine has the ability to do some book-keeping of the *transformation trace*. A transformation trace can be understood as a runtime footprint of a transformation execution. The exact form and user visibility of such a trace is widely different between different M2M languages. Usually, imperative transformation languages have a more explicit look-up procedure, while declarative languages have an implicit way of exploiting the trace.

  Even the three QVT languages expose the trace in very different ways. Transformation traces are also required to implement all sorts of optimization and debugging scenarios. As with any practical programming language, the ability to debug is of critical importance to uncover difficult and subtle bugs. This is no different for model transformations. For example, sophisticated model-to-model transformations may behave subtly differently depending on the values of one small property in the source model. If the resulting target model shows anomalies, it is extremely helpful for a transformation writer to follow the execution trace and discover which rules are making the wrong decisions.

- Apart from when M2M transformations are first run, users rarely generate models on a clean sheet. If a transformation is run again, it is important that it only makes required changes to the existing target model, and does not simply keep adding model elements for every transformation run. This kind of target model understanding is only possible if there is an identification mechanism on the target model. This can be achieved by intrinsic target metamodel properties, such as the name of a model element or even the MOF id, or, alternatively, by encoding the identification in the actual relationship between a target model and its source model. Usually, the transformation trace contains this information. The requirement to have repeated transformation runs correctly update the target model is sometimes known as *change propagation*.
- A transformation implicitly or explicitly defines a relationship between its source and target models. In some scenarios, both the source and target models exist before a transformation is executed and therefore, before their relationship was established. In this case, a transformation may be asked to verify if the relationship exists and optionally change the target model just enough to make the relationship happen. This problem is different from the change propagation scenario, because for the former one can assume the existence of a transformation trace, whereas in the latter scenario the transformation may not have ever run.
- Generally, the source model of a transformation may be extremely large. After the first transformation execution, only comparatively small changes are usually made to the source model. In this case it should be possible to approximate which transformation rules have to be executed again and on what subset of the source model elements. An *impact analysis* on the transformation rules, as well as the availability of trace information, may be required to implement this. The need for such an optimization cannot be underestimated, as transformation users will expect reasonably fast turn-around cycles during development. This is comparable with non-MDSD development scenarios today in which entire builds of large projects are rare and usually happen overnight on a build server. This optimization requirement is sometimes referred to as *incremental update*.
- There are many use scenarios in which the target models, which are often platform-specific, may require manual changes by modelers. Such changes can often be avoided, as they might merely indicate a problem with the more abstract platform-independent model, or even with the transformation itself. Nevertheless, the PIM sometimes does not fix specific platform details on purpose and expects the platform modeler or developer to make controlled changes before another model transformation or code generation is applied. For this to be possible, the model transformation writer needs the ability to define where in the target model such changes are permitted. The transformation engine can then avoid overwriting these – and only these – manual changes. This is reminiscent of the protected area problem for code generation discussed in Section 8.3.1. The ability for a transformation writer to define how target models can be changed manually is sometimes known as the *retainment policy.*
- Model-to-model transformations often make substantial structural changes to the source model when they map it to a target model. Usually, this structures the transformation into multiple phases. In a typical M2M scenario, some of the classes in the source metamodel are mapped onto classes in the target metamodel in an initial phase. In a second

phase, some of the source metamodel associations are mapped onto target metamodel associations. However, since the latter are usually encoded as properties, it might not be possible to construct one target model element in one operation. More generally, it is important that target model elements can be constructed incrementally. When this is the case, it is said that the language allows the definition of *M x N transformations*.

- It is contentious whether support for *bidirectional transformations* is a requirement. This can be achieved by writing two or more unidirectional transformations, or one transformation, which can be executed in both directions. The latter would only be possible in a declarative scenario. However, we question the usefulness of this requirement in practice. A bidirectional transformation can only be defined meaningfully when it describes an isomorphic relationship. In practice, this is almost never the case, as different metamodels usually describe aspects at different levels of abstraction with different completeness.

  Even if a bidirectional transformation for a non-isomorphic problem is given (whether by two unidirectional transformations, or in one transformation that can be executed in two directions), it would be difficult to use in practice, as changes in both source and target model can easily oscillate in an unexpected and uncontrolled manner. In the worst case, changes in source and target models by different parties will render both models useless, as the transformation by necessity has to make defaulting assumptions because problem is not isomorphic. Such transformations would invalidate the architectural approach advocated in this book that architectural decisions should be made at the PIM level only.

The above are some of the more important reasons why it may not be sufficient to write model-to-model transformations in a general-purpose programming language like Java. If one does decide to write transformations in Java, a sophisticated framework would be required to support the transformation writer in managing the input and output models and the transformation trace: without the availability of an implicit or explicit trace, no useful transformations can be written. However, this would make the use of the Java debugger very hard, because users would have to deal with the internals of the framework. The ability to debug a transformation is however mandatory in practice.

One of the design goals of QVT was to either support, or at least not to prohibit, an implementation from fulfilling at least some of the requirements listed above. QVT ended up providing three domain-specific languages, each of which addresses model-to-model transformations in its own way. In Section 10.6 we give an assessment of whether QVT has lived up to its expectations and achieved its design goals.

You may wonder why QVT stands for ‘Queries, Views, and Transformations’, as it really only deals with model-to-model transformations. *Queries* are an intrinsic part of any model-to-model transformation, as they are used to gather model elements from the source model. In QVT, this is achieved with OCL [OCL]. The idea of a *View*, meanwhile, is to provide a means of looking at a specific aspect of a metamodel. It is thought that model transformations provide a mechanism to do so, although the currently-adopted specification explicitly avoids addressing the view problem. If the view needs to be editable, we bump into the bidirectional transformation requirement and its problematic consequences again. QVT claims to support bidirectional transformations because two of the three QVT languages have the ability to specify bidirectional rules, but it does not indicate how views could be defined with it.

10.3 Overall Architecture

The QVT specification comprises three different model-to-model transformation languages: two, the *Relations* language and the *Core* language are declarative, one, the *Operational Mappings* language, is imperative. The hybrid nature of QVT was introduced to accommodate different types of users who have different needs, requirements, and habits. This strategy may not come as a surprise if you consider the numerous initial submitters to the RFP, each with different expectations on the use and functionality of QVT:

- The *Relations* language is a declarative user-friendly transformation language that is primarily built around the concept of object patterns. The user is not responsible for the creation and deletion of objects, nor for the management of transformation traces. The language expects a user to describe the relationships between parts of the source and target metamodels by means of object patterns and OCL expressions. It provides a mechanism to identify target model elements, which is required to support change propagation. The QVT Relations language also defines a simple graphical syntax. We discuss the Relations language by means of an example in Section 10.4.1.
- The *Core* language is defined as an absolutely minimal extension to EMOF[2] and OCL. Here too the user is not responsible for object creation and deletion, but traces are not automatically generated. The user is instead expected to define transformation rules and trace information as a MOF metamodel. The Core language does not provide patterns, nor any direct mechanism for the identification of target model elements.

  This absolutely minimal approach makes the Core language beautiful in its simplicity, but almost impossible to use in practice. This is partly caused by the absence of automatic trace management, as well as the difficulty of dealing with target model element identification. The latter has to be explicitly encoded in rules and subrules, requiring a complex transformation specification for relatively simple transformation problems. Because of this, we do not elaborate the Core language any further.
- The *Operational Mappings* language is the imperative cornerstone of QVT. It provides a domain-specific imperative language for describing transformations. OCL is used as its query language, but extended with imperative features to describe computations. The Operational Mappings language can be used in two different ways. First, it is possible to specify a transformation purely in the operational mappings language. We illustrate this possibility by means of our example in Section 10.4.2.

  Alternatively, it is possible to work in a hybrid mode. The user then has to specify some aspects of the transformation in the Relations (or Core) language, and implement individual rules in the Operational Mappings language as *black-box mappings*.

Although the QVT standard specifies three transformation languages, they are not entirely disconnected. Figure 10.1 illustrates the relationships between the different QVT languages.

[2] EMOF is a minimal subset of MOF 2.0 akin but not identical to EMF, the meta meta model defined as part of Eclipse.

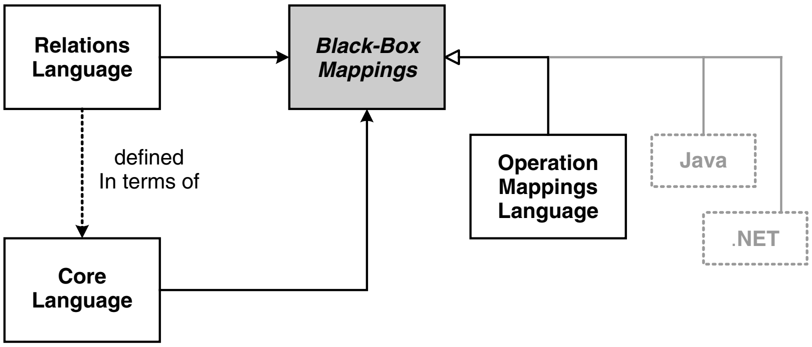


Figure 10.1 QVT language architecture

The Relations language is semantically defined in terms of the Core language. In the specification, this is modeled by a model-to-model transformation from the MOF metamodel of the Relations language to the MOF metamodel of the Core language. This semantic transformation is itself specified in the Relations language. A discussion of this mapping would be beyond the scope of this book, but if you're interested, you can find this transformation in the QVT specification document, where it is also extensively commented [QVT].

This explicit transformation between the Relations and Core languages suggests that a Relations language could be implemented on top of a Core language engine by translating a Relations transformation with the above transformation. Although this is possible in theory, it seems hardly a practical approach. It is clearly more viable to develop an optimized QVT engine for the Relations language, which better supports the different requirements for model-to-model transformations, than taking a detour via the Core language.

In connection with this relationship between the Relations and Core languages, the QVT document makes an analogy to the JVM and the Java programming language: the Core language is more like Java byte code, whereas the Relations language is a little like the Java language itself. You must judge the value of this analogy for yourself.

In practice it may not always be possible to specify all aspects of a transformation in the Relations (or Core) language because the user has only OCL to express computational problems. For example, it is possible that model-to-model transformations are required to use complex or legacy libraries that it would not be economical to re-implement in pure QVT. To accommodate this, the QVT specification explicitly allows for black-box mappings. As Figure 10.1 suggests, these black-box mappings may be written in the Operational Mappings language, which is then used in hybrid mode. As an alternative, it is also possible to use external programming languages such as Java.

In the sections that follow we investigate a concrete transformation problem and discuss its implementation in both the Relations and Operational Mappings languages. However, since it is not possible to discuss all the features and intricacies of these two M2M transformation languages, we recommend that interested readers work with the prototypes of these languages as soon as they are available (see Section 10.5).

10.4 An Example Transformation

In Section 6.11 we discussed the ALMA metamodel example, which can be used to model astronomic observational data. It provides a platform-independent abstraction over all the possible artifacts required for the ALMA software infrastructure. One possible platform-specific incarnation of ALMA data could be a relational database. In this section, we discuss a transformation from the ALMA metamodel to a simple relational database metamodel, referred to as *DB*.

Before discussing the actual transformation, we have to specify the input (or source) and output (or target) metamodels of the QVT transformation accurately. Since QVT assumes MOF-based metamodels for its input and output, we show a MOF rendition of the ALMA metamodel. Note that it is in theory possible to specify a QVT transformation directly on the UML profile, where the source metamodel would then have to be UML itself. However, because the UML metamodel as an instance of MOF is extremely large and complex, such transformation are often difficult to understand, let alone write. The QVT transformation writer is then required to understand the UML metamodel as an instance of MOF, which may not be straightforward and, more importantly, may distract from the domain-specific intentions of the metamodel.

A UML Profile is in many ways a concrete syntax model. From practical experience, we have observed that well-structured and adaptable transformations[3] written directly against a UML profile tend to first transform the profile into an implicit MOF-like metamodel before implementing the actual transformation logic. This is particularly important when the transformation is the basis for reuse and adaptation[4].

Figure 10.2 shows the ALMA metamodel as an instance of MOF.

The main differences with the UML profile are that we introduce the abstract class *Record* and the concrete class *Field*, instead of hanging on to UML's *Class* and *Attribute* metaclass. We also introduce specific metaclasses for the *PhysicalQuantityType* and its different incarnations for each primitive type. This reduces a large number of constraints on the metamodel that were required in the definition of the UML profile.

Figure 10.3 shows the example of Figure 6.28 with some slight adaptations to show the use of a physical quantity type. Note that this example is not quite an instance of the UML profile of Section 6.11, which illustrates that there is some freedom in how to map a metamodel onto a profile.

[3] These observations stem primarily from model-to-text transformations: however, we believe that they equally apply to M2M transformations.

[4] The attentive reader might wonder how we relate a UML profile with a MOF metamodel. In theory, this could also be achieved with an M2M transformation, but in practice, it is more likely the job of the hosting MDA tool because to be practical, the connection between the profile and metamodel has to be live and bidirectional. More importantly, the tool may want to put restrictions on the permitted mappings from the metamodel to the profile, and thus might have to provide its own mechanism or language to express this relationship.

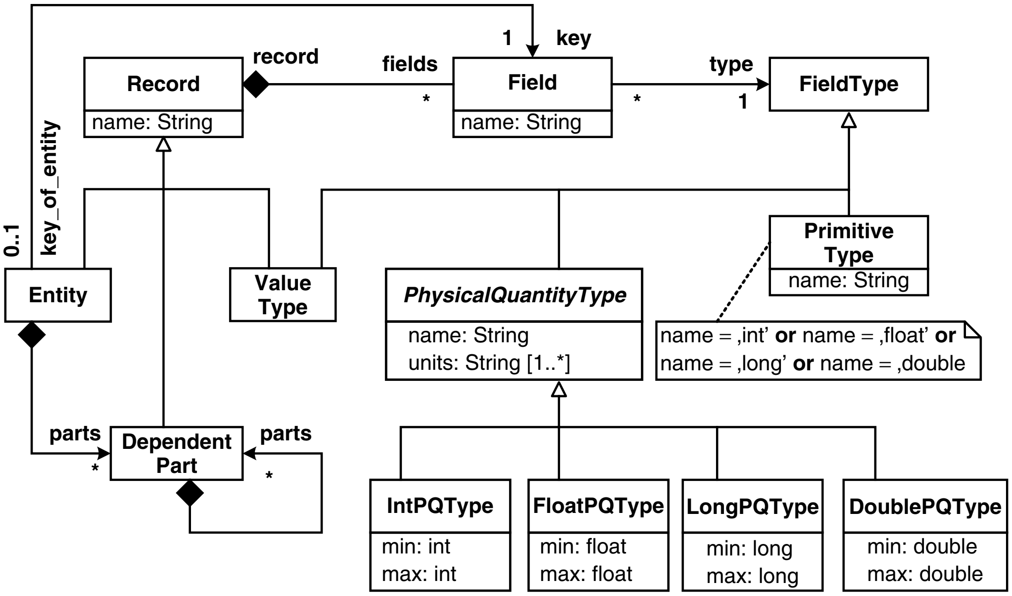


Figure 10.2 ALMA metamodel as an instance of MOF

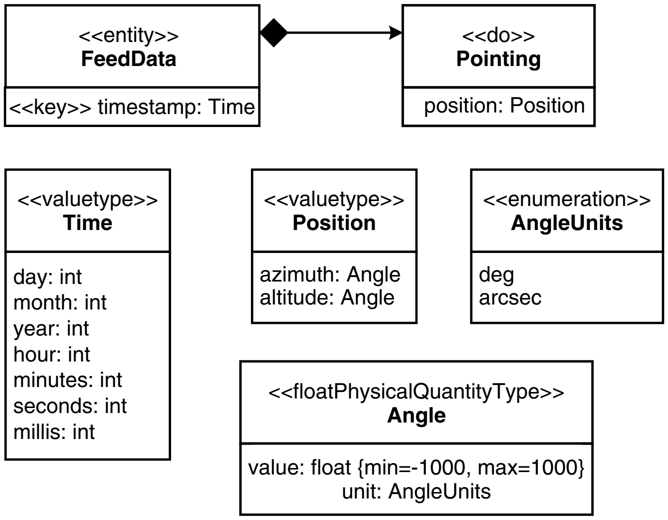


Figure 10.3 An example ALMA model instance

Our target DB metamodel is a simple rendition of relational database tables. Figure 10.4 shows its MOF instance.

A DB model may contain any number of tables, each having one primary key, any number of columns, and any number of possible foreign keys. The key refers to one or more columns, where a constraint requires those columns to be owned by the table of the key. Each foreign key of a specific table refers to the primary key of another table, and we disallow a foreign key to refer to its own table. Finally, both tables and columns have names and each column has a primitive type, for which we only allow numeral types.

The example transformation from the ALMA metamodel to DB metamodel can be described informally as follows:

- All fields of a record are mapped to one ore more columns depending on the field type:
  – If the type is primitive, we construct one primitive typed column.
  – If the type is a value type, we recursively construct columns for each of its fields, where the name of the encompassing field is propagated to disambiguate the names of the nested fields.
  – If the type is a physical quantity, we construct one column for each unit, where the name of the column incorporates the unit name and its type is that of the concrete physical quantity.
- Each ALMA entity is mapped to a DB table:
  – All its fields lead to columns, as described before.
  – Its key leads to the table key.

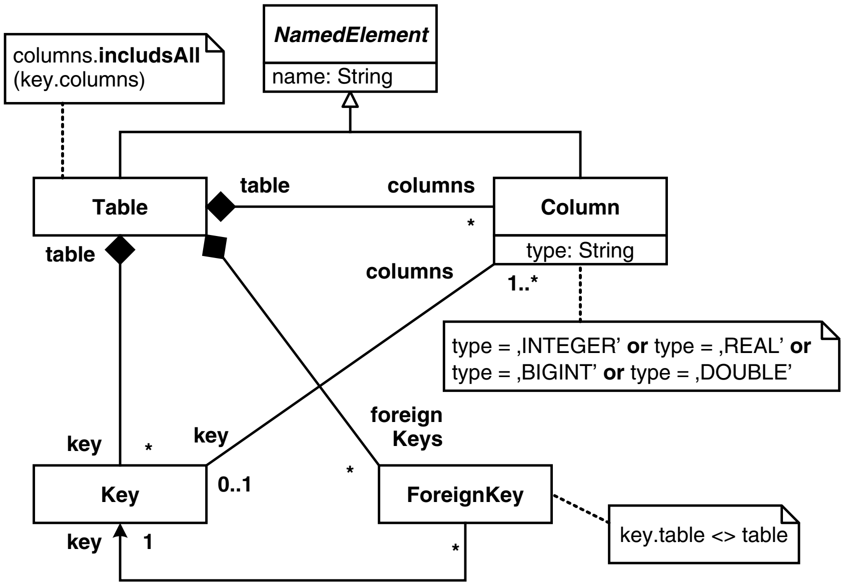


Figure 10.4 Simple DB metamodel as an instance of MOF

- Each ALMA-dependent part that is owned by an entity is mapped to a DB table as well, where its name is a concatenation of the entity name and the dependent part name:
  – All its fields lead to columns, as described before.
  – Its key is artificially constructed and of type INTEGER.
  – The containing table for the entity obtains a foreign key pointing to this artificial key.

- All ALMA-dependent parts that h are parts of other dependent parts have all their columns (for each of their fields) expanded into the table of the topmost dependent part. This happens recursively as well.

The simple example of Figure 10.3, transformed with the above transformation description, would lead to the following two tables[5]:

1. Table **FeedData**:
   - key *timestamp_day* : INTEGER
   - key *timestamp_month* : INTEGER
   - key *timestamp_year* : INTEGER
   - key *timestamp_hours* : INTEGER
   - key *timestamp_minutes* : INTEGER
   - key *timestamp_seconds* : INTEGER
   - key *timestamp_millis* : INTEGER
   - fk *key_FeedData_Pointing* : INTEGER
2. Table **FeedData_Pointing**:
   - key *key_FeedData_Pointing* : INTEGER
   - *position_azimuth_as_Angle_in_deg* : REAL
   - *position_azimuth_as_Angle_in_arcsec* : REAL
   - *position_altitude_as_Angle_in_deg* : REAL
   - *position_altitude_as_Angle_in_arcsec* : REAL

10.4.1 The Example in the QVT Relations language

The QVT Relations language is a declarative member of the QVT trio. A user of QVT Relations has to describe the transformation of a source metamodel to a target metamodel[6] as a set of *relations*. A transformation execution means that these relations are verified and then, if necessary, enforced by manipulating the target model.

For our example, we describe how an instance of ALMA relates to an instance of DB. A transformation declaration looks like this:

```
transformation alma2db(alma : AlmaMM, db : DbMM) {
...
}
```

The direction of an execution is not fixed when the transformation is defined, which means that in theory both *alma* and *db* can be the source and target model and vice versa. Only when invoking the transformation must the user specify in which direction the transformation has to

[5] One might define a UML profile as a concrete syntax for DB models, but since this is not relevant for the transformation we do not discuss it further.

[6] All QVT languages allow for multiple input and output models.

be executed. This direction alters the interpretation of the individual relations, as we explain below.

Here is the relation rule that maps an entity to a table:

```
top relation EntityToTable {
    prefix, eName : String;

    checkonly domain alma entity:Entity {
        name = eName
    };
    enforce domain db table:Table {
        name = eName
    };
    where {
       prefix = '';
       RecordToColumns(entity, table, prefix);
    }
}
```

A relation rule always has as many *domain declarations* as there are models involved in the transformation. A domain is bound to a model (for example *alma*) and declares a *pattern* that will be matched with elements from the model to which the domain is bound. Such patterns consist of a variable and a type declaration, which in itself may specify some of the properties of that type, and recursively so for the types of those properties. Such patterns simply constrain the model elements and properties in which we are interested for this relation rule.

For example, in the relation *EntityToTable*, the domain binding the *alma* model will match all elements of type *Entity* in the *alma* model to the variable *entity*, provided they at least define a property *name*, which has to be bound to the string variable *eName*. Similarly, in the domain for *db*, the pattern binds the variable *table* of type *Table*, while property *name* is bound to the variable *eName*. Observe that both patterns refer to the same variable *eName*, which implicitly is a *cross-domain* constraint. Additional cross-domain constraints can be specified in the *where* clause of a relation rule.

Before discussing the contents of the *where* clause, we want to draw your attention to the domain qualifiers *checkonly* and *enforce*. These qualifiers constrain how a relation can be executed for a given direction. For example, if the *alma2db* transformation is executed in the direction of *db*, the QVT engine will try to match all domain patterns of a rule that are not part of the direction (conveniently called *source domains*). For each source domain match, the engine will search for matches in the domains of the direction (called *target domains*). If there is no or only a partial match and the target domain is qualified with *enforce*, the engine will alter or create the model elements of the target domain to (re-)enforce the relation. If, on the other hand, the target domain is qualified with *checkonly*, the engine will notify the user of an inconsistency, but will not try to correct the model in the target domain.

In our example, when *alma2db* is executed in the direction of *db*, the *entityToTable* rule will match elements in *alma* of type *Entity* and check if a corresponding element of type *Table* with the same name exists in *db*. If not, the QVT engine will change or create a table in *db*.

The QVT Relations engine will also delete any tables that have no corresponding entity in the *alma* model. The other way around, whenever we execute *alma2db* in the direction of *alma*, the *entityToTable* rule will match elements in *db* of type *Table* and check if a corresponding element of type *Entity* exists in *alma*. If this is not the case, the user will be notified, but no changes will be made in the *alma* model.

In theory it is possible to qualify all domains of a relation with *enforce*, which amounts to a bidirectional or even multidirectional transformation rule. However, as we pointed out in Section 10.2, this does not work well in practice. Moreover, the QVT relation language constrains the format of expressions involved in the *enforced* domain to guarantee executability. It is considerably more difficult to express all the required computations when all domains are enforced.

After a successful match of the source domain pattern, the target domain pattern is checked and enforced with the constraints of the *where* clause. In the example, the *entityToTable* rule demands that the variable *prefix* is bound to the empty string and demands that the relation *RecordToColumns*, with the given arguments exists or is constructed. In a way, the predicate *RecordToColumns*(*entity*, *table*, *prefix*) can be interpreted as a rule call.

Relation rules can either be top-level or non-top-level. The former are qualified with the keyword *top* and are executed automatically as soon as a match exists for the source domain patterns. In contrast, non-top-level relation rules are only executed when explicitly called from the *where* clause of another relation. The above *entityToTable* rule is an example top-level relation, whereas the *RecordToColumns* rule, which is called from the *where* clause of *entityToTable*, is non-top-level:

```
relation RecordToColumns {
    checkonly domain alma record:Record {
       fields = field:Field {}
    };
    enforce domain db table:Table {};
    primitive domain prefix:String;
    where {
      FieldToColumns(field, table);
    }
}
```

This relation rule matches records and their fields in *alma* and tables in *db*. It also defines a special *primitive domain*. This mechanism allows non-top level relations to specify parameters of primitive type. They can only meaningfully occur in non-top-level relations, because primitive domains have an infinite number of instances.

You may have wondered how the QVT engine repairs broken elements in an in an enforced target domain. A simple answer could be that for each failing target domain pattern match, the engine simply deletes the entire match and replaces it with a correctly-calculated match. Although semantically correct, this strategy would be very inefficient and, worse, would destroy the underlying identities in the target model, which is very problematic in practice.

For example, consider the following relation rule of the *alma2db* example:

```
relation PhysicalQuantityTypeToColumn {
  pqName, pqUnit, fieldName : String;

  checkonly domain alma field:Field {
    name = fieldName,
    type = pq:PhysicalQuantityType {
      name = pqName,
      units = pqUnit
    }
  };
  enforce domain db table:Table {
    columns = column:Column {
      name = prefix + fieldName + '_as_' +
             pqName + '_in_' + pqUnit,
      type = AlmaPhysicalQuantityTypeToDbType(pq)
    }
  };
  primitive domain prefix:String;
}
```

In the *alma* domain, this rule matches a *Field* of type *PhysicialQuantityType* with the properties *name* and *units* bound to the variables *pqName* and *pqUnit* respectively. In the *db* model, the engine would therefore want to match or repair a column of a table with a name based on *pqName* and a type calculated from the *fieldType*.

Suppose now that for a match in *alma*, a corresponding column is found in *db*, but with a non-matching name: hopefully only the *name* property of *Column* is recalculated and not the entire *Column* object. In QVT Relations, this is achieved with key definitions, which a transformation writer has to provide at the beginning of a transformation. As an example, consider:

```
key Column {table, name};
```

This key definition says that columns are uniquely defined by their name *and* the table to which they belong. If in the above example both the table and the name property of the column had changed in the enforced target domain, a new column object would have been created.

Top-level relations may need additional cross-domain constraints *before* a successful source domain pattern match is allowed to happen. This is illustrated in the following rule:

```
top relation EntityKeyToTableKey {

  checkonly domain alma entity:Entity {
    key = entityKeyField:Field {}
  };
  enforce domain db table:Table {
    key = tableKey:Key {}
  };
  when {
    EntityToTable(entity, table);
  }
  where {
    KeyRecordToKeyColumns(entityKeyField, table);
  }
}
```

This top-level relation specifies a *when* clause. It requires that the pattern bound to *entity* in the *alma* domain is only considered successful whenever there exists an instance of *EntityToTable* with *entity* and *table* as arguments. Only then do we match or construct a *Key* object in the enforced target domain with a link to the *Column* object that was previously calculated and call the non-top-level relation *KeyRecordToKeyColumns*.

Sometimes, it is also useful to write auxiliary functions that do simple calculations. Consider the function *AlmaTypeToDbType* from the *alma2db* example:

```
function AlmaTypeToDbType(almaType : String) : String {
  if (almaType = 'int') then 'INTEGER'
  else if (almaType = 'float') then 'REAL'
  else if (almaType = 'long') then 'BIGINT'
  else 'DOUBLE'
}
```

This calculates the correct string name for the primitive types in *db* for each primitive type of *alma*. This type of function should be understood as a macro: that is, it does not contribute to the transformation trace, and behaves as if it was in-lined.

The QVT Relations language also provides a graphical notation, which extends UML object diagrams. As an example, Figure 10.5 shows the *EntityKeyToTableKey* relation rule in the graphical notation.

The notation is relatively straightforward. Each rule has its own frame with the name at the top-left corner. The two patterns are drawn as object graphs in which the domain variable and type are given the stereotype *«domain»*. The domain bindings and their qualifications are drawn in the middle with a new symbol, where *C* indicates *checkonly* and *E* means *enforce*. Both the *when* and *where* clauses are drawn as compartments of the rule frame. Their contents is then inserted as text.

Unfortunately, the current-adopted QVT specification is not complete with regard to the graphical notation of all possible QVT relation rules. For example, it does not define how primitive domains are to be specified. More generally, it is not clear how much of an advantage the graphical notation gives over the textual representation. Graphical rules are comparatively large and require a lot of work to specify. Moreover, unlike UML or MOF, which are languages for defining structural information, it is not clear how much more readable rules of a behavioral language such QVT become in a graphical form.

You can find the entire *alma2db* example in its QVT-relational textual form in Appendix A.1.

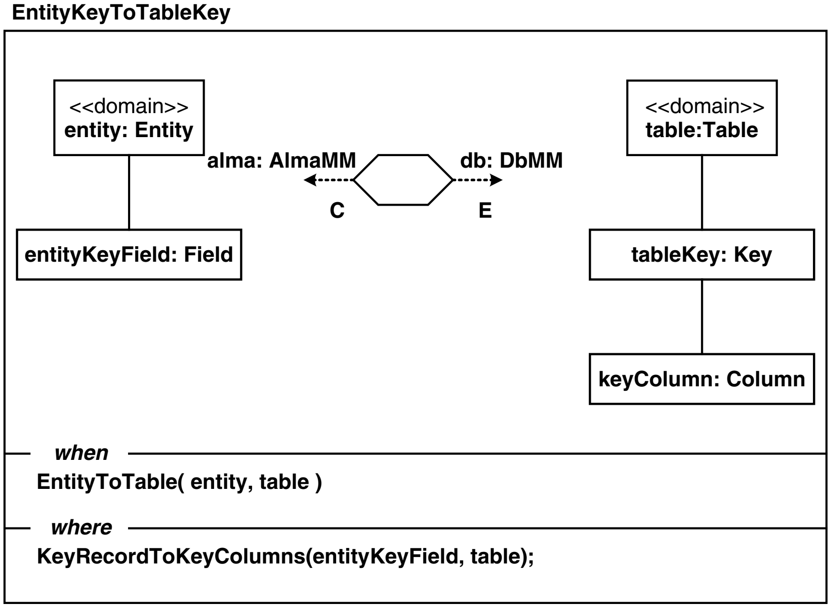


Figure 10.5 Graphical notation of a QVT rule

10.4.2 The Example in the QVT Operational Mappings language

The Operational Mappings language (OM) is the imperative member of the QVT language specification. It can be used in conjunction with QVT Relations in a hybrid manner, or stand-alone. For the sake of the example, we only consider the stand-alone method.

An Operational Mappings transformation starts with the transformation header, which specifies the input and output models of the transformation. An OM transformation can only be executed in the statically-declared direction, as it generally does not address multidirectional transformations. For the *alma2db* example, we have:

```
transformation alma2db(in alma : AlmaMM, out db : DbMM);
```

OM transformations consist primarily of *mapping operations*, which are attached to source metamodel classes. For example, consider the following mapping example from the *alma2db* example:

```
mapping DependentPart::part2table(in prefix : String) : Table
inherits fieldColumns {
  var dpTableName := prefix + recordName;
  name := dpTableName;
 columns := mainColumns +
             object Column {
               name := 'key_' + dpTableName;
               type := 'INTEGER';
               inKey := true;
             }

  end { self.parts->map part2columns(result, dpTableName + '_'); }
}
```

The mapping *part2table* is attached to the *DependentPart* metaclass. It has one input parameter *prefix* of type *String* and, when invoked, either leads to a new *Table* instance or, if the rule is called with a table binding already, updates that table.

The body of the mapping contains property assignments as well as temporary variables (qualified with the keyword *var*). Additionally, mappings may declare an *init*- and *end*-clause. These contain statements that have to be executed before and after the instantiation of the metaclass respectively.

All statements in the body and its clauses are specified in an imperative extension of OCL. We don't discuss this extension in any detail, as it is mostly self-explanatory. Within expressions of mapping statements, one commonly has to refer to the containing object, as an instance of a source metaclass. This object can be referenced with the *self* keyword.

Expressions may also contain *in-lined mappings*, which are qualified with the *object* keyword. In the example, the *columns* property assignment contains such an in-lined mapping. It constructs a *Column* instance, which in its turn is assigned some properties[7].

You may wonder where the property *inKey* came from. In OM transformations, it is possible to extend metaclasses with auxiliary properties. For the example it is sufficient to add the following statement:

```
 intermediate property Column::inKey : Boolean;
```

The *end*- clause of the *part2table* mapping example exploits two features that deserve explanation. The *parts* property of the self object is fed to the *map* operation. This operation simply applies its argument mapping to each element in the collection assigned to the property. Moreover, the mapping *part2columns*, which results in a table, is passed, next to a normal *String* argument, the special variable *result*. As a consequence, mapping *part2columns* will not create a new table, but merely use the result of the *part2table* mapping and update properties as specified in the *part2columns* mapping. The special variable *result* stores the result of the mapping implicitly[8].

[7] In fact, a mapping body that directly assigns properties is merely a syntactic simplification of an in-lined mapping, bound to the result of the mapping. For details, we advise you to consult the specification.

[8] This is again a syntactic convenience. The assignment to *result* can be explicit whenever the mapping defines an explicit object creation. Consult the specification for details on this.

The *part2table* mapping has expressions that refer to the variables *recordName* and *mainColumns*, which don't seem to be declared anywhere. These variables were in fact inherited from the abstract mapping *fieldColumns*:

```
abstract mapping Record::fieldColumns(in prefix : String) : Table {
  init {
    var mainColumns := self.fields->map(f)
                            field2Columns(prefix, self.key = f);
    var recordName := self.name;
  }
}
```

The inheritance semantics of mappings implies that the sub-mapping implicitly *calls* the super-mapping, passing its own result. The call happens before the body of the sub-mapping is executed. In the example, some intermediate results only are calculated. The fact that *fieldColumns* is abstract merely implies that it cannot be called explicitly.

The result of a mapping cannot always be implicitly assigned, as has been the case with our previous sample mappings. This is typically the case when the result of a mapping is a collection. Consider the following example from *alma2db*:

```
mapping PhysicalQuantityType::pqType2Columns (in prefix : String,
                                              in iskey : Boolean)
: Sequence(Column) {
  init {
    result := self.units->map(u)
                object Column {
                  name := prefix + '_as_' + self.name + '_in_' + u;
                  type := self->convertPQType();
                  inKey := iskey;
                };
  }
}
```

The *pqType2Columns* mapping results in a sequence of *Column* objects and the *result* variable is explicitly assigned in the *init-* clause. Also notice that the *map* operation can bind each of the collection elements to a parameter. In the example, each unit from *self.units* is bound to the parameter *u*.

OM transformations also have a means of specifying functions, which do not contribute to the runtime footprint of a transformation. Consider the example:

```
query PrimitiveType::convertPrimitiveType() : String =
  if self.name = "int" then 'INTEGER'
  else if self.name = "float" then 'FLOAT'
  else if self.name = "long" then 'BIGINT'
  else 'DOUBLE'
  endif endif endif;
```

Operational Mappings queries are comparable with functions in QVT relations.

Finally, we have to tell the transformation where and how to start transforming. This is achieved with the main declaration, which in the case of *alma2db* looks like this:

```
main() {
  alma.objectsOfType(Entity)->map entity2table('');
}
```

The operation *objectsOfType(Entity)* simply returns all model elements of metaclass *Entity* from the *alma* input model. The *entity2table* mapping is called on each of these with the prefix bound to the empty string.

This exposition only covered some of the very basic OM language aspects, which are a small (but useful) subset of entire OM language feature collection. It is beyond the scope of this book to delve into the details of OM's numerous possibilities. Some language features are purely for convenience or address scalability issues (for example library usage). Others allow the user to inspect the transformation trace.

One of the more curious OM language properties is the support for integrating it more tightly with a QVT Relations engine. For example, it is possible to include *when* and *where* clauses within mapping definitions. In this case, a transformation writer has to be aware that a mapping invocation may fail and return the null value when the *when* clause evaluates to false. Such *when* and *where* clauses function as pre- and post-conditions respectively.

The entire *alma2db* OM language example is given in Appendix A.2.

10.5 The OMG Standardization Process and Tool Availability

As we mentioned at the beginning of this chapter, the QVT specification as it is currently adopted by the OMG Architecture Board is not yet finalized. This means that the current specification is open to the public for *issue reporting* to allow participants from inside and outside the OMG to point out potential problems and errors, or suggest small improvements. The deadline for issue reporting is 20th March 2006.

Subsequently, a small group of OMG members involved in the QVT language definition will process all the reported issues and either change the adopted specification, or explain why specific issues do not have to be addressed. This process leads to a finalization report, which is then presented to the Architecture Board. In the case of QVT, this report is due by 7th July 2006.

If the Architecture Board is happy with the finalization report, they will issue a recommendation to the OMG Board of Directors, which usually follows the Architecture Board's recommendation to publish the specification. In the case of QVT, this may happen by the end of 2006.

In the meantime, however, tool developers are free to use the adopted specification and provide the MDA community with prototype implementations. Unfortunately, at the time of writing, hardly any prototypes for a QVT language are publicly available. TCS[9] has promised a public non-open-source prototype for QVT Relations by early 2006. France Telecom[10] expects a prototype for the QVT Operational Mappings language by April 2006. Borland has already announced a QVT-based model-to-model language prototype implementation in its Together Architect 2006

[9] TCS (Tata Consulting Services) is one of the main QVT Relations language contributors.

[10] France Telecom leads a group of companies committed to the QVT Operational Mappings language.

product. This QVT prototype is essentially a small subset of the QVT Operational Mappings language with small deviations from the adopted standard.

In the near future we expect an increased number of available QVT tools, both commercial and Open Source. However, you should be aware that the OMG is not very strict about standards compliance. In fact, the OMG has no official way of verifying the level of compliance for a tool[11]. In the case of QVT, this will almost certainly lead to a large number of vendors claiming support for QVT without really being standards-compliant.

It is also interesting to consider what it means to be QVT-compliant. According to the specification, a tool ought to indicate which of the three QVT languages it supports. For each supported language, a tool has to indicate whether it can import or export a specification either in its abstract syntax (for example via XMI) or its concrete syntax. Moreover, the ability to use black-box rules has to be explicitly stated as well. Clearly, this leads to a large number of possible ways of being QVT-compliant.

10.6 Assessment

Whether the QVT standard as it stands today will survive the turbulent and fast evolution of new techniques and methodologies within MDSD, only time will tell. The fact that the standard specifies three different languages indicates that model-to-model transformations remain an ill-understood domain. There is also a multitude of non-QVT model-to-model transformation languages and tools available, some proprietary, some Open Source, but all tackling the problem in their own way.

Obviously, the intentions of the QVT standard were to avoid wild growth of such systems, but the problem is that nobody knows today what is really required to make M2M work in practice. This is partly because few projects have used M2M on a large scale in industrial project settings, and partly because model-driven development is an ‘early’ market.

The M2M requirements in Section 10.2 only scratch the surface of what is really required for an M2M transformation language to become practical and usable. The difficulty of developing a new M2M language is to find the balance between a usable language for the transformation writer and the possibility of properly implementing the language reasonably efficiently and well-integrated in an MDA tool landscape. While a standard should not specify how to build tools for the language, it should define a language that can be reasonably implemented. In the case of QVT, it should not be forgotten that a transformation works on models that ‘live’ in some kind of MDSD repository environment. Users of the transformation expect smooth integration of a QVT engine into such an environment, as the transformation is only a means, not an end, after all.

But even with the most basic requirements from Section 10.2, it is not clear how the current QVT languages address them all. For example, QVT Relations has no way of specifying a retainment policy: as QVT Operational Mappings is an imperative language, it is not clear at all how an incremental update mechanism can be supported, as its impact analysis would be horrendously complex.

In terms of language use, it seems that both QVT Relations and QVT Core have paid a high price for wanting to support bidirectional mappings, something that we believe has no practical

[11] Standard compliance could be verified with a test suite or a reference implementation, or, as is the case with J2EE, with both.

value. Moreover, these two declarative languages do not have mechanisms for working with exception conditions, which is an important omission. The QVT Operational Mappings language was only touched on in this chapter, but the full language is fantastically complex: it almost seems that object orientation here has been driven too far to remain usable. An M2M transformation is an intrinsically functional problem that does not seem to integrate easily into the object-oriented paradigm.

We also believe that M2M transformation development will only be able to take off when M2M development environments have become available and are as good as modern programming language IDEs. However, this requires sophisticated tools with intelligent editors and advanced debugging facilities. Considering the fact that at the time of writing hardly any QVT tools are on the market in the first place, we are not anticipating QVT transformation writing to become a mainstream activity in the near future. Moreover, the lack of one standardized QVT language is a further hindrance for the acceptance and usage of QVT-style M2M transformation writing.

Finally, it should be mentioned that the QVT specification is a complex document that underwent countless revisions. It is difficult, if not impossible, to verify whether the language standard is consistent and sound. This can partly be explained by the fact that the OMG has little experience in language definitions with a behavioral bias. Another reason is probably that too many people were involved in its specification over a long period of time. However, a sound language standard is a must for widespread acceptance and easy implementation. We anticipate that revisions and follow-up versions of QVT will appear in the near future, provided the standard is not bypassed by the possible emergence of a defacto model-to-model transformation standard.

11 MDSD Tools: Roles, Architecture, Selection Criteria, and Pointers

In this chapter we want to address important properties of generic MDSD and MDA tools more closely[1]. Such properties can be used as selection criteria for tools. However, for obvious reasons, we cannot recommend any commercial tools. The OMG offers a list of MDA tools [OMGT], but not all of the tools listed there meet the requirements we discuss in this chapter. One reason is that the tools on the list are merely registered by the manufacturers and not certified by any kind of authority such as the OMG.

This chapter is divided into three parts. Section 11.1 describes the kinds of tools that play a role in MDSD, while Section 11.2 describes some MDSD tool foundations. The characteristics described there should be considered when selecting and implementing MDSD tools, or when building tool chains. Section 11.3 finally points to a number of specific tools, concepts, and ideas that might be interesting starting points in the context of MDSD tool selection.

11.1 The Role of Tools in the Development Process

Model-Driven Software Development doesn't make sense without tool support. This section provides a brief overview of the typical tool categories that should be used in the context of an MDSD project or a domain architecture.

11.1.1 Modeling

The central tool is the modeling tool. Depending on the DSL, different concrete tools can be used in this category. Nevertheless, one should try to provide a suitable editor for a specific DSL – an editor that 'knows' the DSL and effectively supports it during modeling. For example, you can use text editors that support a textual DSL through syntax highlighting and code completion. Form-based editors are often useful too. For graphical DSLs, graphical editors are the tool of choice, yet the effort required to create them can be considerable.

[1] 'Real' MDA tools are also MDSD tools, but not every MDSD tool is also an MDA tool. This is due to the MDA's focus on UML or MOF. This distinction is not essential to our examination. however.

If the models are created using UML profiles, a suitable UML tool is needed. In principle, you can work with any UML tool, but there are some recommendations you should observe. The following list begins with the highest priority argument and moves to the lowest.

- *XMI export*. First and foremost, the tool must be able to export the model in XMI format. Most MDA/MDSD generators can accept XMI as an input format, amongst others. XMI still offers many degrees of freedom, and thus possesses a certain level of ambiguity. However, a model represented as an XMI file is a good starting point for the further processing of models.
- *Stereotypes and tagged values*. The annotation of stereotypes on model elements is feasible in practice with every modeling tool and cannot therefore be used as a basis for comparison. When dealing with tagged values, this cannot be taken for granted. Good UML tools treat stereotypes as model elements, not just as ‘strings on a class«. You need to be able to define to which modeling elements (that is, elements of the UML metamodel) a certain stereotype should be assigned. Moreover, it should be possible to ensure that a specific stereotype also requires or allows specific tagged values. The tool should force the developer to provide consistent input during modeling.
- *Metamodeling/profiles*. A tool should ideally supports real metamodeling and the creation of UML profiles. This includes the definition of individual metatypes as well as the definition of constraints in the metamodel. Ideally, the graphical rendering of the new model elements (that is, their concrete syntax) and the tool’s GUI should also be adaptable. For example, a button for inserting a custom model element type should be configurable. Such tools are rare at present, but they do exist [GME], [MC04].
- *OCL*. It is useful if the tool not only allows OCL expressions at the M1 level (see Chapter 6) as comments, but also checks the constraints syntactically regarding the model. In consequence, a constraint is only allowed if it constitutes a valid expression in terms of the current model.

11.1.2 Model Validation and Code Generation

The vast majority of UML tools currently available are unable to check a model for correctness with a domain-specific metamodel. Yet code generation can only function if the model is correct with respect to the metamodel. Also, in the case where a model is not modeled with UML (+ profiles), a check of the model must take place prior to code generation.

In most cases, this check of the model is conducted with a separate tool – the *generator*. It is important that the validation phase in this tool is separate from the code generation phase. The transformations should not have to deal with the validation of the model, but assume correctness of the model regarding the metamodel, otherwise the transformations will become unnecessarily complicated. Moreover, the validation of a model is completely independent of the question of what is generated from it — that is, of the transformations. Validation is exclusively a question of the domain for which the model was built. If various transformations are executed on the same model, one would need to integrate the problem-domain-related correctness constraints with each transformation, which is not very feasible.

Several aspects are essential to the validation and transformation phase of the generator, here listed in descending priority:

- *Abstraction from the concrete syntax*. The validation of the models should always take place independently of the concrete syntax in which the model is represented. Such an approach allows changes to the concrete syntax without having to adapt the validation rules. A new parser that is able to read the altered, concrete syntax must be provided. We explain this point in more detail later. The abstraction from the DSL's concrete syntax ideally takes place via an explicit representation of the metamodel.
- *Explicit representation of the metamodel*. It is essential for both validation as well as for transformation of models that the metamodel underlying the model that is currently processed has an explicit representation in the generator. This requirement implies that the developer can adapt the metamodel used by the generator. A tried and trusted solution of this problem is for example that the metamodel elements are represented as classes of a programming language. The model then consists of objects, that is, instances of the respective metamodel elements (see Section 9.3.2). We address this point later as well.
- *Declarative constraints*. The metamodel constraints that verify a model's correctness should ideally be definable in a declarative manner. OCL is one example of such a declarative constraint language, but it is not as yet fully supported by most tools. Even if this ideal cannot be achieved, one should try to formulate constraints as declaratively as possible. We explain such a pragmatic approach in detail in the context of this book's second comprehensive case study in Chapter 16.
- *Workflow control*. Developers should be able to control the sequence of steps – instantiation, validation, transformation, code generation – for building non-trivial scenarios.

Based on the Open Source generator openArchitectureWare [OAW], all these properties can be studied. Figure 11.1 shows the workflow of a non-trivial example, including two cascaded domain architectures. Initially, the application model is loaded. It is built based on the first domain architecture's DSL. The parser builds the AST representation, the instantiated metamodel, which is an instance of the metaclasses that represent the first domain architecture's metamodel. Next, the constraints of this metamodel are checked. If violations – failed constraints – are detected, the workflow ends here and the errors are reported. If everything is ok, the model-to-model transformations included in the first domain architecture are executed, creating an instance of the second domain architecture's metamodel. Again, constraints are checked. If this check succeeds, the model-to-code transformations of the second domain architecture are executed, resulting in generated code that now has to be integrated with handwritten code snippets.

11.1.3 Build Tool

In many cases a large number of different artifacts are created by the generator. In the next step, these must be suitably compiled, packed, and processed. A suitable build tool is required for this. In principle any scripting language can serve this purpose: Ant has become widely accepted in the Java world.

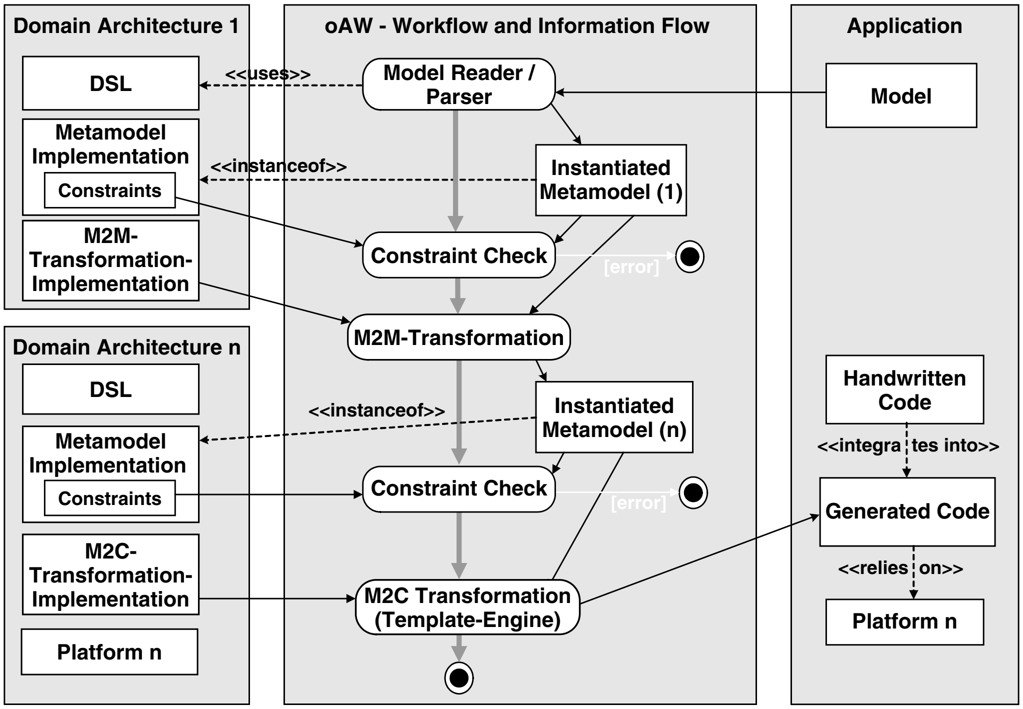


Figure 11.1 Internal workings of openArchitectureWare

In many cases, one would generate the build script as part of the generator run, then execute it directly.

11.1.4 Recipe Frameworks

If you require your developers to write specific aspects of the system manually, and if specific rules must be follows when doing so (such as 'you have to extend from the generated base class and implement the abstract methods') then it is useful to guide the developer. The generator can't really help, since it will just generate the base classes and then terminate, hoping that the developer 'does the right thing' and writes the subclass.

Recipe frameworks manage a developer's implementation tasks after running the code generator. Typically, the generator, in addition to generating the code, also instantiates a number of checks that are subsequently checked by the IDE against the generated code to verify that the handwritten code is complete and correct. Such a tool can make development using the generator significantly simpler. For an example of this approach, see Section 17.4.3.

11.1.5 IDE Toolkit

An ‘IDE toolkit’ is not mandatory, but can be very useful if available The idea is that the models, the different process steps, and the generated artifacts can be manipulated and managed by the application developer in a single, customized IDE. Adapted editors for configuration files, access to the model, as well as access to generated artifacts, should all be available. In particular, the domain-specific IDE should only display those artifacts or aspects that are relevant to the developer. Irrelevant intermediate results should only be shown if needed.

The *Eclipse* platform is an example of one such IDE toolkit. It allows the creation of project- or domain-specific IDEs with acceptable effort – see for example [RV05].

11.2 Tool Architecture and Selection Criteria

11.2.1 Implement the Metamodel

A formally-defined metamodel for a domain is a good starting point, but to be really useful, it must be actually used during application development. As long as the metamodel is only documented on paper or in a modeling tool, without further tools using it, it has no productive value.

Manual checking of models for consistency with the underlying metamodel is time-consuming and error-prone. Standard modeling tools such as current UML tools are usually of no use here, because they do not ‘understand’ the rules of the domain-specific metamodel, and therefore cannot use them for checking the models. The only rules applied by a typical UML tool are the rules checking the well-formedness of UML models in general – that is, a check against the UML metamodel. In this regard we can only hope that UML tools will offer improved support for profiles in the near future.

You should therefore implement the metamodel in a tool that can read models and check them for correctness against the metamodel. The correctness checks must cover all rules and constraints that the metamodel prescribes – this is the only way to ensure sensible subsequent model transformations and code generation.

Checking a generator’s input data is one practical application of the metamodel. This added value is an essential part of both the MDSD process and the domain architecture. The implementation of the metamodel itself can of course be achieved using model-driven techniques, such as using a corresponding meta-domain architecture.

11.2.2 Ignore the Concrete Syntax

Each model must inevitably be rendered in a concrete syntax, for example UML XMI for MOF-based models, and XML for textual models. Nevertheless, transformations that are defined based on the concrete syntax are rather unyielding, because they must consider the concrete syntax, whereas the transformation of instances of the metamodel elements should be their priority. This makes transformations unnecessarily complicated. Furthermore, the transformations can no longer be used if the concrete syntax of the models is changed – which does indeed happen from time to time in the course of a project. How can one ensure that the transformations and the model's validation do not depend on the concrete syntax?

The definition of transformations on the basis of concrete syntax is typically error-prone and inefficient. XMI, for example, has a very complicated syntax. To define transformations via XSLT on this basis is practicable only for trivial cases. In many cases, it is also useful to have several concrete syntaxes for the same metamodel, for example if different DSLs are used for the description of various technical subdomains or if the concrete syntax is changed in the course of a project. Definitions of transformations and model validations based on concrete syntax unnecessarily bind the transformation to a specific concrete syntax.

The transformation definitions should thus be based on the source metamodel (and the target metamodel for model-to-model transformations). For this purpose, implement a three-step approach in the transformation tool:

- First, the source model is parsed and an abstract representation of the model is created in the generator, typically in the form of an object structure, for example through instantiation of the metamodel classes.
- This model is then transformed into the target model, working only on the abstract object-graph representations.
- Finally, the target model is rendered into the concrete syntax of the target DSL.

This approach allows a significantly more efficient and productive method of specifying transformations. It also makes the transformer considerably more flexible, because it can now work with any concrete syntax. This is especially important for XMI-based concrete syntaxes, because different UML tools export different XMI dialects. You should avoid binding your transformations to a certain tool or even to a certain tool version.

The approach described in this section has been practiced in compiler construction for some time. Compilers always work in several phases, and the implementations of these phases are often exchangeable. Compilers can therefore be adapted with relative ease to different target platforms, or to 'understand' different source languages. Figure 11.2 explains the principle for code generators.

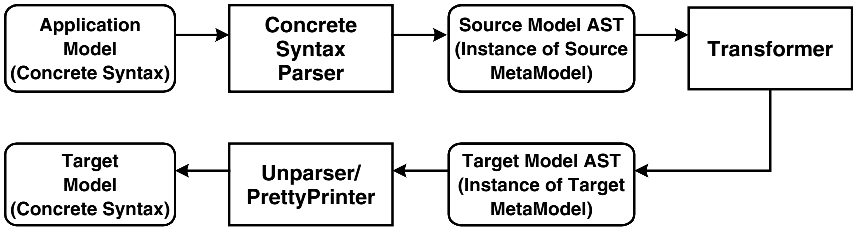
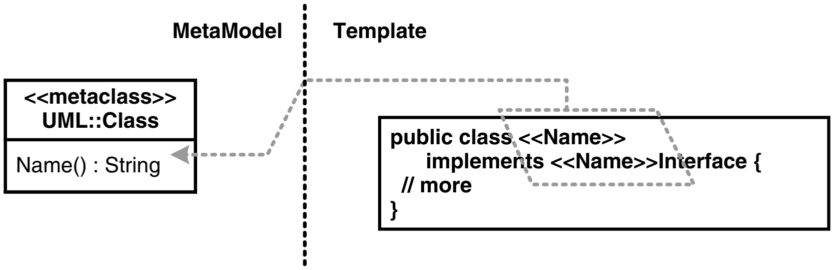

Application Model (Concrete Syntax)
Concrete Syntax Parser
Source Model AST (Instance of Source MetaModel)
Transformer
Target Model AST (Instance of Target MetaModel)
Unparser/ PrettyPrinter
Target Model (Concrete Syntax)

Figure 11.2 A typical AST-based generator's mode of operation

MDSD code generators often don't use the complete three-step approach, but generate textual output directly based on the original model instance. It would be too complicated to create an abstract syntax tree of the target model, which would be the target language's abstract syntax in this case. Instead, template languages are used to navigate over the source model. However, the source model should be represented as an object graph, an instantiated metamodel.

This approach is well-suited for implementing the metamodel. Ideally, the same implementation is used for both purposes. The templates can the work directly with the meta-objects, and the meta-objects' properties can be used to provide information for template evaluation, as is shown in Figure 11.3. (See also Chapters 3 and 16.)



```
public class <<Name>>
     implements <<Name>>Interface {
  // more
}
```

Figure 11.3 Access to a model AST from the templates according to [OAW]

11.2.3 Modular Transformations

We have already described the usefulness of modular transformations from the domain architecture's perspective. From the MDSD tools' perspective, the support of this concept constitutes a construction feature or a selection criterion.

11.2.4 Model Transformations are 'First-Class Citizens'

As we have seen, model transformations are not a minor matter in domain architectures, but fully-fledged, essential artifacts, exactly like models and manually-created source code. This affects how developers deal with the transformations, as well as on how the tools do:

- Developers should structure sensibly, modularize, and refactor frequently.
- The transformation languages must provide sufficient means for structuring the transformations, for example by using modularization, delegation, inheritance and polymorphism, as well as aspect-orientation.
- Tools must store the transformations in such a way that they can be versioned and managed with the tools used for those purposes in the project, if applicable in distributed teams.

Some of today's tools still treat transformations as being 'secondary' compared to models and traditional source code. Fortunately, template-based generators usually store code generation templates as separate (text) files that can be versioned easily, for example with CVS, and if necessary merged. It should be mentioned, however, that there is no such thing as a proven and generally-accepted paradigm or syntax for the representation and handling of transformations.

11.3 Pointers

In this section we want to point to a number of interesting tools and frameworks, most of them Open Source. Some of these tools are available today, others will become available in the near future.

11.3.1 The Eclipse World

Many interesting MDSD-related technologies are emerging in the context of the Eclipse platform. This section points out some of the most important.

Eclipse Modelling Framework (EMF)

EMF is a framework for MDSD based on Eclipse, and serves as the basis for a great number of interesting tools. Its primary building block is eCore, a meta meta model implementation that is aligned closely with the eMOF (*essential MOF*, see Section 12.2.2). EMF allows the definition of metamodels using various techniques such as tree-based editors, programming, and so on. You can then generate implementation classes from these metamodels that provide a concrete API for building instances of the metamodel. These generated classes also provide runtime access to the metadata that cannot be represented directly with Java classes.

EMF also comes with a couple of additional generators that generate editors and a generic editing framework for editing the models.

The importance of EMF comes from that fact that it is quickly evolving into the de-facto industry standard onto which MDSD tools are built. A rich and vibrant community of experimental tools has developed around it.

The most important parts of the eCore meta meta model are shown in Figure 11.4. In addition to the aspects rendered in that diagram, eCore contains the following elements:

- A number of additional (derived) associations
- Operations and exceptions
- Enumerations
- The usual primitive data types
- Packages and factories (required to instantiate the model elements)

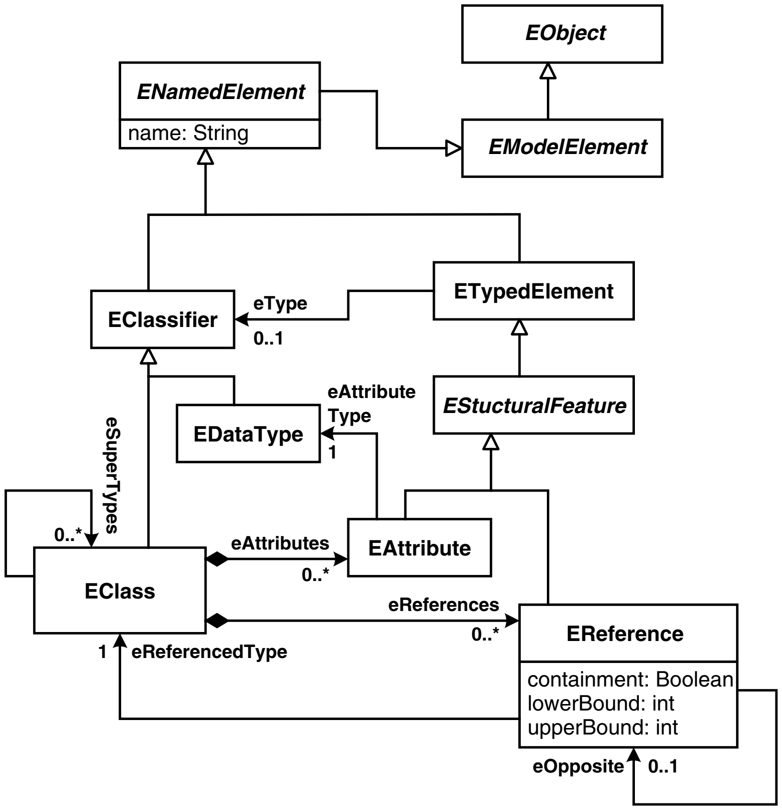


Figure 11.4 The essential parts of eCore

Graphical Modelling Framework (GMF)

GMF is one of the frameworks based on EMF. It supports the automatic generation of graphical editors for EMF meta models. To get a graphical editor for your modeling language, you:

- Define the metamodel using EMF.
- Define an additional model that describes to the GMF generator how you want the generated editor to look like and behave.
- Generate your editor.
- Possibly add some specific behavior or graphics using manual coding.

The process is illustrated in Figure 11.5. The generated editors are based on GEF, the Eclipse Graphical Editing Framework. As such, the generated editors integrate nicely with the Eclipse platform.

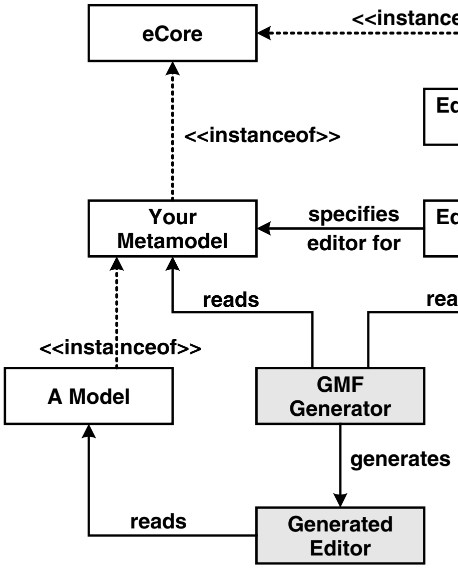


Figure 11.5 Generating an editor using GMF

This approach to generating editors is not new – it has been used before, albeit with other specific technologies, for example in [RV05]. Again, the significance of GMF is due to the fact that it is based on EMF, which itself has a large group of followers, and that it is strongly supported by industry.

Generative Model Transformer (GMT)

The GMT subproject serves as a container for a number of subprojects that each explore different aspects of MDSD. The projects have different levels of maturity.To briefly point out some of the GMT subprojects:

- ATL is a model-to-model transformation engine that has matured over the past few years and is in widespread use. ATL is tightly integrated into Eclipse: for example, syntax highlighting editors are available. While it provides its own meta meta model (KM3) there is integration with EMF. The following is a simple transformation taken from the ATL documentation to give you a feeling of ATL:

```
module Author2Person;
create OUT: Person from IN: Author; -- Person and Author are metamodels

rule Author {
  from
    a: Author!Author
  to
    p: Person!Person( name <- a.name, surname <- a.surname )
}
```

- AMW is a tool for representing correspondence between a number of models. These correspondences are stored in a separate model, the *weaving model*.
- AM3 is aimed at supporting modeling-in-the-large ('megamodeling'). This involves meta model and meta meta model independent cross-model references.

Model-Driven Development Integration (MDDi)

The MDDi project aims at providing what is called a 'model bus', a facility for inter-tool model interchange. Contrary to file-based interchange, as provided for example by XMI, the primary goal of MDDi is to provide real-time model synchronization. The project is supported by the European Union's ModelWare project. As of December 2005, the public CVS is still empty – the project does not seem to be very active.

11.3.2 Trends in UML tools

UML tools have traditionally been used as the 'entry-level' drawing tool for many MDSD projects. You use class diagrams for basically everything, then use stereotypes and tagged values to add semantics. A more or less standards-compliant XMI export serves as the import format for the generator. However, a couple of interesting improvements have occurred in this area.

For example, EMF is being used as the foundation for Eclipse-based UML tools such as IBM's Rational Software Modeller or Omondo's EclipseUML. If your generator tool is also EMF-based, you can work directly from the live models and don't need to use a file-based means of moving the model into the generator. This is a big improvement, for two reasons: first, annoying XMI incompatibility issues are solved, and second, performance is improved because serialization and deserialization (instantiation) of the model is no longer necessary. Using EMF as a basis for UML tools – with the UML metamodel implemented as an instance in eCore – also allows you to extend the UML metamodel with your own metaclasses, rather than just to use the rather awkward profiling mechanism. We have not yet come across a tool that actually supports this approach, however.

Another interesting development is that more and more UML tools have open APIs that are becoming increasingly powerful. This allows the automation and customization of domain-specific workflows. For example, in MagicDraw you can even define your own custom diagram types.

11.3.3 UML 2 Composite Structure Diagrams

It is worth pointing out a specific new feature of UML 2 that is being adopted in most UML tools: composite structure diagrams. In UML 1.x it was not possible to decompose structures into smaller structures hierarchically – that is, it was not possible to have 'boxes within boxes'. This was a big shortcoming: considering that UML class diagrams have always been (mis-)used as a generic graph-drawing tool – classes were nodes, associations and dependencies were the edges – the ability to 'zoom into' nodes has been missing. This has changed with UML 2. Using composite structures in combination with ports and connectors is powerful, as the following example shows.

The example maps the concepts of a functional domain (power grids) to a UML 2 model, to calculate the cost of transporting electricity from the power generators to energy distribution companies. The DSL is used by engineers to model power grids. Figure 11.6 shows the core metamodel on which the power grid models are based.

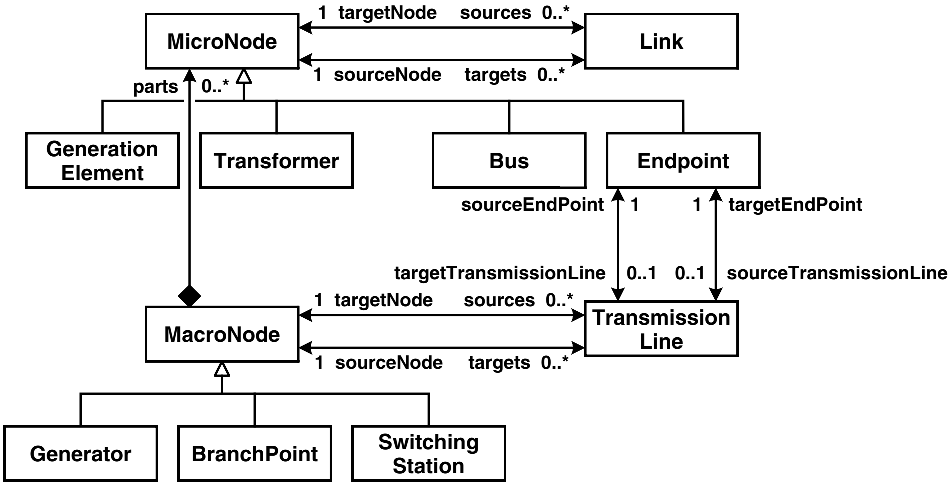


Figure 11.6 Power grid metamodel

The resulting DSL is ideally suited for mapping to UML component structure diagrams:

- All subclasses of *MacroNode* are mapped to instances.
- *EndPoints* are mapped to ports.
- All other subclasses of *MicroNode* are mapped as parts of instances (parts).
- *Links* and *TransmissionLines* are mapped to connectors.

Using this notation, standard UML 2 tools can be used to model power grids graphically. Figures 11.7 and 11.8 show some examples.

To understand why ports are required, it is worth looking at the example from a greater distance, on an abstraction level where the content of *MacroNodes* is irrelevant.

Without ports, the diagram in Figure 11.8 would be imprecise and not a clear abstraction from the diagram in Figure 11.7. Ports offer the option of modeling complex (component) systems in a top-down process and refining them later. As our example shows, this is useful not only for the modeling of software, but also in other domains.

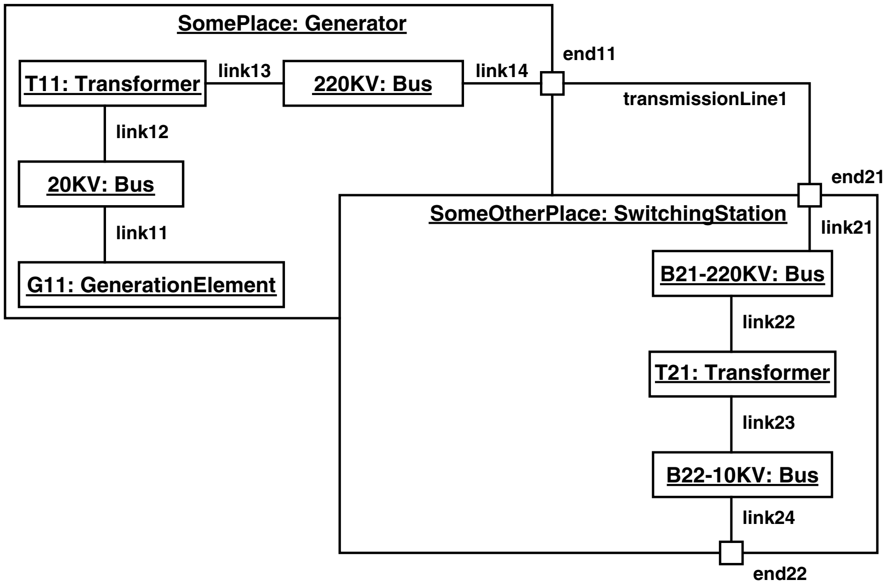


Figure 11.7 Excerpt of a power grid – micro scale

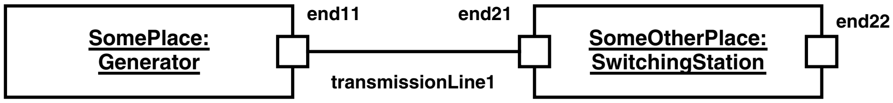


Figure 11.8 Power grid – macro scale

11.3.4 Other Kinds of Editors

There are some tools that can be used as modeling frontends that would not occur to one immediately. A good example is Microsoft Visio. Visio can be considered an 'object-oriented' drawing tool. If you build a drawing using Shapes, the elements of the drawing reference the 'shape type' with what can be considered an *instanceof* relationship. A Visio drawing is not just a collection of graphical elements, therefore, but a collection of *typed* graphical elements and their typed connections. Visio also comes with an editing tool to build you own shape types, effectively defining the metamodel and concrete syntax. Because Visio has been able to store drawings in XML since Version 2002, access to the drawings as well as to the instantiated shapes is simple. It is therefore easy to build, for example, an instantiator frontend for openArchitectureWare that reads Visio models and further processes them.

Feature modeling tools are another interesting alternative for modeling systems, especially if the modelled system is a product (variant) in the context of a software system family. The feature modeling plug-in developed by Krzysztof Czarnecki's team at the University of Waterloo [FMP], or pure-systems' commercial pure::variants [PV] product can be used for this purpose.

11.3.5 Integrated Metamodeling IDEs

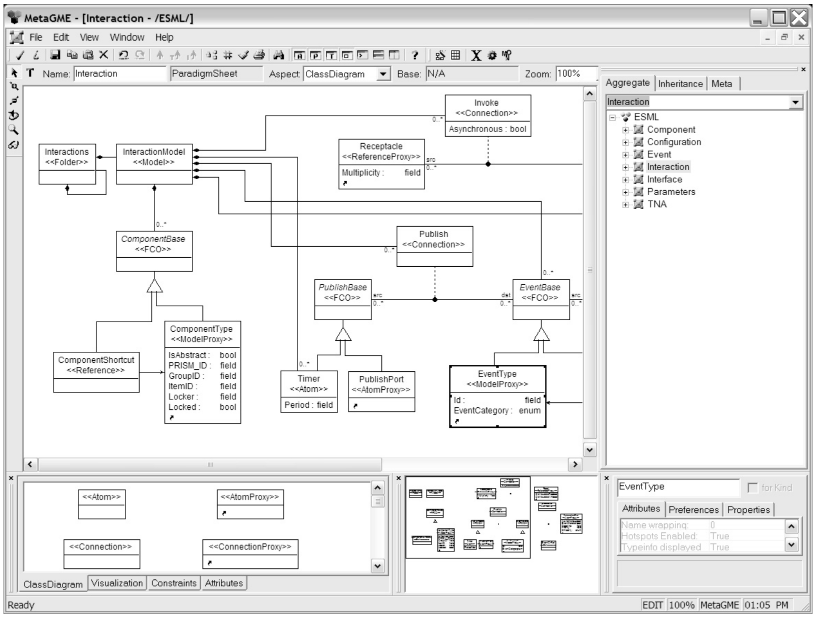
Integrated Metamodeling IDEs are tools that support two distinct tasks:

- They support the definition of metamodels, constraints, concrete syntax, and editors for user-defined DSLs.
- In a second phase, they make the newly-defined DSLs available in the tool, to allow application developers to use the user-defined DSLs[2].

We want to introduce the two most widely-used members of this family of tools briefly. The first is GME [M. Völter, A. Schmid, E. Wolff, Server Component Patterns, John Wiley & Sons, 2002], the Generic Modeling Environment. This is an Open Source tool developed by Vanderbilt University's Institute for Software Integrated Systems (ISIS). The tool has been developed and is used mostly in the context of Model-Integrated Computing (MIC) projects in the industrial and defense worlds. It provides access to models via COM and Java interfaces. Figure 11.9 shows the metamodeling facilities of GME, while Figure 11.10 shows an editor based on the previously-defined metamodel.

The other example is a commercial tool built by MetaCase called MetaEdit+ [MC04]. It uses the following five concepts as its meta meta model: graph, object, relationship, role, and property. It provides graphical as well as table-based editors. MetaEdit+ also comes with a code-generation facility. As can be seen from MetaCase's reference page, the tool is in relatively widespread use. Figure 11.11 shows how meta model elements. as well as their graphical representation. are defined. Figure 11.12 shows a state-machine-based definition of the behavior of a stop watch.

[2] Technically, these tools usually consider the metamodeling and editor-construction part as 'just another DSL' – the tools are usually self-bootstrapping.



MetaGME - [Interaction - /ESML/]

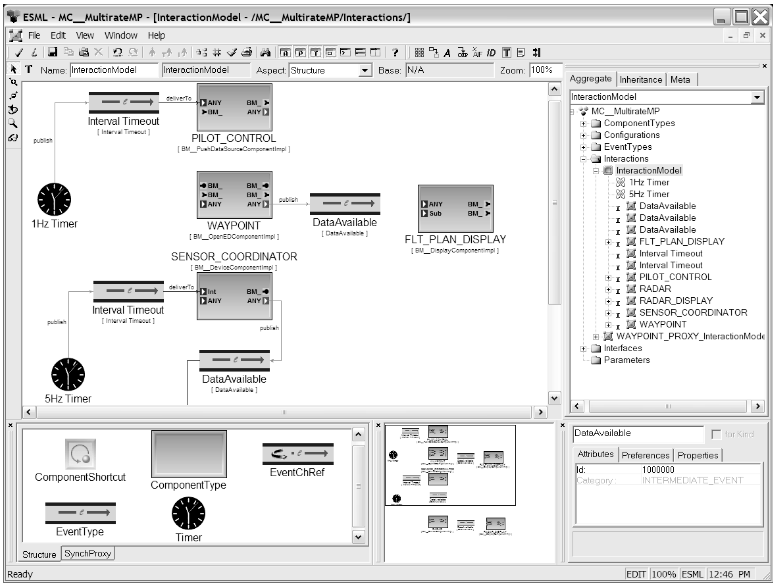
Name: Interaction ParadigmSheet Aspect: ClassDiagram Base: N/A Zoom: 100%

EventType

Name wrapping: 0
Hotspots Enabled: True
TypeInfo displayed: True

Ready EDIT 100% MetaGME 01:05 PM

Figure 11.9 Screenshot of GME, during metamodeling



ESML - MC__MultirateMP - [InteractionModel - /MC__MultirateMP/Interactions/]

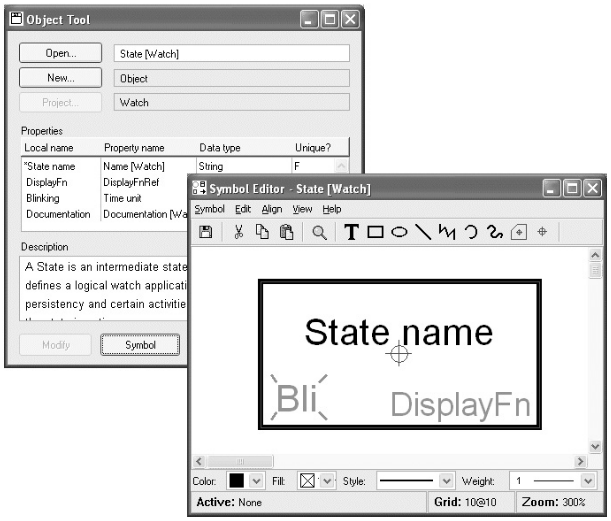
Name: InteractionModel InteractionModel Aspect: Structure Base: N/A Zoom: 100%

DataAvailable

Id: 1000000
Category: INTERMEDIATE_EVENT

Ready EDIT 100% ESML 12:46 PM

Figure 11.10 Another screenshot of GME, now modeling based on the metamodel defined before



Object Tool

Open... State [Watch]

New... Object

Project... Watch

Properties

Local name	Property name	Data type	Unique?
*State name	Name [Watch]	String	F
DisplayFn	DisplayFnRef		
Blinking	Time unit		
Documentation	Documentation [Wa		

Description

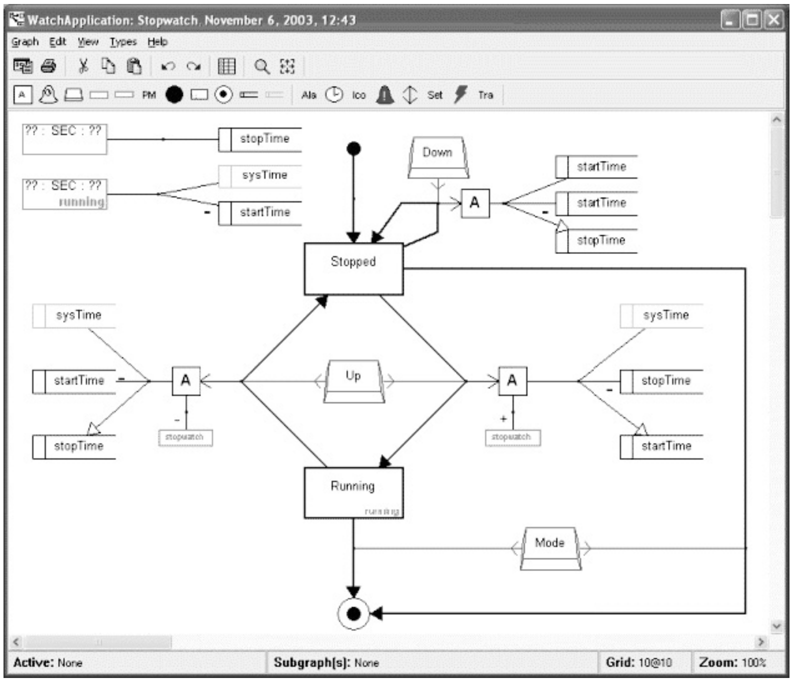
A State is an intermediate state
defines a logical watch applicati
persistency and certain activitie

Symbol Editor - State [Watch]

Color: Fill: Style: Weight: 1

Active: None Grid: 10@10 Zoom: 300%

Figure 11.11 Defining a metamodel element and its associated symbol



WatchApplication: Stopwatch, November 6, 2003, 12:43

Active: None Subgraph(s): None Grid: 10@10 Zoom: 100%

Figure 11.12 Another screenshot of MetaEdit+, modeling the behavior of a stop watch

12 The MDA Standard

This chapter describes important aspects of MDA, although we do not address all details here, because external literature [Fra02], as well as other resources and the standard itself are available.

12.1 Goals

Model Driven Architecture (MDA) is a term with several different meanings. In the context of this chapter, when we speak of MDA we mean the standardization initiative of the OMG in respect to MDSD. Since MDA does not yet cover the whole MDSD spectrum, one can also think of it as a specific flavor of MDSD.

MDA is a young standard established by the Object Management Group [OMG]. The OMG was founded in 1989 and is an open consortium currently of about 800 companies worldwide. The OMG creates manufacturer-independent specifications to improve the interoperability and portability of software systems. Traditionally the OMG is a platform for middleware and tool manufacturers, serving the synchronization and standardization of their fields of activity. CORBA (Common Object Request Broker Architecture) and IDL, UML (Unified Modeling Language), MOF (Meta Object Facility), and XMI are popular results of this process. MDA is the OMG's new flagship.

According to the OMG's directive, the two primary motivations for MDA are the *interoperability* (independence from manufacturers through standardization) and *portability* (platform independence) of software systems – the same motivations that resulted in the development of CORBA. In addition, the OMG postulates that the system functionality specification should be separated from the implementation of its functionality on any given platform. From this perspective, the MDA pursues the goal of providing guidelines and standards that will lead to a respective structuring of specifications in the form of models. Last but not least, this approach promises improved *maintainability* of software systems through a *separation of concerns* as well as *manageability of technological changes*.

12.2 Core Concepts

This section looks at the core building blocks of MDA. These are UML 2.0, the Meta Object Facility, XML Metadata Interchange, the three kinds of models (PIM/PSM/PDM), Multi-Stage Transformations, Action Languages, the various core models, model marking, and Executable UML.

12.2.1 UML 2.0

From the MDA perspective UML is central, because many tools are or will be based on UML and profiles. To ensure that this will actually work, the OMG has recently made a few adaptations in the context of UML 2.0 that we introduce briefly here[1].

- *Infrastructure*. Internally, the UML is no longer loosely based on the MOF: the complete UML standards document contains definitions of UML language constructs (that is, of the metamodel) via MOF models. UML is defined formally. This is – as should be clear now – a prerequisite for MDSD, particularly for model transformation and code generation. The OCL also now uses the MOF as its meta meta model: it was necessary to extend the MOF to this end. UML and OCL are now based on the same meta meta model. This makes them conceptually compatible at their cores.
- *Extension, profiles, stereotypes*. The definition and in part also the notation of profiles and stereotypes – that is, the UML's native extension mechanism – have been reworked. We will not address this issue here, since it has already been explained in Chapter 6.

Even though formally all MOF-based models can be used in the context of MDA, the UML and corresponding profiles for modeling in the MDA field primarily will prevail, as can be seen from the *core models* mentioned below. In [Fra02] David Frankel discusses the advantages and disadvantages of the UML in this context. Let’s first look at the advantages:

- Separation of concrete and abstract syntax
- Extensible (via profiles)
- Platform-independent
- Standardized
- Pre- and post-conditions and invariants are possible via OCL (design by contract[2])

On the other hand, numerous disadvantages still exist that mainly apply to UML 1.x. Much has been improved with the introduction of UML 2:

- UML is big and badly partitioned
- There is no conceptual support of viewpoints
- Components and patterns receive only little support (improved in UML 2)
- The Relationship model is vague
- It suffers from limited expressiveness of profiles, or generally limited means of adaptation of the metamodel
- UML and MOF are not (yet) correctly fine-tuned to each other (improved in UML 2)
- Diagram interoperability is missing (improved in UML 2)
- There is no abstract syntax for OCL (improved in UML 2)

[1] Much more than we describe here has actually been added. Here we only focus on those aspects that are directly relevant to MDA.

[2] *Design by contract* is well-known term in computer science that describes the idea that operations define what they expect to be true when called, and that they define what they guarantee to be true after their execution.

12.2.2 MOF – The Meta Object Facility

The Meta Object Facility (MOF, see Chapter 6) constitutes the core of the MDA. It describes the meta meta model on which MDA-conformant tools are based – or should be based. The definition of custom DSLs or metamodels should use the mechanisms of the MOF, or extend UML via profiles, which in effect constitutes a 'lightweight' metamodel extension. As we have already pointed out, UML is defined via the MOF. The MOF itself uses UML's concrete syntax, which can cause confusion[3]. We have implicitly used the MOF's mechanisms many times in this book. Whenever we extended the UML metamodel, we automatically used the MOF.

The MOF is not only important as a formal basis of metamodels, it is also of concrete relevance for the construction of MDA tools such as repositories, modeling tools, code generators, and so on. Generic tools need a solid basis: this can only be the meta meta model. Similarly, to guarantee the portability of data used in tools, one must agree on a meta meta model. Thus it is essential for the OMG's standardization efforts and the tool market to define the meta meta model completely, formally, and inherently correctly.

Figure 12.1 shows a part of the MOF.

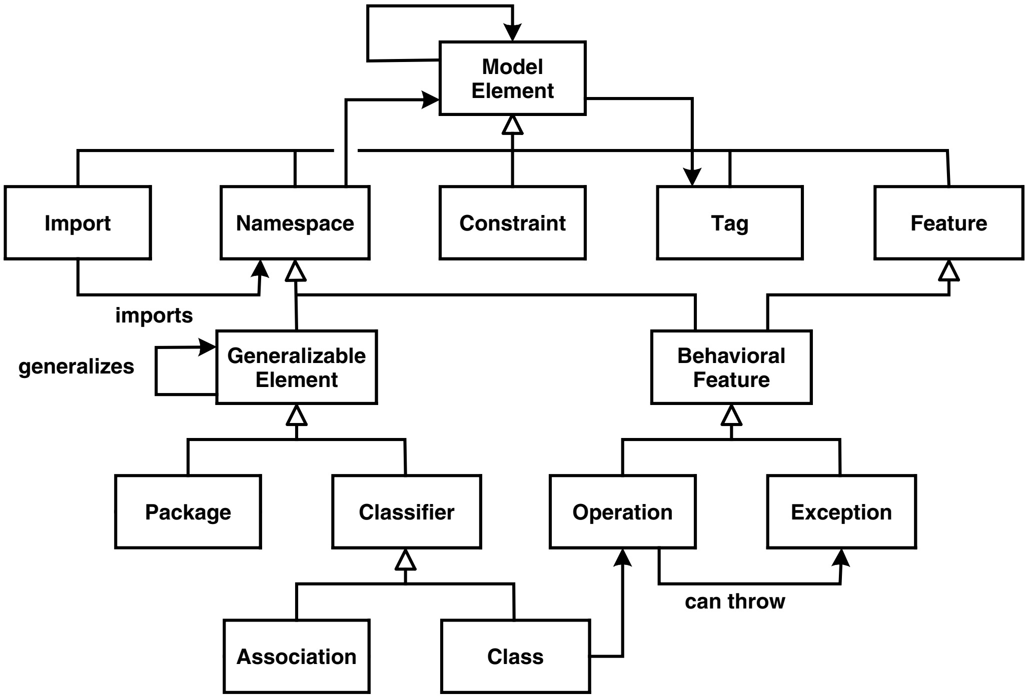


Figure 12.1 A part of the MOF

The MOF also has a few disadvantages. One can argue about whether the following aspects should be part of the MOF or not. For example, the MOF does not offer any help in defining a

[3] More precisely, the core of UML (that is, the *classifier* package) can be applied on all metalevels since the introduction of UML 2.0. Thus it also constitutes the core of the MOF.

concrete syntax for DSLs or for versioning issues, and the composition of metamodels from partial metamodels is not addressed.

Essential MOF and Complete MOF

Several implementations of the MOF have been created. However, they all only implemented a relevant practical subset of the MOF. Perhaps the most influential implementation was EMF, the Eclipse Modelling Framework [EMF] and it’s eCore meta meta model. Implementation of EMF in turn had an influence on the standardization of MOF 2.0. As a consequence of that influence, the OMG identified a subset of MOF called the essential MOF (eMOF) during the standardization of MOF 2.0 in 2003 that would be sufficient for most meta meta model implementations. Consequently, EMF’s eCore is now compliant with the OMG EMOF standard. You can find a more thorough discussion of eCore (and thus EMOF) in the tools chapter (Section 11.3.1).

Consequently the counterpart of the eMOF is the CMOF. It is used for more complex metamodels such as UML. For example, it combines mechanisms to extend packages by importing, merging, or combining them. It also combines more powerful reflective features.

12.2.3 XMI

XMI stands for *XML Metadata Interchange* and is an XML mapping for MOF – not just a DTD/schema for UML, as it is often incorrectly stated[4]. Currently, XMI is the basis for interoperability between different MDA tools because (real, database-based) MOF repositories are yet not widely in use. Since the release of version 2.0, XMI also allows the serialization of diagram layout information, which is mandatory for a practical and useful model exchange between modeling tools, not only for code generation, if you don't want to rely on the auto-layout mechanisms of the established tools, which are usually poor.

At present there are still many incompatibilities between the XMI formats of different tools, which complicates diagram exchange between modeling tools. However, the use of XMI as a basis for code generators is not a serious problem in practice, since all popular generators support parsers for the various XMI dialects.

There are two flavors of XMI. The first is a completely generic one that can store all MOF-based models via a generic DTD. The document structure is defined at the MOF level, causing the XMI documents to become rather verbose. A positive fact is that it can be generically applied to all MOF-based models. However, its ability to be read by humans or its suitability for XSLT-based transformations is rather limited.

The XMI standard, however, encompasses the option of generating a DTD or a schema specifically for a given metamodel based on the MOF. As a consequence, documents are only able to store instances of this particular metamodel, but the resulting file is more compact and concrete, because the structures are mapped at the metamodel level and not at that of the meta meta model. For obvious reasons, most tools use the first approach for exchange.

Note that a simplified XMI mapping (XMI 2.1) was defined as part of the MOF 2.0 EMOF definition. This should provide better interoperability between tools in future.

[4] Typically XMI is specifically used for the serialization of UML models.

12.2.4 PIM/PSM/PDM

The OMG has a concrete concept of what the MDA should represent. The OMG is not concerned with software system families, the involvement of domain experts, or increased agility in software development, but mainly with platform independence of the application logic. Due to the fact that technological solutions – as well as the business logic – continue to develop quickly yet independently of each other, it is pivotal for reasons of longevity to be able to specify the application logic independently of an implementation platform: in other words, to be able to specify its essence. For this purpose, the OMG considers MOF- or UML-based modeling to be the best solution, because it allows the fully-automated generation of implementations for different platforms via transformers. The platform-independent model (PIM) plays a central role here in describing the business logic undiluted by technical concerns.

A platform-specific model (PSM) is created from the PIM via model transformation. The PSM is, as its name indicates, platform-specific for J2EE, .NET, or other implementation platforms. Further transformations can create increasingly specific models, until eventually the source code for a platform is generated, which is turned into an executable artifact via compilation and packaging.

This discussion emphasizes that both PIM and PSM are relative concepts. A PSM may be specific for J2EE, but still independent of a specific application server. It thus constitutes a PIM in respect to the concrete application server platform.

Figure 12.2 shows a taxonomy of the models that play a central role in MDA. (See also [Fra02]).

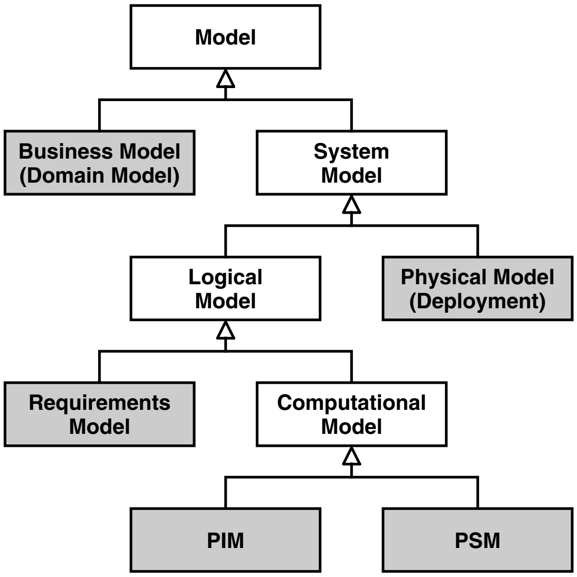


Figure 12.2 Classification of models in the context of the MDA

A further important type of model exists in the MDA, the platform description model (PDM). This is the metamodel of the target platform. Due to the fact that model transformations are always defined as transformation rules between metamodels, it is essential to define the target platform via a metamodel as well. The source models use a domain-specific metamodel anyway – possibly a standardized core model (see below). In the context of architecture-centric MDSD, modeling builds directly on the PDM, the architecture metamodel of the target platform.

12.2.5 Multi-stage Transformations

In the examples and best practices described in this book, we have in most cases generated the source code for a certain platform from (PIM) models. We did this mainly for pragmatic reasons: model transformation tools that support multi-stage transformations for large systems in a manner that is suitable for everyday use do not yet exist.

The MDA pursues the goal of obtaining source code via several subsequent model-to-model transformations. This clearly has advantages given that the right tools are available. The example in Section 11.2.3 shows why, and also clarifies why one tries to do as much as possible at the model level in MDA and to leave the information in model form as long as possible. The transformation engines are defined based on the MOF. As long as we deal with MOF-based metamodels such tools can be used, but as soon as we enter the 'lowlands' of programming, they aren't much use. One would have to define classical programming languages via the MOF, with the result that the transformations would become very complex compared to simple templates.

In some cases it is necessary to configure the intermediate products manually to control their further transformation stages. The OMG calls such a configuration *model markings*. Model markings cannot be annotated directly in the PIM, because this would involve the risk of losing platform independence. For consistency reasons, it is also critical not to modify the intermediate models. We suggest the use of external model markings, as explained in Section 8.3.5.

12.2.6 Action Languages

At some point the justifiable objection that today you cannot model a *complete* software system via UML (or other MOF-based languages) may have come to mind while reading this book. This is partially true. Yet we have already shown that you can model domains to a large extent if you limit the domain sufficiently, strongly standardize its concepts, and define a suitable DSL. Many aspects will of course remain unresolved or impracticable: specifically, a way of specifying algorithmic behavior is still missing.

To address this problem, the OMG defines the action semantics with UML 2.0, which also be used with other MOF-based languages. Action semantics allow the modeling of procedural behavior. Note that the OMG only defines only the abstract syntax, not a concrete syntax. The semantics are described verbally. It is therefore up to the tool manufacturers to define their own textual or graphical notations for standardized semantics. It is thus possible to represent the same behavior both textually and graphically.

The action semantics comprise the following elements:

- Variables (*instance handles*): assigning, reading, also for sets of variables (*sets, bags, sequences*)

- The usual arithmetic and logical operations
- Typical features of sequential programming languages such as *switch*, *if*, *for*, statements, the block concept
- Instance creation, destruction of instances
- Class *extents* that can be prompted with SQL-like queries
- Navigation across associations
- Creation of links (instantiation of associations) and the deletion of links
- Generation of signals, including parameters
- Definition of functions with input and output parameters, and ways of calling them
- Timers

Action semantics do not contain structural constructs such as classes, attributes, and relationships. These are already defined in the structural part of the model. Action semantics merely define ‘behavioral building blocks’ that only make sense in connection with other (partial) models. As a consequence, action semantics segments are always associated with elements of the regular UML model, for example with the operations of classes, or *onEntry* actions in state machines.

Here is a simple example of the use of action languages. Figure 12.3 shows the class diagram, which serves as a basis for the example. We use the syntax that is also used in the tool *iUML* by Kennedy Carter [IUML] for the concrete syntax for the action semantics.

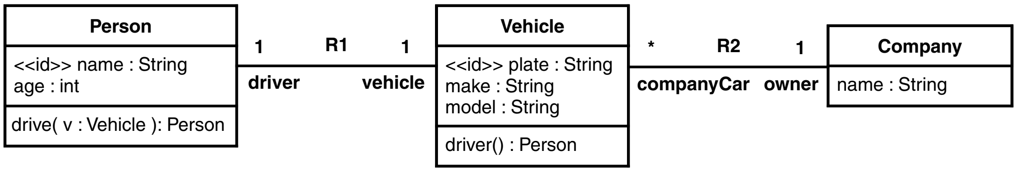


Figure 12.3 A simple model to illustrate action semantics

First, we implement a ‘main program’ that works with instances of classes from Figure 12.3 and creates an instance of the class *Vehicle*. Then we assign a value to the make and the model.

```
myVWBus = create Vehicle with plate = "HDH-GS 142"
myVWBus.make = "Volkswagen"
myVWBus.model = "Transporter T4 TDI"
```

The *with* clause assigns values to the identifying attributes (see the *«id»* stereotype) as early as during object creation, similar to passing constructor parameters in OO languages. Then we can define an instance of *Person* that will subsequently become the driver.
We can now call the operation *drive()* to let the driver drive the vehicle.

```
[actualDriver] = drive[aVehicle] on john
```

What is still missing, of course, is the implementation of the operation *drive()*. The least it must do is to instantiate the association *R1* – that is, to create a link between the two relevant objects.

```
link this R1 aVehicle
```

This establishes the bidirectional association between the *Vehicle* and a driver. Now you can ask the car who is currently driving it. This corresponds with the implementation of the *drive()* operation of the class *Vehicle*.

```
theCurrentDriver = this.R1."driver"
```

Now we briefly introduce the query operations. Let's assume that we want to find all the individuals in the system:

```
{allPersons} = find-all Person
```

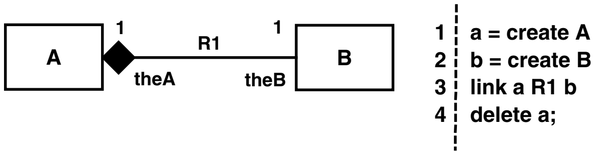
The braces state that *allPersons* is a set of objects instead of just one. It's also possible to limit such a search. For example, all vehicles of the brand *Audi* can be looked for.

```
{audis} = find Vehicle where make = "Audi"
```

One could criticize the fact that action semantics are just another programming language. In principle, this is correct, but it misses the point:

- One doesn't have to deal with platform specifics such as memory management, the definition of a link between two *EntityBean* instances, or the use of relational keys.
- We are dealing with 'semantic building blocks' here. The concrete syntax can look different. In principle, the same could be accomplished with traditional languages, but this is not common.
- Action semantics are totally integrated in the model: the concluding example clarifies this.

Let's assume we are dealing with the trivial model in Figure 12.4. To the right we can see a few lines of action language.



```
1 a = create A
2 b = create B
3 link a R1 b
4 delete a;
```

Figure 12.4 Another example of a model

During the mapping to a programming language – that is, during implementation code generation – the following approximate code is generated:

Line 1	Creation of a new instance of *A*, creation of variable *a*.
Line 2	Creation of a new instance of *B*, creation of variable *b*.
Line 3	Now things become more interesting: the attribute *theB* is assigned a pointer to *b* by *a*. Since the association is bidirectional, *b.theA* will also automatically point to *a*. This is something the programmer need not explicitly program – the generator can do this automatically using the information in the class diagram.
Line 4	Due to the fact that *B* is compositionally associated with *A*, the generator can create code that will delete instance *b* when *a* is deleted.

The information for code generation is also taken from the structure diagram. The developer only has to write the *link...* statement in ASL: the – potentially significantly more complex – implementation code is generated automatically.

Let's once more address the abstraction effect of action semantics. This textual representation says nothing about its realization, of course. If the respective system is implemented with EJB, for example, you would generate different code than you would were you generating the implementation for some embedded system.

12.2.7 Core Models

To be able to benefit as much as possible from the MDA, as many aspects as possible must be standardized. This includes platforms — already done via J2EE, .NET, CORBA or Web Services, at least on the technical level — and transformation languages, which happens in the context of QVT (see Chapter 10). To allow users to model the application logic independently of the platform, it is also necessary to standardize the metamodel for specific domains. Thus it not only becomes possible to standardize the transformation languages, but also to capture reusable transformation rules in *transformation libraries*. The developer models their application via the standardized UML profile or metamodel for the respective domain, and commercial or Open Source transformation modules generate the platform-specific code from it.

These standardized metamodels are called *core models* by the OMG in the context of the MDA. Various core models are being developed[5] currently that are at present all defined as UML profiles. Among these are:

- *UML profile for CORBA*, which defines the mapping of PIMs to CORBA.
- *UML profile for CCM*, which defines the mapping to CCM, the CORBA component model.
- *UML profile for EDOC*, which defines the metamodel for the definition of PIMs for distributed enterprise systems.

[5] If this looks familiar to you it is probably because, prior to the introduction of MDA, attempts were made to standardize profiles for certain domains.

- *UML profile for EAI*, which defines the metamodel for PIMs for EAI applications and other loosely-coupled systems.
- *UML profile for Quality of Service (QoS) and Fault Tolerance*, for real-time and safety-critical systems.
- *UML profile for Schedulability, Performance and Time*, which defines a metamodel for PIMs whose real-time properties can be analyzed quantitatively.
- *UML Testing Profile*, which supports automated testing in MDA-based systems.

The current status of the various profiles can be looked up at [OMGP].

12.2.8 Controlling the PIM to PSM Transformation

In some cases the transformer cannot transform a model because the information in the source model is not specific enough. For example, the target metamodel can offer different ways of realizing a construct of the source metamodel. In [Fra02] David Frankel describes four alternatives way of proceeding in such a case:

- You can encode into the transformation the fact that a particular alternative shall always be used.
- The developer can define which of the alternatives shall be used manually.
- Developers can state directly in the PIM which alternatives should be used in the PSM as *model markings* – see below.
- The decision criteria that let the transformer decide which alternative should be used can be abstracted into the PIM (more about this later).

Model Markings

The MDA proposes the concept of *model markings*. Model markings are additional information in a transformation's source model that control the transformation. These annotations depend typically on the target metamodel. Figure 12.5 illustrates the principle.

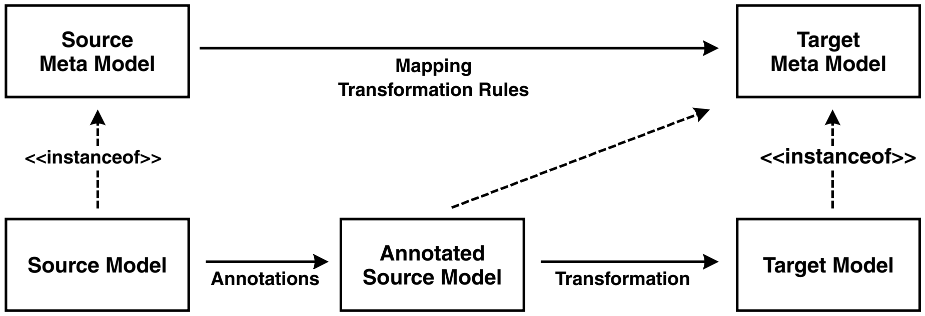


Figure 12.5 An example of model markings

J2EE again serves as an example here. In the PIM we define a *BusinessEntity* called *Account*. In J2EE, *BusinessEntities* can be represented in two different ways: either as *EntityBeans*, or as *Stateless Session Beans* that process data transfer objects. One could mark the source model as shown in Figure 12.6.

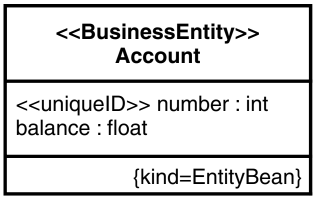


Figure 12.6 An example of an entity with EJB-specific markings

It is important to ensure that the source model itself is not changed: instead, a 'reference copy' is defined that contains additional information. Figure 12.7 illustrates this idea.

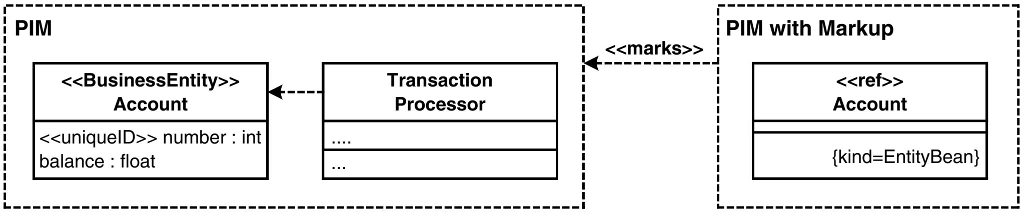


Figure 12.7 The relationship between PIM and marked PIM

The marked PIM only contains model elements of the original PIMs that should be marked, or rather, only the additional markings themselves. This means that the marked PIM does not need to be adapted manually if the PIM is changed[6].

Decision Criteria in the PIM

There is another option that allows you to work mostly without markings if you are prepared to accept other consequences. This mechanism requires you to extend the source metamodel in such a way that enough information is present in the model for the generator to choose between the different alternatives. Figure 12.8 shows how this could appear:

[6] This clarifies how important it is for MDA that the modeling tools provide the according powerful repositories and functionalities. Most of today's modeling tools have not reached this level yet.

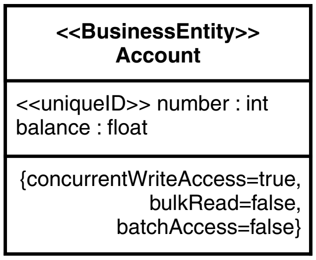


Figure 12.8 *Entity* with target-platform-independent markings

The decision about which implementation alternative to use in which case in J2EE can now be delegated to the J2EE transformations. Of course, the metamodel must be extended, and this is not always possible. However, the metamodel is not extended using the concepts of the target metamodel, but with general information that can be used by the transformer. This constitutes an enrichment of the source metamodel, rather than a 'pollution' of it with target metamodel-specific constructs.

12.2.9 Executable UML

The term *executable UML* is often heard in the context of MDA. This is not a formal standard, but a collective term for various endeavors that all pursue the goal of establishing UML as a fully-fledged programming language. To this end, UML must be purged of all redundancy and ambiguities, resulting in executability of UML diagrams: the smaller the metamodel of the modeling language (here UML), the easier it is to implement a compiler or an interpreter for it. Another necessary ingredient of executable UML is an action language (see Section 12.2.6), which is necessary to define complete implementations of software systems.

It is important to understand that – contrary to the MDSD approach – executable UML is *not* a *domain-specific* profile of UML[7]. The idea is rather to define a universal, UML-based programming language.

Further information about executable UML can be found in Steve Mellor's book [Mel02] or in the documentation section of Kennedy Carter's *iUML* [IUML].

[7] Unless you consider Turing-calculable functions a domain.

Part III
Processes and Engineering

After approaching the constructive and technical aspects of domain architectures in the Part II, we now want to demonstrate how the domain architecture is created in the course of a project, and how its creation can be synchronized with actual application development. A suitable process is pivotal for the success of MDSD projects. This part of the book therefore examines MDSD from the perspective of correct project execution, and introduces relevant process building blocks, as well as engineering methods.

We start with a number of best practices that can be combined into a practical and pragmatic development process. These include architecture elaboration/definition and an introduction to product-line engineering. Next, we tackle testing and versioning in the context of MDSD. The part concludes with two case studies: one looks at component-based development for distributed embedded systems, the other one comes from the world of enterprise systems.

13 MDSD Process Building Blocks and Best Practices

with Jorn Bettin

13.1 Introduction

This chapter introduces important proven process building blocks that enable and support the successful use of Model-Driven Software Development in projects. We abstract from the architecture-centric case outlined in Chapters 2 and 3. The techniques that were used partly intuitively in those chapters are explicitly elaborated, generalized, and detailed in this chapter.

Most processes and practices can quite easily be transferred to general – that is, non architecture-centric – MDSD. Techniques that only make sense in architecture-centric cases, or that require a specific interpretation, are explicitly marked as such. We are going to build on the MDSD terminology defined in Chapter 4, so we recommend that you read that chapter first.

We do not intend to introduce a self-contained and complete development process – enough literature is already available, ranging from agile to heavyweight. Instead, we are going to focus on those process-related aspects that are specifically relevant in the context of MDSD. This also means that there is a certain degree of freedom over how formally these best practices are applied concrete projects.

We recommend that the best practices are embedded into an iterative-incremental, and in particular, agile development method. MDSD does not conflict with the latter, but is in fact well suited to enhance its advantages. Theoretically, MDSD can even be combined with a waterfall development process. However, the well-known risks of waterfall approaches remain, which is why we – quite independently of MDSD – regard them critically.

13.2 Separation Between Application and Domain Architecture Development

13.2.1 The Basic Principle

In the case study in Chapter 3 we saw the advantages that are gained from the separation of domain-related application development and technical infrastructure. We are able to formalize and generatively support the software architecture completely independent of the concrete application. In our example, we obviously dealt with the architecture-centric case: the domain was software infrastructure for e-business applications. A domain architecture with a correspondingly architecture-centric DSL (UML profile), a corresponding platform (J2EE and Struts), and suitable generator templates were developed. If we now generalize the principle, we get Figure 13.1.

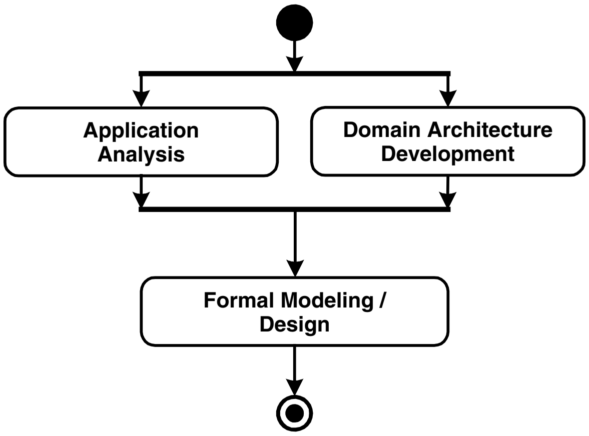


Figure 13.1 Domain-related analysis and domain architecture as a basis for formal modeling

One of the most important basic ideas behind MDSD is the realization that the formal modeling step implies two prerequisites that are not without reciprocity, but which can for the most part be developed in parallel:

- The functional/professional requirements for an iteration or an increment of the concrete application must be known.
- The formal language to be used for modeling (the DSL) must be defined. In addition, for automatic further processing, the language must be bound to the concrete MDSD platform in the shape of transformation rules. This is what the term *domain architecture* sums up (Chapter 4).

As its name indicates, the domain architecture formalizes and supports a domain. In principle, this domain is independent of a single application (unique product), or in other words, it covers a software system family.

The activity diagram in Figure 13.1 should not be misunderstood as a waterfall process. It primarily shows the basic principle on which each iteration is based, independently of its weighting.

Formal modeling serves to connect the concrete application's concepts with the concepts provided by the domain architecture – more precisely, the functionality is expressed in the language (DSL) provided by the domain architecture. The formal model is then transformed with the generator's support and mapped to the platform. We have already seen how this can look in an architecture-centric case, and an example from the embedded systems domain is given in Chapter 16. In the case of the insurance domain, the DSL could contain constructs relevant for the insurance domain, for example by supporting effective modeling of insurance products. In consequence, the platform would consist of prefabricated domain-specific components such as *tariff calculator* or *contract data entry*, and the transformations would for example generate configurations for the domain-specific components from the insurance product model, which would then be evaluated at runtime. An insurance application could thus be created 100% automatically from the model. In contrast to the architecture-centric case, this use case does not require any manual coding.

The basic principle introduced above suggests that one should also apply the separation between application and domain architecture development for the process and organizational level, as well to maximize the positive effects. Accordingly, we suggest a separation of the (domain) architecture development thread and the application development thread.

13.2.2 Domain Architecture Development Thread

There are several artifacts and activities that are necessary or helpful for the creation of a domain architecture. The MDSD architecture development thread aims at reusability as well as at quality and efficiency enhancement. From a process perspective, it is the central aspect of MDSD. Figure 13.2 therefore first zooms in on the *creation of a domain architecture* activity, as discussed in the previous section.

The partitions displayed in the diagram divide the activities into the categories *domain*, *transformation*, and *platform*. In each category, artifacts of the domain architecture are produced (shown in light gray). Only the normal case scenario with the most important dependencies is shown here, because – in our experience – including everything else would obscure the essential issues. In particular, no iteration cycles are shown.

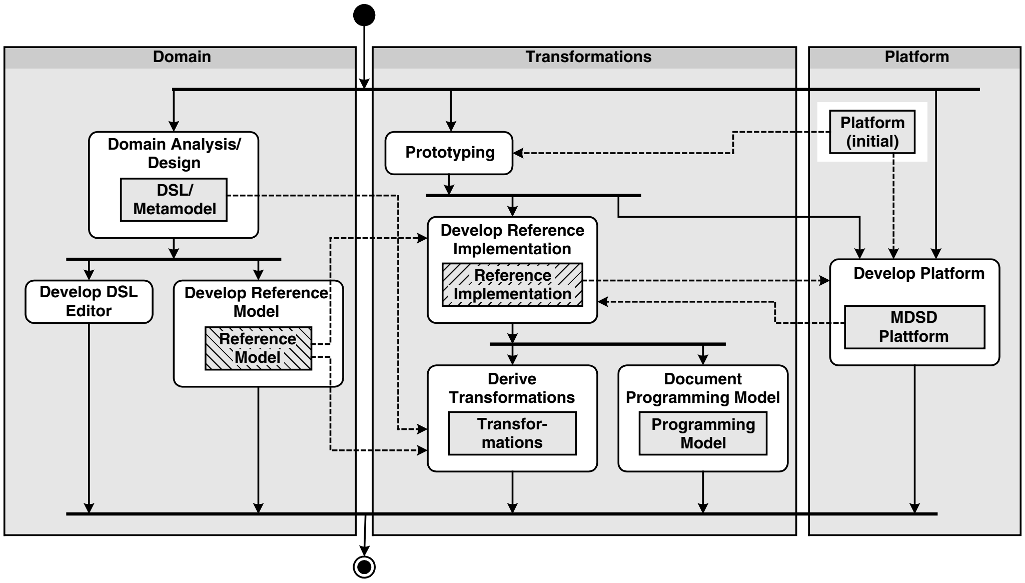


Figure 13.2 Creation of a domain architecture

At the beginning of a project, a complete iteration though the stages shown in Figure 13.2 can take several weeks, compared to a couple of hours or even minutes in the course of the project, depending on the scope of the extensions or modifications. Parts of the domain architecture might already be present at project start-up (for example re-use), or a derivative of an existing domain architecture might be created.

At the beginning of a project we recommend an *elaboration phase* in which the architecture development thread is initially run completely, as a kind of bootstrapping activity for the project. This can be omitted if the project is conducted on familiar territory, for example if it is part of an existing software system family.

The following sections list the most important steps (actions/activities) and result types in detail.

Prototyping

A platform that is supposed to be used is often already in existence at the beginning of a project, such as J2EE or specific frameworks. One goal of the MDSD architecture development thread is to merge these artifacts with a semantically-rich and domain-specific MDSD platform (Section 7.6). To this end, it always makes sense first to gather experience with a prototype in terms of a *proof of concept*. Among other things, this prototype can also be considered a first step towards the MDSD platform.

Developing the Platform

We defined the term *MDSD platform* in Chapter 4 and explained its constructive aspects in Chapter 7. The term *runtime system* is used synonymously. Runtime system components are ideal candidates for re-use, even across the boundaries of software system families.

As we have seen, the platform constitutes the foundation on which to base generated and non-generated code, and for keeping transformations simple. The generative portion of the domain architecture possesses a dependency on the runtime system components (platform) used, but the opposite is not true.

Development of the platform should also progress iteratively. Refactoring techniques (see [Fow99]) can be applied beneficially in this context.

It should also be observed that the border between platform and generated code can change in the course of the domain architecture's evolution, in either direction. Chapter 7 elaborated the criteria for this.

Creating a Reference Implementation

The *reference implementation* is merely an intermediate result of the MDSD architecture development thread, but a very important one when it comes to the creation of a domain architecture.

The reference implementation should not be misinterpreted as a simple, isolated example from which one can derive suggestions for implementation if necessary. It can be created from a prototype, but serves a more significant purpose: together with the reference model/design, it demonstrates the application and realization of the DSL belonging to the domain. This two-part reference exemplifies the transition from model to implementation on the respective platform. For a new software system family, the reference implementation is first created by hand. Later, the transformations are derived from it. The generative implementation of the reference model, possibly plus the manually programmed domain logic, must then result in a runnable reference implementation.

The full added value of a reference implementation will only be obtained from the interplay between the reference implementation and a reference model (and thus a DSL – see below). The concrete functional content of a reference implementation *per se* is irrelevant – only the domain matters. Yet as a rule a more or less sensible use case is implemented, if only in a minimalistic form. Sufficient coverage of the DSL's constructs and their combinations is much more important. At the same time, the reference implementation demonstrates the use of the MDSD platform and its API.

For a new software system family, the reference implementation is first created completely by hand, but as soon as automatic transformations are available, the reference implementation is reduced to manually-programmed domain logic, if it exists. The rest of the application can then be generated from the reference model.

If you already have a number of applications in a specific domain and want to switch to a model-driven development process, the domain architecture can also be extracted from the existing applications, as long as the implementations are well-structured. This is often not the case with typical legacy applications, however.

It is also important to note that the domain architecture's evolution, and particularly the early stage of bootstrapping (such as DSL definition and stabilization), will usually have repercussions for the reference implementation, and maybe also for the platform. This is completely normal in the context of iterative-incremental software development: a strict waterfall model would most likely be counterproductive here.

Domain Analysis/Design

This activity primarily serves to find the domain's metamodel and a suitable, concrete DSL. Here we only list the best practices for constructing a DSL.

An *architecture-centric* DSL is also called a *design language*. The use of UML as its basis is typical (but not mandatory) for such a design language, as the case study in Chapter 3 shows in depth. UML is completely unusable for some aspects, however, such as for example the modeling of a GUI layout, so that one may have to use another notation. Ultimately the concrete syntax always assumes a less important role than the abstract one (see Chapter 4 and Section 11.2.2).

The following rules should be observed when designing any DSL:

- The DSL should be as abstract and as free of any technical terms that constitute an implementation detail of the MDSD platform as possible, for example the use of *EntityObject* instead of *EntityBean* as a stereotype. This measure leads to models whose technical realization will only be recognized in the context of a platform binding. This makes later migration or architectural changes, for example, easier. Such DSLs are also reusable for various software system families. We have sometimes intuitively labeled such models platform-independent (PIM), but this expression is relative (see Chapter 4): typically, models are *independent* of an industry standard platform such as J2EE, but *dependent* on the concepts of the MDSD platform of the domain architecture, because the DSL precisely serves the purpose of enabling the use of these concepts on the model level.

- If possible, the DSL should cover all relevant concepts of the domain with language elements. Ideally, all schematically-implementable code fragments of the reference implementation should be covered by constructs of the DSL. Our case study exemplifies this: The stereotypes *«Entity»*, *«ProcessObject»*, *«Presentation»*, *«SystemUseCase»*, *«Activity Controller»*, *«ControllerState»* and so on name and cover precisely the architectural concepts of our example domain architecture.
- The DSL should be as compact as possible and not have any redundancy. It can also possess dynamic constructs, for example to map business processes and controller logic in the shape of activity or state diagrams.
- The DSL must make the well-formedness of models verifiable. It must be guaranteed that all modeling notations offered by the base language are excluded, in case they are not legal for the DSL, especially if the DSL is a specialization of a more general language, such as a UML profile as a specialization of the UML.

With the conception of the DSL, the architect inevitably also draws the dividing line between generated code and domain logic – and thus the degree of freedom of the developers. One extreme is the attempt to expand the DSL to such an extent that manual programming is no longer needed (see Chapters 4 and 7). These days, this approach – if applied in the extreme – is neither practicable nor useful for typical business applications in the context of architecture-centric development.

The pivotal question here is which implementation aspects should be covered by an architecture-centric DSL and which should not. The following questions can serve as a guideline:

- Does the reference implementation feature code fragments with copy-paste characteristics that have not been generated yet?
- Would modeling of these aspects be simple and compact, or would it in contrast require even more effort and be more comprehensive than manual coding?

This important assessment requires some experience and sensitivity toward the subject matter.

The DSL must be documented in order to be usable. This includes the following aspects:

- The concrete syntax, for example a UML profile or an XML schema.
- The abstract syntax, for example as a MOF diagram – see Chapter 6.
- The static semantic — that is, constraints or modeling rules: which constructs are not allowed, which are mandatory. In the case of a UML-based modeling language, this is a part of the profile (see Chapter 3).
- The semantics — the meaning of the language constructs in textual form. The semantics are formally defined by the reference implementation and the transformation rules, so their definition is geared to a concrete platform. To avoid this, and to increase its reusability level, the semantics should be described in as general form as possible, using the architectural concepts rather than the platform itself.
- An example model, for example, the reference design (see the next section)

The definition of an adequate modeling language is certainly one of the greatest challenges in MDSD. Section 13.5 offers a couple of concrete tips. Some practical experience, or an existing basis such as the UML profile from this book's first case study, are required. From that point the modeling language can undergo evolutionary development. For example, it is typical for it to take a while before one notices that the original design of specific modeling constructs is not sufficiently abstract and needs to be generalized. Refactoring is therefore a strong ally at this metalevel. Admittedly, *fundamental* changes in the abstract syntax – of the actual language structure – and the semantics can bring about extensive changes of existing transformation rules. In contrast, language *extensions* are uncritical. During the elaboration phase an adequate emphasis should be put on the DSL, so that the transformation rules are only derived when reference implementation and reference design are coherent.

As a rule, modeling languages are not created in a vacuum. To get a feel for whether the chosen constructs are adequate and ergonomic, you have to use the language in practice. This purpose is being served by the reference design.

Creating the Reference Model/Design

The *reference model* is an instance of the DSL, in that it expresses a domain example via the means of the DSL.

The interplay with the reference implementation is important: the reference model and the reference implementation together exemplify the syntax and semantics of the DSL, and thus make concrete the concept of the domain architecture in detail.

Documenting the Programming Model

The definition of a programming model is only relevant if the domain architecture contains ‘semantic gaps«, meaning that a code framework emerges via the model transformation that must be supplemented by the application developer in a programming language to enable the creation of a runnable application.

We established the term *programming model* in Chapter 7. CORBA, for example, defines an abstract interface definition language (IDL) with mappings to various programming languages such as C++ and Java. The programming model of the respective language mapping then defines naming conventions for the mapping of IDL constructs to language constructs such as classes, attributes, and methods. These conventions are obeyed by the IDL compiler (generator), which generates the respective signatures and skeletons. Generation from the IDL definition creates a defined API that allows application developers to program the application logic. The programming model defines specific idioms and patterns that describe how to treat the respective architecture correctly – not all rules can be automatically checked by the compiler or the runtime system.

Transferred to MDSD, the programming model describes the application developer's view of the domain architecture, or more precisely, transformations or generated code and platform, as well as rules for how to handle it correctly. The goal is among others to make it transparent to the developer which programming language-related artifacts are created from the DSL's constructs that are relevant *to them*. An association present in the model, for example, can mean a set of access operations, such as *getElementAt()*, *addElement()*, *removeElement()* and so on, on the implementation level. From the developer's viewpoint, the implementation is irrelevant: they only needs the signatures. These will then constitute the programming model of the DSL association thus denoted.

The programming model is quite easily to document in table form. The table contains – besides the respective construct of the DSL – a reference to an adequate excerpt of the reference model and the reference implementation. Opinions over the need for explicit documentation of the programming model vary, but the deliberate definition in the form of MDSD transformation and platform API is necessary. The reference implementation constitutes an *implicit* documentation. However, if the programming model contains 'do's' and 'don'ts' that cannot be enforced or controlled via tools, explicit documentation is mandatory.

It turns out that a tutorial in the form of a walkthrough that explains to developers how to develop concrete applications using the DSL is most appropriate. Such a tutorial should cover the DSL as well as other aspects of the programming model, including the code that needs to be written manually, or how to operate/integrate the generator tool.

The initial programming model and its documentation are typically constructed in the course of the elaboration phase. The programming model is also subject to iterative improvements, of course.

Deriving Transformations

This activity formalizes the mapping of a DSL to a platform and programming model, to the extent that an automatic transformation can transform a given application model into an implementation or a skeleton.

In our case study a set of generator templates was created in this step. The templates were derived from the reference implementation with the help of the reference model. The generator framework used in the case study in Chapter 3 relocates part of the metaprogramming to the creation of a DSL's metamodel, implemented in Java. From the process viewpoint, this separation is merely an implementation detail and thus irrelevant.

If the domain logic of an application cannot entirely be expressed by the DSL, techniques for the integration of generated and non-generated code are needed (see Chapter 7).

Creating a DSL Editor

Not all DSLs are UML profiles, so that a standard tool can be applied, with varying degrees of effectiveness. In the case of highly-specialized domains it is common practice and advisable to create a specific tool for defining DSL-conforming models, for example to further increase the ergonomics and therefore the efficiency of the MDSD approach. It comes down to a question of the cost-value ratio, which can only be answered for each individual case. This topic is considered further in Section 13.5.

13.2.3 Application Development Thread

This section deals with the viewpoint of the application developer's who works with a given domain architecture (see Figure 13.3). (We see this as a role.)

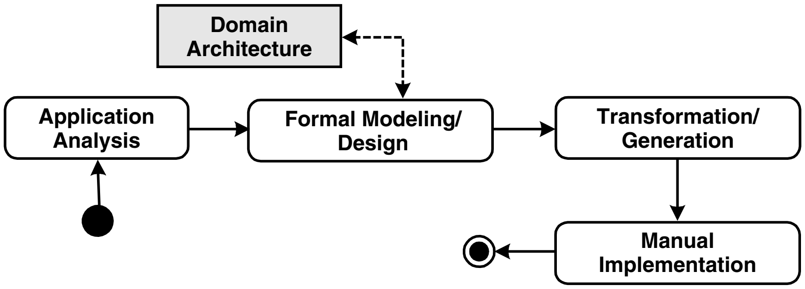


Figure 13.3 Activities in the application development thread

Here too the simple, normal case without iteration cycles is displayed. One cycle can take anything from a couple of days to only minutes, depending on the intensity of the single steps.

Formal Modeling/Design

The analysis and architectural threads meet in this step: the functional requirements are now expressed in the domain architecture's language – the DSL. The reference model serves as an orientation guide in this context. This step constitutes a real gain of information and insight and therefore cannot be automated.

A working feedback loop to the architecture development thread should be established here, because deficiencies or errors in the DSL are regularly discovered, especially in the early stages of a domain architecture. It is also typical for the potential for further automation to be discovered during application development, which leads to extension or even generalization of the DSL.

Generation

This step can be executed purely mechanically. No information gain occurs when compared to the formal model: it is transformed automatically into a form suitable for the MDSD platform via the domain architecture's transformations. During this process integration points for manually-programmed domain logic can emerge in defined locations. These can be protected regions, whose content remains intact during iterative regeneration, or implementation classes to which the implementation framework delegates (Section 8.2.6).

Manual Implementation

Domain logic that *cannot* be expressed in the DSL must be added manually after generation has taken place. In our case study, these are exactly the contents of the protected regions in the implementation skeleton.

Repercussions for the domain architecture can occur even during implementation. The project's organization must allow for the necessary feedback loops.

Organizational Aspects

The separation between application development and domain architecture development should ideally be supported not only by a suitable process structure, but also by adapting the organizational structure of a team, project, or company. These are treated in Chapters 19 and 15.

13.3 Two-Track Iterative Development

We have now discussed the separation of roles and artifacts between application development and domain architecture development. This section is about the synchronization of both threads. There is obviously a dependency from application development to domain architecture development – in the same way that you might depend on the development of a framework that you use. From a requirement management viewpoint, this means that the application development team assumes the customer's role for domain architecture development.

When developing the domain architecture, you should simultaneously also develop at least one application based on that domain architecture as if it were a separate project. In practice, this means that in one iteration the application developers always use the domain architecture developed in the last iteration, so that they are always one iteration step ahead of the application developers. Make sure that the application developers always provide feedback to the domain architecture developers.

New versions of the domain architecture are always integrated at the beginning of an iteration. To reach a sufficient level of agility during the development process, iterations should not take longer than a maximum of four to six weeks. Ideally and to simplify matters, we recommend that a fixed timeframe is set for all iterations (*timeboxing*). This leads to a regular development rhythm that the teams will get used to.

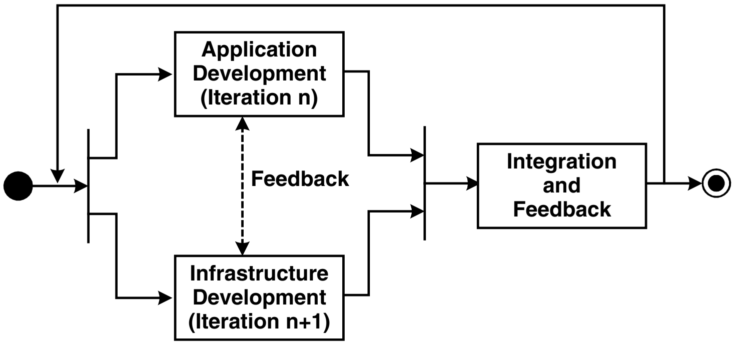

Application Development (Iteration n)
Feedback
Infrastructure Development (Iteration n+1)
Integration and Feedback

Figure 13.4 Iterative two-track development

Note that the incremental, iterative process based on synchronized timeboxes does *not* exclude a domain analysis prior to entering the iterative cycle. On the contrary, a good understanding of the basic concepts of the domain is actually needed. As soon as application development is under way, further domain analysis takes place iteratively – as part of the architecture development thread that is now delegated to a project of its own.

Infrastructure teams (the domain architecture developers) sometimes show a tendency to jump at interesting technologies to impress the world – of course with the best intentions – with some new ‘silver bullet’. This risk is alleviated most efficiently by the formation of an architecture group that consists of representatives from the application development team (see Chapter 19). Such a group is entitled to determine the functionality and features of infrastructure development via scope trading from iteration to iteration, and decides over the acceptance of infrastructure functionality via the validation of (domain architecture) iterations. This guarantees that the developed domain architecture constitutes real added value for application development and actually supports application developers in their everyday work.

Timeboxing with a fixed budget, scope trading, and validation of iterations are agile techniques that support iterative requirements management and can be particularly helpful in combination with two-track, iterative development for MDSD. Here, we only want to sum up the basic ideas in a few sentences, because these topics are in principle independent of MDSD.

A fixed budget is available for each timebox. At the beginning of each iteration, the features and priorities for the iteration are negotiated with stakeholders, for example customers and end users, in a scope-trading workshop. For reasons of risk minimization, architectural aspects must be considered as well. The timebox budget must not be exceeded. Within the timebox, the feature set remains constant, so that developers can pursue their goal of delivering software that can be validated at the end of the timebox. The validation at the end of an iteration conducted by the stakeholders decides which features meet the requirements and which features must be reengineered. New requirements that have been recognized in the meantime are reprioritized on an equal footing with unfulfilled requirements.

Further sources dealing with these topics can be found on the Web at *http://www.mdsd.info.*

In the context of MDSD these practices can be especially helpful for establishing a working feedback loop between application development and domain architecture development. In this context the application developers serve as a representative team of scope trading stakeholders in domain architecture development.

13.4 Target Architecture Development Process

Best practices for the domain architecture development process are one thing. Other important concerns are:

- How do you come up with a reasonable target architecture?
- How do you make it ‘ready for MDSD’?
- How do you implement it in non-trivial projects?

The following sections provide some help in this area: they deal with process best practices, with a focus on the target architecture – which of course is reflected by the domain architecture at the metalevel too. In other words, it’s a relevant perspective for all activities in the architecture development thread introduced above. Many of the following statements and suggestions concerning target architecture development are independent of MDSD, but some of them are MDSD-aware, as we will see.

Software architecture is generally too technology-driven. You hear statements such as “We have a Web Service architecture”. This statement is not very informative, because it describes only one aspect of the overall system (its communication), and because Web Services are a particular implementation technology for that aspect. There is much more to say about the architecture, even about its communication aspects, than just a realization technology. The same is true of 'EJB architectures' or a 'thin-client architecture'. Too early a commitment to a specific technology usually results in blindness to the concepts and too tight a binding to the particular technology. The latter in turn results in a complicated programming model, bad testability, and no flexibility to change the technology as QoS requirements evolve. It obscures really important issues.

Another problem is the 'hype factor'. While it is good practice to characterize an architecture as implementing a certain architectural style or pattern [POSA1], some of the buzzwords used today are not even clearly defined. A 'service-based architecture' is a classic. Nobody knows what this *really* is, and how it is different from well-designed component-based systems. There are many such misunderstandings. People say 'SOA', and others understand 'Web Service'! Also, since technologies are often hyped, a hype-based architecture often leads to too early – and wrong – technology decisions.

Another problem is what we usually call *industry standards*. A long time ago, the process of coming up with a standard was basically as follows: try a couple of alternatives, see which one is best, set up a committee that defines the standard based on previous experiences. The standard is therefore usually close to the solution that worked best. Today this is different. Standards are often defined by a group of (future) vendors. Either they already have tools, and the standard must accommodate all the solutions of all the tools of all the vendors in the group, or, there is no practical previous experience and the standard is defined from scratch. As a consequence of this approach, standards are often unusable because there was no previous experience, or overly complicated (because it must satisfy all the vendors). Thus, if you use standards for too many aspects of your system, your system will be complicated!

Finally, there's *politics*.

All these factors, taken together, prevent people from thinking about the really relevant aspects of an architecture. In our opinion these include architectural patterns, logical structures (architectural metamodels), programming models for developers, testability, and the ability to realize key QoS concerns.

The following sections sketch what we consider a reasonable approach to software architecture – that is, MDSD-target-architecture. It also paves the way for automation of many aspects of software development, a key ingredient to MDSD and product-line engineering.

We are not the only ones seeing this problem in current software architecture, of course. There are good architectural resources that you should definitely read, such as [POSA1], [POSA2], and [POSA3], as well as [JB00], [VSW02] and [VKZ04].

13.4.1 Three Phases

The development of a software architecture, especially one that can be used in the context of MDSD, should be executed in three phases. In each of these phases certain core artifacts are created – these are highlighted in smallcaps in the following:

- *Elaboration*. In the first phase, the elaboration, you define a TECHNOLOGY-INDEPENDENT ARCHITECTURE. Based on it, you define a workable PROGRAMMING MODEL for the developers that work with the architecture. To let developers run their applications locally, a

MOCK PLATFORM is essential. Finally in this phase, you define one or more TECHNOLOGY MAPPINGS that project the TECHNOLOGY-INDEPENDENT ARCHITECTURE on a particular platform that provides the required/desired QoS features. A VERTICAL PROTOTYPE verifies that the system performs as desired – here is where you run the first load tests and optimize for performance – and that developers can work efficiently with the PROGRAMMING MODEL.

- *Iteration*. The second phase iterates over the steps in the first phase. While we generally recommend an agile approach, we emphasize the fact that you typically don't get it right first time. You usually have to perform some of the steps several times, especially the TECHNOLOGY MAPPING and the resulting VERTICAL PROTOTYPE. It is important that you do this *before* you dive into Phase 3, automation.
- *Automation*. The third phase aims at automating some of the steps defined in the first phase and refined in the second phase, making the architecture useful for larger projects and teams. First, you will typically want to GENERATE GLUE CODE to automate the TECHNOLOGY MAPPING. Also, you might often notice that even the PROGRAMMING MODEL involves some tedious repetitive implementation steps that could be expressed more briefly with a DSL-BASED PROGRAMMING MODEL. Finally, MODEL-BASED ARCHITECTURE VERIFICATION helps to ensure that the architecture is used correctly even in large teams.

In the following sections we outline each of these steps, while an example of this approach is given in the case study in Chapter 17.

13.4.2 Phase 1: Elaborate

The best practices of this phase are relevant for the activities of *prototyping, document programming model* and *platform development* of our domain architecture development thread.

Technology-Independent Architecture

How do you define a software architecture that is well-defined, long-lived and feasible for use in practice? The architecture has to be reasonably simple and explainable on a beer mat. You want to make sure that the architectural concepts can be communicated to stakeholders and developers. Implementation of functional requirements should be as efficient as possible. The architecture must survive a long time, longer than typical hype or technology cycles. The architecture might also have to evolve with respect to QoS levels such as performance, resource consumption, or scalability.

To achieve these goals, define the architectural concepts independently of specific technologies and implementation strategies. Clearly define concepts, constraints, and relationships of the architectural building blocks – a glossary or an ARCHITECTURAL METAMODEL can help here. Define a TECHNOLOGY MAPPING in a later phase to map the artifacts defined here to a particular implementation platform. Use well-known architectural styles and patterns here. Typically these are best practices for architecting certain kinds of systems independently of a particular technology. They provide a reasonable starting point for defining (aspects of) your system's architecture.

If you use less complicated technology, you can focus more on the structure, responsibilities, and collaborations among the parts of your systems. Implementation of functionality becomes more efficient, and you don't have to educate all developers with all the details of the various technologies that you'll eventually use.

However, the interesting question is: how much technology is in a technology-independent architecture? For example, is AOP[1] ok? In our opinion, all technologies or approaches that provide additional expressive concepts are useful in a TECHNOLOGY-INDEPENDENT ARCHITECTURE. AOP is such a candidate. The notion of components is also one such concept. Message queues, pipes and filters, and, in general, architectural patterns are also useful.

When documenting and communicating your TECHNOLOGY-INDEPENDENT ARCHITECTURE models are useful. We are *not* talking about formal models as they're used in MDSD – we'll take a look at these later. Simple box and line diagrams, layer diagrams, sequence, state or activity charts can help to describe what the architecture is about. They are used for illustrative purposes, to help reason about the system, or to communicate the architecture. For this very reason, they are often drawn on beer mats, flip charts, or with the help of Visio or PowerPoint. While these are not formal, you should still make sure that you define what a particular visual element means intuitively – boxes and lines with no defined meaning are not very useful, even for informal diagrams.

Programming Model

Once you have defined a TECHNOLOGY-INDEPENDENT ARCHITECTURE and your architecture is rolled out, developers have to implement functionality against it.The architecture is a consequence of many non-functional requirements and the basic functional application structure, which might make the architecture non-trivial and hard to comprehend for developers. How can you make the architecture accessible to (large numbers of) developers?

To make sure that it's benefits can actually materialize, you want to make sure the architecture is used correctly. You have developers of differing qualifications in the project team. All of them have to work with the architecture. You want to be able to review application code easily and effectively. Your applications must remain testable.

To achieve all this, define a simple and consistent programming model. A programming model describes how an architecture is used from a developer's perspective. It is the 'architecture API'. The programming model must be optimized for typical tasks, but allow for more advanced applications if necessary. Note that a 'how to' guide that walks developers through the process of building an application is a main constituent of a programming model.

The most important guideline when defining a programming model is usability and understandability for the developer. This is the reason why the documentation for the programming model should always be in the form of tutorials or walkthroughs, not as a reference manual! Frameworks, libraries, and – as we'll see in DSL-BASED PROGRAMMING model on page 270 – domain-specific languages are useful here.

Sometimes it is not possible to define a programming model completely independently of the platform on which it will run (see the next section, TECHNOLOGY MAPPING). Sometimes the platform has consequences for the programming model. For example, if you want to be able to deploy something as an Enterprise Bean, you should not create objects yourself, since this will

[1] Aspect-Oriented Programming [Lad03].

be done later by the application server. There are a couple of simple guidelines that can help you to come up with a programming model that stands a good chance of being mapped to various execution platforms:

- Always develop against interfaces, not implementations
- Never create objects yourself, always use factories
- Use factories to access resources (such as database connections)
- Stateless design is a good idea in enterprise systems
- Separate concerns: make sure a particular artifact does *one* thing, not five.

A good way to learn more about good PROGRAMMING MODELS and TECHNOLOGY-INDEPENDENT ARCHITECTURE can be found in Eric Evans wonderful book on domain-driven design [Eva03].

One of the reasons why a technology decision is made early in the project is political pressure to use a specific technology. For example, your customer's company might already have a global lifetime license for IBM's Websphere and DB2: you therefore have no option but to use those. You might wonder whether the approach based on a TECHNOLOGY-INDEPENDENT ARCHITECTURE and explicit TECHNOLOGY MAPPINGS can still work. If the imposed technology is a good choice, the benefits of the approach described here still apply. If the technology is not suitable (because it is overly complicated or unnecessarily powerful), life with the technology will be easier if you isolate it in the TECHNOLOGY MAPPING.

Technology Mapping

Your software has to deliver certain quality of service (QoS) levels. Implementing QoS as part of the project is costly. You might not even have the appropriate skills in the team. Also, your system might have to run with different levels of QoS, depending on the deployment scenario.You don't want to implement the advanced features that enable all the non-functional requirements yourself. You want to keep the conceptual discussions, as well as the PROGRAMMING MODEL, free from such technical issues.

Therefore, map the TECHNOLOGY-INDEPENDENT ARCHITECTURE to a specific platform that provides the required QoS. Make the mapping to the technology explicit. Define rules about how the conceptual structure of your system (the metamodel) can be mapped to the technology at hand. Define those rules clearly to make them amenable for GLUE CODE GENERATION.

Decide about standards usage here, not before. As mentioned, standards can be a problem, but they can also be a huge benefit. For issues that are not related to your core business, using standards is often useful. But keep in mind: first solve the problem, *then* look for a standard, not the other way around. Make sure PROGRAMMING MODEL hides the complexity too.

Use technology-specific design patterns here. Once you decided on a specific platform, you have to make sure you use it correctly. A platform is often not easy to use. If it is a commonly-used platform, though, platform-specific best practices and patterns should be documented. Now is the time to look at these and use them as the basis for the TECHNOLOGY MAPPING.

Let's recap: the TECHNOLOGY-INDEPENDENT ARCHITECTURE defines the concepts that are available to build systems. The PROGRAMMING MODEL defines how these concepts are used from a developer's perspective. The TECHNOLOGY MAPPING defines rules about how the PROGRAMMING MODEL artifacts are mapped to a particular technology.

The question is now, which technology should you chose? In general this is determined by the QoS requirements you have to fulfill. Platforms are good at handling technical concerns such as transactions, distribution, threading, load-balancing, failover, or persistence – you don't want to have to implement these yourself. So always use the platform that provides the services you need, in the QoS level you are required to deliver. Often this is deployment-specific.

Sometimes you have to decide on your platform based on politics, of course. If a company builds everything on Oracle and Websphere, you'll have a hard time arguing against these. However, the process based on this and the two aforementioned best practices, TECHNOLOGY-INDEPENDENT ARCHITECTURE and PROGRAMMING MODEL, IS still useful, because it allows you to understand the consequences of *not* using the ideal platform. You might have to use a compromise, but at least you will know that it is one!

Mock Platform

Based on the PROGRAMMING MODEL, developers now know how to build applications. In addition to that, developers have to be able to run (parts of) the system locally, at least to run unit tests. How can you ensure that developers can run 'their stuff' locally without caring about the TECHNOLOGY MAPPING and its potentially non-trivial consequences for debugging and test setup? You also want to ensure that developers can run their code as early as possible. You want to minimize dependencies of a particular developer on other project members, specifically those caring about non-functional requirements and the TECHNOLOGY MAPPING. You have to make sure developers can efficiently run unit tests.

Define the simplest TECHNOLOGY MAPPING that could possibly work. Provide a framework that mocks or stubs the architecture as far as possible. Make sure that developers can test their application code without caring about QoS and technical infrastructure.

This mock platform is essential in larger and potentially distributed teams to allow developers to run their own code without caring too much about other people or infrastructure. This is essential for unit testing! Testing one's business logic is simple if your system is well modularized. If you stick to the guidelines given in the PROGRAMMING MODEL (interfaces, factories, separation of concerns) it is easy to mock technical infrastructure and other artifacts developed by other people.

Note that it's essential that you have a clearly-defined programming model, otherwise your TECHNOLOGY MAPPING will not work reliably. Also, the tests you run on the MOCK PLATFORM will not find QoS problems – QoS is provided by the execution platform.

Vertical Prototype

Many of the non-functional requirements your architecture has to realize depend on the technology platform, which you selected only recently in the TECHNOLOGY MAPPING. This aspect cannot be verified using the MOCK PLATFORM, since it ignores most of these aspects. The mapping mechanism might even be inefficient. How do you make sure you don't run into dead-ends? You want to keep your architecture as free of technology-specific concerns as possible. However, you want to be sure that you can address all the non-functional requirements. You want to make sure you don't invest into unworkable technology mappings.

Thus, as soon as you have a reasonable understanding of the TECHNOLOGY-INDEPENDENT ARCHITECTURE and the TECHNOLOGY MAPPING, make sure you test the non-functional requirements. Build a vertical prototype: an application that uses all of the above and implements it only for a very small subset of the functional requirements. This specifically includes performance and load tests.

Vertical prototypes are a well-known approach to risk reduction. In the approach to architecture suggested here, the vertical prototype is even more critical than in other approaches, since you have to verify that the programming model does not result in problems with regard to QoS later. You have to make sure the various aspects you define in your architecture really work together.

13.4.3 Phase 2: Iterate

Now that you have the basic mechanisms in place, you should ensure that they actually work for your project. Therefore, iterate over the steps given above until they are reasonable stable and useful.

Then, roll out the architecture to the overall team if you have larger project teams. If you want to start a model-driven development process, continue with Phase 3.

13.4.4 Phase 3: Automate

The best practices of this phase are relevant for the activities *derive reference implementation/model*, *derive transformations* and *domain analysis/design* of the domain architecture development thread.

You have a TECHNOLOGY-INDEPENDENT ARCHITECTURE. You want to automate various tasks of the software development process. To be able to automate, you have to codify the rules of the TECHNOLOGY MAPPING and define a DSL-BASED PROGRAMMING MODEL. For both aspects, you have to be very clear and precise about the artifacts defined in your TECHNOLOGY-INDEPENDENT ARCHITECTURE. Automation cannot happen if you can't formalize translation rules. An architectural definition based on prose is not formal enough: you want to be able to check models for architectural consistency.

Therefore, define a formal architectural metamodel. An architectural metamodel formally defines the concepts of the TECHNOLOGY-INDEPENDENT ARCHITECTURE. Ideally this metamodel is also useful in the transformers/generators that are used to automate development.

Formalization is a double-edged sword. While it has some obvious benefits, it also requires a lot more work than informal models. The only way to justify the extra effort is by gaining additional benefits. The most useful benefit is for the metamodel not to just collect dust in a drawer, but really to be used by tools in the development process. It is therefore essential that the metamodel is used, for example as part of the code generation in DSL-BASED PROGRAMMING MODELS and ARCHITECTURE-BASED MODEL VERIFICATION.

Glue Code Generation

The TECHNOLOGY MAPPING – if sufficiently stable – is typically repetitive and thus tedious and error-prone to implement. Also, often information that is already defined in the artifacts of the

PROGRAMMING MODEL have to be repeated in the TECHNOLOGY MAPPING code (method signatures are typical examples). A repetitive, standardized technology mapping is good, since it is a sign of a well-though-out architecture. Repetitive implementations always tend to lead to errors and frustration.

To take care of these issues, use code generation based on the specifications of the TECHNOLOGY MAPPING to generate a glue code layer, and other adaptation artifacts such as descriptors, configuration files, and so on. To make that feasible you might have to formalize your TECHNOLOGY-INDEPENDENT ARCHITECTURE into an ARCHITECTURAL METAMODEL. To be able to get access to the necessary information for code generation, you might have to use a DSL-BASED PROGRAMMING MODEL.

Build and test automation is an established best practice in current software development. The natural next step is to automate programming – at least those issues that are repetitive and governed by clearly-defined rules. The code and configuration files that are necessary for the TECHNOLOGY MAPPING are a classic candidate. Generating these artifacts has several advantages. First of all, it's simply more efficient. Second, the requirement to 'implement' the TECHNOLOGY MAPPING in the form of a generator helps to refine the TECHNOLOGY MAPPING rules. Code quality will typically improve, since a code generator doesn't make any accidental errors – it may well be wrong, but then the generated code is typically *always* wrong, making errors easier to find. Finally, developers are relieved from having to implement tedious glue code over and over again, a boring, frustrating, and thus error-prone task.

DSL-based Programming Model

You have defined a PROGRAMMING MODEL. However, your PROGRAMMING MODEL is still too complicated, with a lot of domain-specific algorithms implemented over and over again. It is hard for your domain experts to use the PROGRAMMING MODEL in their everyday work. The GLUE CODE GENERATION needs information about the program structure that is hard or impossible to derive from the code written as part of the PROGRAMMING MODEL. The PROGRAMMING MODEL is still on the abstraction level of a programming language. Domain-specific language features cannot be realized. Parsing code to gain information about what kind of glue code to generate is tedious, and the code also does not have the necessary semantic richness.

Define domain-specific languages that developers use to describe application structure and behavior in a brief and concise manner. Generate the lower-level implementation code from these models. Generate a skeleton against which developers can code those aspects that cannot be completely generated from the models.

A DSL-based programming model marks the entrance into the Model-Driven Software Development arena. Defining DSLs for various aspects of a system and then generating the implementation code – fitting into the PROGRAMMING MODEL defined above – is a very powerful approach. On the other hand, defining useful DSLs, providing a suitable editor, and implementing a generator that creates efficient code is a non-trivial task. So this step only makes sense if the generator is reused often, and the 'normal' PROGRAMMING MODEL is so intricate, that a DSL boosts productivity, or if you want to do complex MODEL-BASED ARCHITECTURE VALIDATION.

The deeper your understanding of the domain becomes, the more expressive your DSL can become (and the more powerful your generators need to be). To manage the complexity you should build cascades of DSL/generator pairs. The lowest layer is basically the GLUE CODE GENERATOR: higher layers provide more and more powerful DSL-BASED PROGRAMMING MODELS.

Model-based Architecture Validation

You now have all the artifact in place and you roll out your architecture to a larger number of developers. You have to make sure that the PROGRAMMING MODEL is used as intended. Different people might have different qualifications. Using the programming model correctly is also crucial for the architecture to deliver its QoS promises. Checking a system for architectural compliance is critical. However, using only manual reviews for this activity does not scale to large and potentially distributed teams. Since a lot technical complexity is taken away from developers (it is in the GENERATED GLUE CODE) these issues need not be checked. Checking the use of the PROGRAMMING MODEL on the source-code level is complicated, mostly as a consequence of the intricate details of the programming language used.

Make sure critical architectural issues are either specified as part of the DSL-BASED PROGRAMMING MODEL, or the developers are restricted in what they can do by the generated skeleton into which they add their 3GL code. Architectural verifications can then be done at the model level, which is quite simple: it can be specified against the constraints defined in the ARCHITECTURE METAMODEL.

This is where you want to arrive. In larger projects you have to be able to verify the properties of your system, from an architectural point of view, via automated checks. Some of them can be done at the code level, by using metrics and so on. However, if the system's critical aspects are described in models, you have much more powerful verification and validation tools at hand.

It is essential that you can use the ARCHITECTURE METAMODEL to verify models/specifications. Good tools for Model-Driven Software Development such as the openArchitectureWare generator [OAW] can read (architecture) metamodels and use them to validate input models. In this way a metamodel is not 'just documentation', it is an artifact used by development tools. The following illustration shows how this tool works.

13.5 Product-Line Engineering

Product-line engineering (PLE) deals with the systematic analysis of domains and covers the design of software production lines. Its goal is to fully leverage the potential for automation and reusability during the development of software systems. Product-line engineering thus seamlessly integrates into the MDSD context as a method for analysis.

In other words, it provides a solid background for all activities in the *domain* part of the domain architecture development thread, as introduced in Figure 13.2.

A comprehensive discussion of PLE would exceed the scope of this book, so we content ourselves with introducing the basic principles and explaining how they tie into MDSD. At the end of this chapter a comprehensive list of further reading is provided. We address the economic and organizational aspects of product-line engineering in Part 4 of this book.

13.5.1 Software System Families and Product Lines

Two key terms, *software system family* and *product line*, were briefly defined in Chapter 4.

The original definition of a software system family is as follows:

> *We consider a set of programs to constitute a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members*. [Par76]

We are faced with a software system family whenever a series of systems is developed – in this context, often referred to as *products* – that have relevant properties in common. In the context of MDSD, these properties are consolidated in the domain architecture. This implies these commonalities can be of an infrastructural, architectural, or functional/professional nature, depending on the nature of the domain.

A product line, on the other hand, consists of a set of functionally-related products with a common target market: its organization is customer group-specific. Ideally, a product line is realized with the help of a software system family.

13.5.2 Integration into the MDSD Process

Let's look at the development process described above. Analytical activities emerge in various places, which we have constructively systematized. These are:

- The definition and boundaries of the domain for which software is to be developed.
- The definition of the most important core concepts of the domain.
- Analysis of the commonalities and differences between software systems of the domain – and thus…
- The separation between application and domain architecture.
- The definition of the most important basic components of the MDSD platform (its solution space).
- The definition of the production process for the software system family.

Product-line engineering provides a methodical basis for systematic reuse in the context of software system families.

MDSD can either be considered an implementation technology for product-line engineering. Similarly, product-line engineering can be seen as an analysis method for MDSD. Product-line engineering practices can and should be used iteratively and incrementally: PLE should not be seen as a separate pre-stage of MDSD, but rather as an accompanying method, even though it is certainly prominent during early phases of an MDSD project.

13.5.3 Methodology

The product-line engineering process consists of three phases – see Figure 13.5. We explain these below.

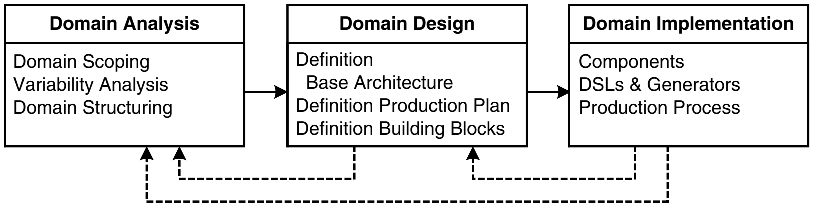


Figure 13.5 The phases of product-line engineering

Domain Analysis

The first step in domain analysis is domain scoping. Here, the boundaries of the domain are determined. If for example, we consider the domain of automotive engine controllers, it is important to define whether this domain includes only gasoline engines or is also suitable for diesel engines, and whether they are used only for personal vehicles or also for trucks. This is important for two reasons:

- Things that are so different that they cannot be reasonably mapped into the context of a family must be excluded to allow a consistent realization. This is a risk minimization strategy.
- Unclear requirements result in on-going discussions during the project. This does not mean that iterative requirements management is prohibited, but merely that the requirements (in this case the scope) should be clearly defined at any given time.

This also explains why it makes sense to use this approach primarily in mature domains: if you don't yet know the differences between the domain products, you cannot make the necessary decisions[2].

Let's return to the first case study of Chapter 3. The domain there was 'application architecture for e-business software'. The car-sharing application is a product of that domain. If you now start with architecture bootstrapping via a reference implementation, you will without doubt only succeed if the architects already have some knowledge about layering, typical J2EE patterns, MVC structures, declarative flow control, and so on. Since this is fairly common knowledge among developers, one can speak of a 'mature' domain here. However, it is not necessary to know the concrete implementation of the architecture's patterns at the beginning of the project. The design language (UML profile) can to a certain extent evolve further in the course of the project (see Chapter 3). Most notably, it can be extended easily.

There are various ways of learning about the differences between the discrete products of the family and documenting them systematically. One powerful method is derived from the FODA method [FODA] and goes under the name of *feature modeling* [EC00].

[2] This may first sound like 'big design up front', but that is not what we mean: we don't say that one analyzes an unknown domain beforehand, but that one knows the domain and its peculiarities from *experience*. This is comparable to industrial mass production: no-one constructs a production line for products that re not yet properly known.

A feature model is graphically expressed by a feature diagram, and shows which features the constituent products of a software system family can or must have. A feature model is a hierarchical structure in which each feature can contain subfeatures. These can be mandatory for a specific product, optional, alternative, or *n*-of-*m*. Figure 13.6 clarifies these terms, showing a (very simplified) feature diagram for the system family *stack*.

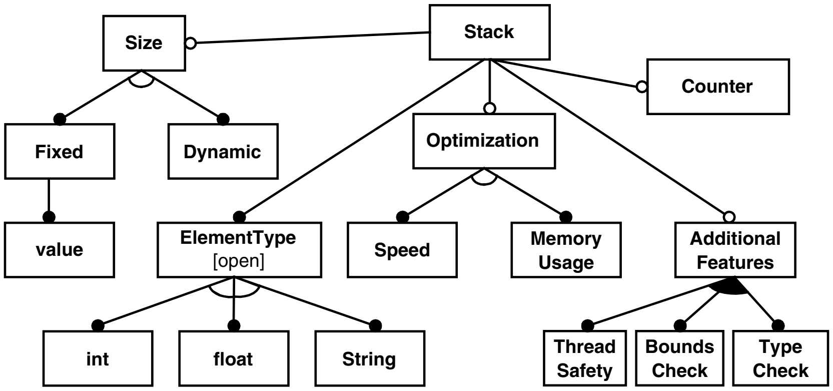


Figure 13.6 Example feature diagram

The diagram describes the fact that each stack must have an element type: ‘must’ is expressed by the filled-in circle denoting a mandatory feature. This can be of the type *int*, *float*. or *string*. This *1-of-n* relationship, also called *alternative*, is recognizable by the open arc between the associations *ElementType-int*, *ElementType-float* and *ElementType-string*. The size of the stack can either be fixed (which means you have to assign a size) or be dynamically adaptable. The stack can optionally have a static counter, shown by an empty circle. If it doesn’t, the size is recalculated each time *size()* is invoked. Further features are thread safety, bounds checking, and type safety. One or more of these features can be contained in one product (an *n-of-m* relationship, shown by the filled arc). Moreover, the implementation can either be optimized for speed or memory consumption.

The diagram in Figure 13.6 therefore describes the ‘configuration space’ for members of the system family *stack*. Discrete members must be built from valid combinations of the features, for example:

- Dynamic size, *ElementType*: int, counter, thread-safe
- Static size with the value 20, *ElementType*: string
- Dynamic size, optimized for speed, bounds check

Apart from the specifications that are visible from the graphical notation, a feature model can contain even more information, such as names for specific combinations of features (macros), the multiplicity of subfeatures, the priority of a feature in the implementation, stakeholders affected by a feature, and so on. Additional constraints that cannot be expressed with the visual notation alone can also be defined. Practice shows that it is possible to go quite a long way in specifying the following additional constraints:

- *Requires*. A specific feature inevitably requires another.
- *Excludes*. The presence of one feature prohibits the simultaneous existence of another.
- *Recommends*. A milder variant of *requires*. A specific feature makes the use of another advisable.
- *Discourages*. A milder variant of *excludes*. One should not use two such features simultaneously.

As an example, here are the two optimization features:

- Optimization for speed: *requires* counter, *requires* fixed size, *discourages* thread safety, *discourages* bounds check.
- Optimization for memory usage: *requires* dynamic size, *discourages* counter.

What is interesting about this method and notation is that they say absolutely nothing about the subsequent implementation of the features. If a system family has already been modelled using UML in this phase, decisions would have to be made regarding inheritance, type genericity, associations, and so on this early in the development. However, this is neither necessary nor helpful during domain analysis. At this point, we are only interested in an analysis of the conceptual commonalities and differences between the products of a software system family. How these variabilities are implemented later is determined during the design phase.

The features shown in Figure 13.6 can even belong to two fundamentally different categories that are not distinguished by the feature model:

- Classic component features define the existence of modular features – that is, whether a product possesses a certain subsystem/module or not.
- In contrast, aspect features are those that cannot be realized as components, for example the features ‘optimized for performance’ or ‘optimized for code size’. Such features may later affect various points in the system and cause different components to be implemented in another way.

Feature diagrams can have both types of features side by side, whereas aspect features especially are very difficult to express with UML.

It is important in this phase to define dependencies between features. A product can maybe only have certain features when it also has certain other features as well. The opposite is also possible, of course – certain features exclude each other. This results in an ordered sequence of decisions that need to be made when building the product. This is essential for defining the production process in the next step. The definition of meaningful defaults in case specific features are not explicitly specified is of equal importance.

You can find further examples of feature models in the second case study, in Chapter 16.

Domain Design and Implementation

The definition of the software structure is one aspect of domain design. One starts by implementing the common features of a domain's products in the form of a platform. Because these common features are identical for all products, it is not necessary to implement them generatively in any way: they constitute the basis of the common target architecture.

For the variable features – those in which the various products differ – one must decide when a feature must be *bound* – that is, the point at which one decides for or against a certain feature for a product. A variety of alternatives exists:

- *At the source code level*. Here, the decision for or against a feature is made during programming – that is, it is hard-wired into the source code.
- *At compile time*. Some decisions can be left to the compiler, such as overloaded functions, preprocessors, code/aspect weaver.
- *At link time*. The linker can be used to configure a product by binding one or another library, for example a makefile that does or does not statically bind specific libraries.
- *At installation time*. For products that contain an explicit deployment step, one can typically still carry out specific configurations. For example, J2EE offers the option of adapting configurations during installation via deployment descriptors.
- *At load time*. You can also intervene while the application is loading. DLLs are one example of this for example optional loading of different DLLs that implement the same functions in different ways.
- *At runtime*. Finally, you can also make decisions at runtime, for example polymorphism, dynamic class-loading in Java, interpreted configuration parameters.

All these options have their advantages and disadvantages:

- *Performance*. Typically the performance of features implemented directly in code is significantly better than decisions made dynamically at runtime. Decisions at compile time usually possess the same level of performance: decisions that are made by the linker or loader are in most cases not much slower.
- *Code size*. If behavior must be changeable at load time or runtime, the code for all alternatives must be present in the program image. The size of the image therefore grows. In this context we have to differentiate between the code size of the image and that of the running program. Decisions made by the compiler are usually quite close to the optimum. However, one must ensure that the compiler does not produce overhead through expansion (a potential risk when working with C++ templates, for example).
- *Flexibility*. Features that are hard-coded cannot be changed unless they are reprogrammed. Flexibility generally increases the later a decision for or against the feature is made – the later it is bound
- *Complexity*. Specific features can easily be removed or reintegrated into the system quite late, such as the dynamic loading of a component. Aspect-like features are more problematic, because they cut across the entire system, or large parts of it. The earlier they are defined, the easier they can be realized.

The production process now describes how an executable product is created from a product specification (the model). MDSD offers one alternative for efficiently and consistently realizing the alternatives just listed.

A generator is the ideal tool for this purpose. At the source code level, this is obvious: the generation of source code is its original task. Aspect features can also be implemented – see Chapter 9 and [Voe04]. Features that are decided at link time can be realized via a generator that creates the respective libraries or a makefile. Deployment can be carried out by the generator by generating a deployment script. The load process can also be influenced generatively. Last but not least, a generator can create (default) configuration files that are interpreted at runtime. The generator can therefore become an integration tool for the software system family.

The generator is configured via the domain architecture (including the DSL). The latter can be divided into subdomains or be partitioned, as shown in Sections 8.3.3 and 15.5. The generator must be able to read the partial models of a system that were modeled with various DSLs and generate a homogenous, consistent system from them. One example of this is provided by the case study in Chapter 16.

13.5.4 Domain Modeling

The domain consists of domain-specific abstractions, concepts, and rules. How can you ensure that the defined DSLs, and thus the modeled applications, constitute a correct subset of the domain?

The key to solving this problem is metamodeling, as described in Chapter 6. The metamodel of the DSL must be restrictive and as close to the domain as possible. Feature models can be very helpful in this regard. The metamodel must, if applicable, ignore unwanted or unnecessary properties of the basic metamodel – for example, that of the UML. Constraints offer a powerful mechanism in this context.

A glossary or an ontology of the domain can be a useful first step towards a suitable metamodel, which is then refined during iterative development (see Section 13.3).

13.5.5 Further Reading

We cannot deal with product-line engineering in sufficient depth in this chapter, so we recommend more comprehensive further reading here.

- P. Clements, L. Northrop*, Software Product Lines – Practices and Patterns* [CN01]. This book, published by the Software Engineering Institute of the Carnegie Mellon University, is a good reference for product-line engineering terminology. The book also provides an overview of classic, not necessarily model-driven product-line engineering processes.
- D. M. Weiss, C. T. R. Lai, *Software Product-line Engineering – A Family-based Software Development Process* [WL99]. This book offers a systematic overview of FAST (Family-oriented Abstraction, Specification, and Translation), a tried and tested product-line engineering method that has been in use for several years now.
- J. Bosch, *Design & Use of Software Architecture – Adopting a Product-line Approach* [Bos00]. This book offers interesting case studies and describes an approach for the architecture development of product lines. The book’s strength lies in the comparison

and evaluation of a number of organizational forms that are considered for product-line engineering.

- C. Atkinson et al., *Component-based Product-line Engineering with UML* [ABB+01]. The book describes the KobrA-approach to product-line engineering as defined by the Fraunhofer Institute for Experimental Software Engineering (IESE). The book focuses on practical standards for component-based development.
- K. Czarnecki, U. Eisenecke, *Generative Programming* [EC00]. Among other topics, this book also contains a good introduction to the product-line engineering topic and offers quite comprehensive information about feature modeling.
- J. Greenfield et al., *Software Factories* [GS04]. Software factories (see also Section 4.5) are Microsoft's approach to product-line engineering. The approach is described extensively in this book. It provides a good introduction to the ideas and concepts, among them also DSLs.

14 Testing

Only in very few cases can software be verified completely based on specifications. Because of this, testing is extremely important in software development.

Up to this point we have not explicitly addressed the role of testing in the software development process. This is not because we believe that testing is unimportant or secondary, but because it (should) essentially play the same role in MDSD as in other methods. Agile processes [Coc01] [Bec02] can serve as role models in this case: testing constitutes a sort of safety net for the modifiability and extensibility of software. This is why tests should not be run at the end of a project, but frequently during the project.

Test automation [HT04] [Cla04] is a key element for enabling continuos and reproducible validation of the software during its construction. We don't want to begin with Adam and Eve in regard to testing, but rather to concentrate on specifics in the context of MDSD in general and architecture-centric MDSD in particular.

We mentioned in the previous chapter that we divide a project into an architecture development thread and an application development thread. Both threads are synchronized via milestones; yet they produce different artifacts: The application development thread yields an application, the architecture development thread a domain architecture. We begin with the aforementioned, but before we get started, we wish to give a brief general introduction to the different types of tests.0.3 (final)

14.1 Test Types

The discussion of which aspects of a software system are to be tested, when, and to what end, leads us to *test types*:

- *Component tests* (also called *unit tests*) are created by the developer in the course of their every-day work and serve to ensure the correct functioning of system parts (classes, modules, subsystems and so on).
- *Integration tests* ensure the correct interplay of existing parts of the system.
- *Acceptance tests* show whether the system makes sense from the customer's viewpoint and meets their requirements.

- *GUI tests*[1] can either take on the role of acceptance tests (ergonomics) or of integration tests.
- *Non-functional tests* validate requirements with the target architecture, such as security, correct transaction management in multi-user environments, or robustness towards hardware failure.
- *Load tests* are a special subset of non-functional tests. They serve to measure performance (response times, data throughput) and scalability. Load tests are often quite sophisticated and thus only carried out at specific times in the course of a project.
- *Regression tests* are used to detect whether unwanted side-effects have emerged due to changes to the source code (additions or corrections). Regression tests are usually realized through repeated execution of other test types.

Test automation is a very important means for increasing the efficiency and effectiveness of tests. Only reproducible, automated tests can form the 'safety net' that allows software to stay changeable. Test automation helps to implement regression tests and is applicable for all test types except for tests of ergonomics. The most sophisticated kind of test automation is *continuous integration* [Cla04] [CRUI]. Here automated tests are carried out almost continuously in parallel to development on special test machines that inform the developer asynchronously about test failures, for example by e-mail.

In the context of MDSD, the domain architecture – due to its generative aspect – plays a similar role to a compiler. However, its correctness cannot always be taken for granted: more about this issue later. On the other hand, however, MDSD offers good opportunities for simplification of the test types listed above for the application development thread. This is what we will address in the following section.

14.2 Tests in Model-Driven Application Development

In the application development thread the same test types as in non-model-driven processes are relevant. The models are part of the code because they are formal and automatically transformed into 3GL code via transformations. From this perspective, the DSL is a programming language (with affinity to the domain) that can have semantic gaps. After generation, you will have exactly the same code as you would have had if you had not used MSDS, and so the code can be tested in the same manner. Even a test-first approach does not constitute a contradiction here: nobody keeps the developer from writing test code *prior* to modeling if this is considered sensible.

The model-driven approach has a lot of potential that can also simplify the creation of test code. A black-box test is always a comparison between an *is*-state and a desired state: we do not differentiate between return values of methods and side-effects such as database changes. Thus a test secures the semantics behind a syntactically-defined interface[2]. In other words, the test 'knows' the signature and the semantics of the system to be tested – this can justifiably be called a *design*. In effect, a developer who writes test code specifies design information in doing so. Equally, no developer can write test code without having at least a rough idea of the design of the system to be tested.

[1] Tests of graphical user interfaces (GUIs).

[2] This can either be a Java interface or a GUI, depending on the test type.

In the context of MDSD, we now possess a more valuable means than a 3GL language to express designs: the DSL. Depending on the semantic depth of the DSL, it can be much more effective to specify design information first in the model rather than right away in the form of 3GL test code, in order to generate exactly that code or parts of it. The advantage of this approach lies in the abstraction and the easier changeability of the test code: a part of it is moved into the domain architecture and can now be centrally maintained and modified. The domain architecture is thus extended to include the generation of *testware*. Whether this procedure is efficient (and if so, to what extent) depends very much on the expected degree of reusability.

14.2.1 Unit Tests

In the context of conventional software development, small but useful frameworks such as *JUnit* (or *C#Unit*, *CppUnit*, and so on) are available. We want to continue to use these tools in model-driven development, and support the creation of tests as efficiently as possible with MDSD techniques.

Separation of Test Cases and Test Data

Unfortunately, unit tests are in most cases programmed in such a way that test cases and test data are mixed in the code. At first glance this seems to simplify matters, but in fact it harbors serious disadvantages for the further progress of the project:

- Test cases are not reusable for different test data.
- Test cases and test data lose their maintainability.
- Test data can only be modified or provided by the developers.
- The actual structure of test cases is obscured, which impairs readability.

It is therefore a good idea to separate test cases and test data when implementing component tests. This can be done for example by rendering the test data in the form of XML documents or Excel files, from which they can be read by the actual test case – for example a JUnit test method – at runtime. This not only works for the test input (the test stimuli), but also for the reference input.

In the context of MDSD, this separation also allows for dealing separately with test case generation. For example, you can only support the test cases generatively and solve the integration of the test data generically using a suitable framework.

The external format for formulating the test data becomes a part of the DSL's test partition – or in other words, test data can be MDSD models, regardless of whether they are transformed or directly interpreted by the (test) platform.

Test Infrastructure Generation

The simplest and most obvious support that MDSD offers for component tests is the generation of test infrastructure from the models. This can for example include (empty) test methods for all business operations of the application, as well as their compositions for test suites. One can also generate a default implementation into the test methods that causes the tests fail, thereby enforcing a real implementation by the developer.

Constraints Checking

The generation of test cases based on constraints in the models is a very powerful technique[3]. These declaratively specify properties of the system, but do not yield any information on *how* these properties are realized. Typically for such constraints one distinguishes between preconditions to describe the prerequisites for invoking an operation, post-conditions to describe the state of the system after the operation has been executed, and invariants to describe such constraints that must always hold during system execution. If you have a suitable model, you can generate a unit test that creates the required test setup, provides valid test data (precondition), invokes the operation to be tested, and afterwards checks the postcondition. In this context, the following questions arise:

- How can the set-up be created?
- How should the test data be composed?
- What environment must be present to enable the test to pass?

Since this is not always easy to define and implement, it often makes sense not to check the constraints using a special unit test, but simply to generate the constraint checks directly into the system, and also check the constraints during the integration tests. Figure 14.1 shows an example:

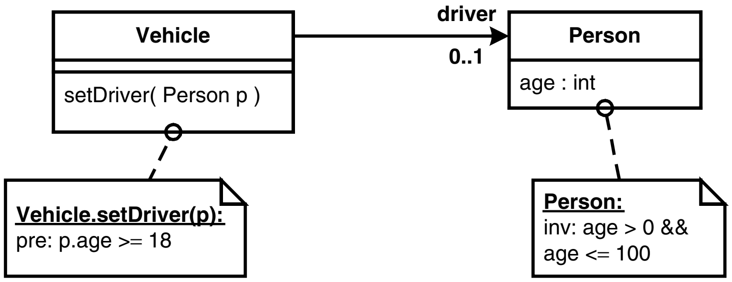


Figure 14.1 An example of a class diagram with constraints

It is actually possible to generate code from these constraints:

```
class Vehicle {
  ...
  public void setDriver( Person p ) {
    if (p.getAge() < 18 )
           throw new ConstraintViolated();
    // ... implementation code ...
  }
}
```

[3] These should not be mistaken for modeling constraints at the metalevel (see Chapter 6) which define valid models in terms of the DSL.

Two things need to be explained. First, the question arises of how one generates code from OCL constraints: suitable tools are available, for example [EMPO], [DRES]. Then there is the question of where and how the constraints can be integrated into the model and whether they would be available for subsequent tools, for example in the XMI export. A trick that works with all UML tools is to abuse the model elements' documentation: for example, you can insert the following text for the *setDriver()* operation here:

```
The purpose of this operation is to assign a driver to a vehicle. Drivers are
always Persons aged 18 or older.
<Constraints>
  <pre id="driverAge">
    <expr>p.getAge() >= 18</expr>
    <error>Driver must be 18 or older.<error>
  </pre>
</Constraints>
```

In the MDSD transformation the documentation text can simply be searched for the *<Constraints>* tag. The XML can be parsed, and thus the constraints are ready for further processing. To simplify matters, we assume that the text inside *<expr>…</expr>* is a valid Boolean expression *in the target language*. This limits portability, but it is feasible if you need a pragmatic approach.

Another interesting question is how you realize this approach if the implementation of the actual method is done manually. How can you make sure that the constraint is nevertheless *always* checked? Here, the use of the Template Method pattern is advisable [GHJ+94]. An abstract class *VehicleBase* is generated that implements the operation *setDriver()* as follows:

```
abstract class VehicleBase {
  ...
  public void setDriver( Person p ) {
    // generated from precondition constraint
    if (driver.getAge() < 18 )
        throw new ConstraintViolated();
    setDriverImpl( p );
    // generated from postcondition constraint
    if (driver == null) throw new ConstraintViolated();
  }

  protected abstract void setDriverImpl( Person p );

}
```

This operation invokes an abstract operation *setDriverImpl()*. The developer now writes the class *Vehicle* manually and implements only the abstract operations defined by the superclass:

```
class Vehicle extends VehicleBase{

  protected abstract void setDriverImpl( Person p ) {
    driver = p;
  }

}
```

One can define the constraints with any required degree of complexity, for example via a protocol state machine that can be assigned to an interface. It is possible to generate code from it that will check the constraints expressed with the protocol state machine.

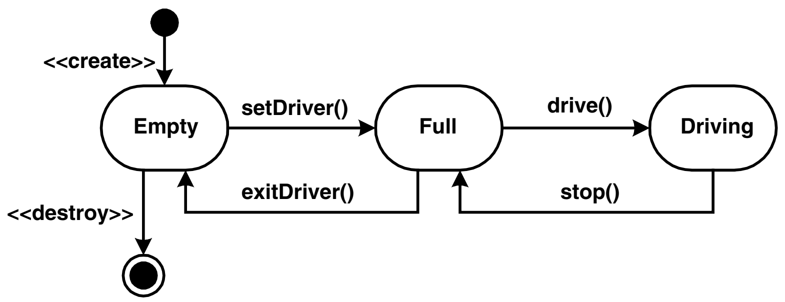


Figure 14.2 An example of a protocol state machine

The state machine in Figure 14.2 states, for example, that you can only start to drive (*drive()*) if a driver is seated in the car. The generated code can automatically track these states. Should the *drive()* operation be invoked while the vehicle is empty, a *ConstraintViolation* exception can be thrown.

Generation of Extreme Value Tests

If no constraints are given for an operation, of course no functionally useful tests can be generated automatically: they must be implemented by hand. However, you can see this situation differently: if no constraints are stated, this means that any input parameter (combination) is valid for the operation. This means that you can generate test cases that use arbitrary test data. You cannot verify the result of the operation (since you don't know what it does) but you can check that it does not fail fatally, for example with a null pointer exception. This is a practicable approach, especially for operations with primitive types. Common values for *int* parameters are for example 0, -1, 87684335: for object parameters, null is recommended. Note that this approach does not replace real tests that check that the operation performs its task correctly, but the approach does hint at typical programming bugs such as not checking for *null* values or negative numbers.

Mock Objects

Mock objects are a useful instrument for software testing. They serve to decouple test code and application infrastructure that is relevant from a testing perspective. To this end, the corresponding interfaces are implemented by mock objects that behave as required for a test scenario, even without a complex infrastructure. Let’s assume we want to check whether a client treats the exceptions thrown by a remote object correctly:

```
public Object doSomething( RemoteObject o ) throws SomeEx {
  try {
    return o.anOperation();
  } catch ( RemoteException ex ) {
    log( ex );
    throw new SomeEx();
  }
}
```

How can we now make sure that the *RemoteObject* really throws an exception for test purposes when the client invokes the operation *anOperation()*? Of course, we can implement the interface *RemoteObject* manually and throw a corresponding exception – but for other test scenarios, we have to implement the same thing again and again.

In addition to generic approaches, such as EasyMock [EASY], code generation can also help here. For one thing, you can very easily generate implementation classes with it that demonstrate a specific behavior. As a specification, you can use the following XML code:

```
<MockConfig type="examplepackage.RemoteObject"
            name="RemoteObjectTestImpl">
  <operation names="anotherOperation, thirdOperation">
    <return value="null” occurrence="1"/>
  </operation>
  <operation names="anOperation">
    <throw type="RemoteException” occurrence="all"/>
  </operation>
</MockConfig>
```

This specifies that a class *RemoteObjectTestImpl* should be generated that implements the *examplepackage.RemoteObject* interface. The operation *anotherOperation* and *thirdOperation* are implemented in such a way that *null* is returned. For *anOperation* it is defined that a *RemoteException* is thrown. This class can now be used in a test.

You can also define such mock objects implicitly, which requires even less effort – see Figure 14.3):

Here a *RemoteObject* is created. We specify that *null* is returned on the next method call (the implementation code for the method is simply placed in braces). We check if the *doSomething()* method returns this correctly, then we define on *RemoteObject* that it throws an exception when the next invocation occurs. Next, we invoke *doSomethingElse()* on the client and expect it to throw a *SomeEx* exception.

To implement this scenario, the code generator can simply generate a matching implementation of the *RemoteObject* interface that behaves as specified in the diagram.

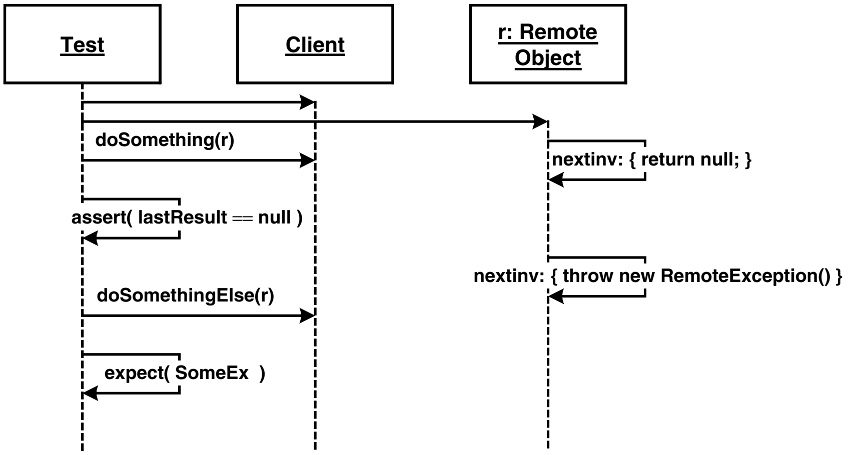

Test
Client
r: Remote Object
doSomething(r)
nextinv: { return null; }
assert(lastResult == null)
nextinv: { throw new RemoteException() }
doSomethingElse(r)
expect(SomeEx)

Figure 14.3 An example for the use of mock objects

14.2.2 Acceptance Tests

These tests *must* be defined independently of the application model. It would be useless to generate acceptance tests from the same model from which the application is generated, because in that case it would *never* fail, as long as the domain architecture is correct. This leads us to the two-channel principle: one needs a second, independent 'channel' to specify tests that are independent of the original specification. However, the second channel can also be supported by MDSD – see Figure 14.4.

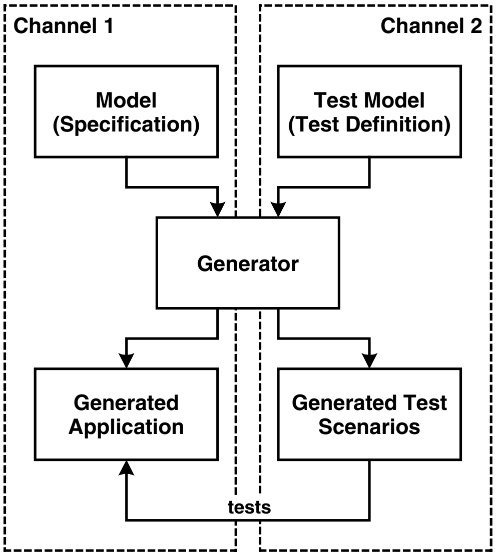


Figure 14.4 An examples of the two-channel principle

As you can see, it is possible to model test scenarios in the same manner as the application itself, but in a separate acceptance test model. Constraints on operations of the test scripts provide the validation rules that are needed in the tests.

When a suitable DSL is used, a domain expert can understand the test cases *much* better and work more productively on their definition compared to programming the integration tests. Figure 14.5 is an example of this that uses a sequence diagram. First, the test client creates a new *Vehicle* and a new *Person*. It then sets the person's age to twenty and defines the person as the driver of the car. An *Assertion* follows that ensures that the driver of the car is exactly the one that has just been set. Then a new person is created who is ten years old. If this new person is set as a driver, a *ConstraintViolated* exception must be thrown that is evaluated with *expect()*. This guarantees that the implementation code actually checks the respective constraint. Finally, we verify that the driver is still the original driver.

Of course one would separate the test data from the test code, as mentioned above. We omitted this here to keep our example simple.

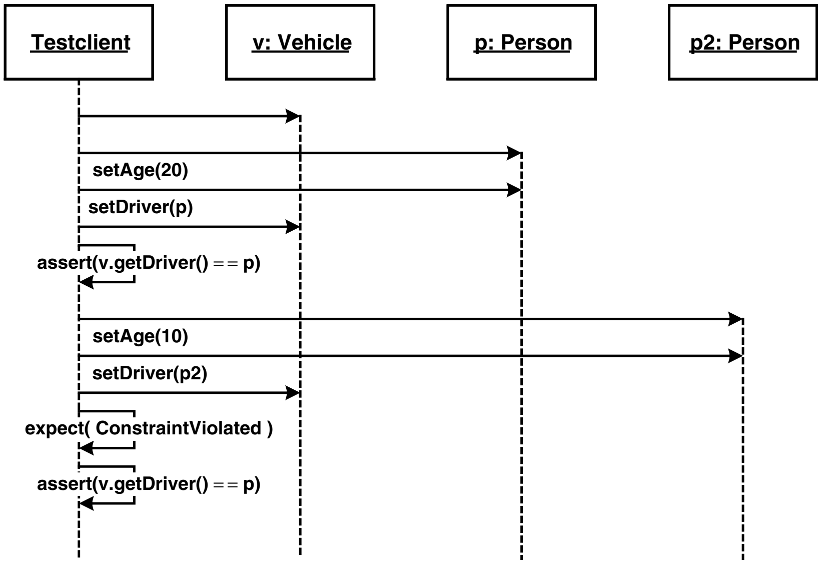


Figure 14.5 An example for the definition of a test scenario using a UML sequence diagram

14.2.3 Load Tests

In principle load tests follow this schema: a set of clients that usually run on a number of computers are simulated. These clients run test scripts. The clients' timing behavior is measured relative to the load – that is, the number of clients and their call frequency. The server's resource consumption is also logged.

The environment's setup (for example, server, network, databases) must be created manually. The following aspects must be defined:

- The test script(s)

- The number of clients
- Their internal parallelism, that is, the number of processes and threads

In most cases the last two aspects are handled by tools that also measure the timing behavior and monitor the application(s). The test scripts are well-suited for generation. A sequence diagram or a state chart constitute a good basis, as can be seen in Figure 14.6.

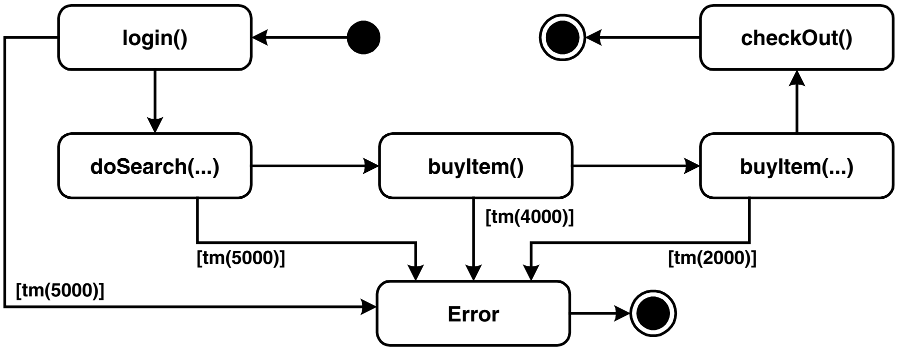


Figure 14.6 State diagram with timeouts for load test definition

14.2.4 Non-functional Tests

Testing non-functional requirements such as reliability, transactional consistency, or deadlock freedom cannot be done in a simple environment, such as in JUnit. Realistic scenarios are needed here. Failures of networks or databases must be simulated, or forced to actually occur. Similarly, security tests can only be conducted manually. Model-driven development does not offer any specific help here.

14.2.5 Model Validation

Model validation is a type of test that is possible only when using MDSD. It offers totally new options for testing. We must distinguish between three different subtypes:

- *Acceptance tests on the model level* to validate the model semantics
- *Well-formedness tests* to check whether the modeling rules are observed (DSL constraints)
- *Simulation of models*

Acceptance Tests on the Model Level

MDSD models have the potential to be validated in direct communication with customers or experts (unless they create the tests themselves), particularly if the MDSD domain and its DSL are business-oriented. This approach can provide additional certainty, especially when code or configuration files are created from the model. The meaning of this code or configuration can thus be checked beforehand on a more abstract level.

Well-Formedness Tests

We have already studied the importance of modeling rules in the context of MDSD from the example of the first case study in Chapter 3. We also saw that such rules take on the role of the DSL's static semantics (Chapter 4) and showed how they are defined in the context of metamodeling (Chapter 6).

From the test perspective, such modeling rules are invariants that are valid for all instances of a DSL – that is, for all models. They *must* be checked before the actual MDSD transformations can take place, otherwise the transformation result is undefined. The well-formedness test can be carried out by the modeling tool (if it is able to) or a by subsequent MDSD tool – see Sections 3.2 and 11.1.2. From a technical standpoint, these tests are nothing more than a check of the static semantics of a classic programming language by the respective compiler. However, from the developer's point of view, they are *far* more effective: the compiler of a classic programming language doesn't know anything about the domain, so it can only issue error messages about the solution space – the programming language. In contrast, modeling rules are specifically created for the domain and can therefore report error messages using the terminology of the problem space.

A specific class of errors that in conventional development are typically not detected even by the compiler, but only show up at runtime, can be detected very early with the MDSD approach – during modeling or prior to code generation. Here is a simple example: the metamodel in Figure 14.7 defines that a configuration parameter (a special type of attribute) that must always have the type *String*.

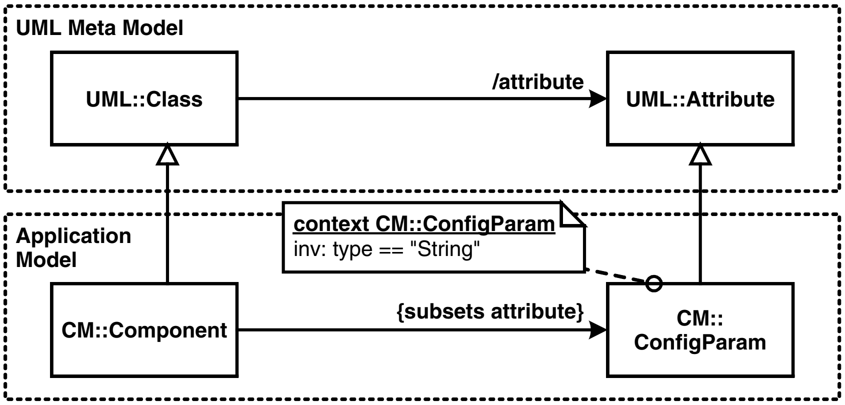


Figure 14.7 An example of constraints in the metamodel

For all models that are specified using this DSL, the given constraint must hold. If a configuration parameter is not of type *String*, a corresponding and meaningful error message can be issued, such as *'ConfigParam must be of type String'*, instead of getting an exception at runtime because somewhere somebody tried to assign an *Integer* to a *String*.

Simulation of the Model

If the dynamic aspects of a system are thoroughly described in a model, they can be validated by simulation of the model – that is, by execution of the model in a test bed. This approach is quite popular for embedded systems, but only for rather specific types of behavior definition, specifically finite state automata. The UML 2.0 and action semantics are also a step in that direction – see Chapter 12. In general practice, the model simulation approach is either not economical or not possible because the dynamic system behavior is not specified in the model.

14.3 Testing the Domain Architecture

The domain architecture is software, too, and thus must be tested adequately. Fortunately, one can break down this problem into single aspects for which well-known testing solutions are available.

14.3.1 Testing the Reference Implementation and the MDSD Platform

The reference implementation plays a central role – at least during the bootstrapping phase of the domain architecture. Since one typically uses a small expert team for this purpose, agile, test-driven methods are quite useful here. The same is true for the MDSD platform.

All the test types introduced in Section 14.1 can be applied here – of course first without model-driven, generative support. Nevertheless, the reference implementation can contain a partition for prototype testware to discover its potential for generation.

14.3.2 Acceptance Test of the DSL

The validation of the modeling language (DSL) is also very important. This happens through its use in the reference model (see Section 13.2.2). The latter is primarily an (acceptance) test of the DSL in terms of its manageability and ergonomics. Since the reference implementation and the reference model try to cover all of the DSL's constructs as minimalistically as possible – the minimum of testing that completely test every feature once – the reference model should be considered a rather significant test. After bootstrapping, the application model from the development thread is the next DSL test – you can use the 'real' application models as a test case for the suitability of the DSL.

14.3.3 Test of the MDSD Transformations

In the architecture development thread, the formalization of the reference implementation's aspects and its casting in a computer-processable form – mostly generator templates or similar transformation rules that are bound to the DSL's metamodel – is accomplished. As a rule, the transformation rules build on a generic MDSD tool whose correctness we assume for our purposes[4]. This leaves us only with the test of the domain- and platform-specific transformations. How can this be carried out in a sensible manner?

A fairly obvious test method is a by-product of bootstrapping the domain architecture: since the transformation rules were derived from both reference implementation and reference model, they should be able to reproduce exactly that part of the reference implementation that is covered by the DSL. In other words, if you apply the newly-created transformations to the reference model, you get – depending on the scope of the architecture – either the complete reference implementation or just its implementation skeleton. If you complement this skeleton with code fragments from the original reference implementation, a complete and testable application should be the result. All the tests that were created during the reference implementation's construction should still work successfully! As a side-effect, one gets a generative reference implementation, and the bootstrapping of the domain architecture is finished.

In the further course of the project, this initial test of the domain architecture is extended for testing the actual application. The architecture is thus implicitly validated by the applications created – that is, by their tests. This is extremely effective and totally sufficient in practice.

What would an explicit test of the transformation rules look like? After all, a generator is a metaprogram – a program that generates programs from models. Its inputs are models, its outputs are programs or program parts (source code fragments). Thus an explicit test of for example a single transformation rule would use a relevant piece of a model as the test set-up, apply the rule to it, and finally compare the result to a specification: that is, to the corresponding desired generated code or source code fragment. Figure 14.8 explains this principle.

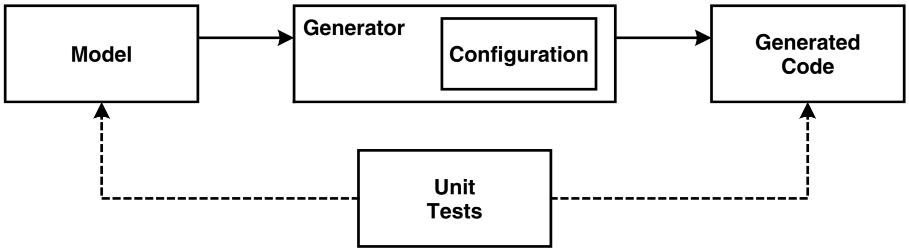


Figure 14.8 An explicit transformation test

When the generators are constructed from modules, such an approach, and the construction of respective test suites is entirely possible. Let's consider the consequences.

A (black-box) test supplies the system under test with specific stimuli (set-up and parameters) and compares the output or side-effects of the executing system with a specification. Abstraction takes place from the implementation of the system to be tested. How does abstraction work in the case of the explicit transformation tests described in this section? Is it helpful to commit to a specific and textually fixed version of the generated code, or aren't the semantics of the generated

[4] Our scenario would look different if the generator needed to be certified, for example for a security-critical system. However, we do not address such a scenario.

code more relevant? Imagine a test suite with explicit transformation tests for a domain architecture in practice: what happens if the architects realize that another implementation is much better suited for a generative software architecture aspect than the existing one? The test suite would not allow the change – even though it would perhaps be totally transparent from the client applications' perspective! Equally, this means that explicit transformation tests constitute a counterproductive over-specification. Transformation tests should *not* take place on the metalevel (testing of the generator), but on the concrete level (testing of the generated code).

What about modularization of transformation tests? The reference implementation (including its concrete test suite) and the reference model provide one rather powerful test case for generative software architecture, while each client application provides another. Depending on the combinatorics of the modeling constructs, it can make sense to generate a test suite from a number of smaller test cases for the domain architecture. This can work as follows: a transformation test case focuses on a specific construct of the DSL (for example, associations between persistent entities) – that is, the set-up consists of a model that is minimal in this respect and if necessary also contains pre-implemented domain logic. The test run includes running the transformations, resulting in code generated from the model that embeds the existing domain logic, if applicable. The runnable generated code is then validated on the concrete level via test cases. A test suite from such transformation tests that covers the DSL validates the generative part of the domain architecture.

15 Versioning

Efficient versioning and configuration management are always important in software development projects. In this chapter we discuss only those MDSD-specific characteristics of this topic.

15.1 What Is Versioned?

In an MDSD project, the following aspects must be managed or versioned:

- The (generic) generation tools. Currently, these tools are often still in development themselves – because of this, it makes sense to have them under version control.
- The generator configuration, the generative part of the domain architecture. This includes the DSL/profile definition, metamodel, templates, and transformations.
- The non-generative part of the domain architecture, the MDSD platform.
- The application itself: models, specifications, and (in most cases) manually-developed code.

The generated code ideally is not versioned, because it is reproducable from the model at any given time, and thus does not constitute a real program source. Of course, this idea can be applied sensibly only if the manually-created and the generated code are separated structurally in the file system. This is one reason why we value this separation (see Section 8.2.6). In practice, it is not always 100% possible or useful, so we sometimes need a more complex procedure for such cases. We discuss this later in this chapter.

One of the goals of model-driven development is the development of several applications based on the same domain architecture. It is therefore essential to separate the platform and the generator configurations from the applications completely.

15.2 Projects and Dependencies

Figure 15.1 provides an overview of a tried and tested project structure. The dashed-lined arrows indicate the dependencies. The main goal is that the domain architecture must be kept clear of application-specific artifacts.

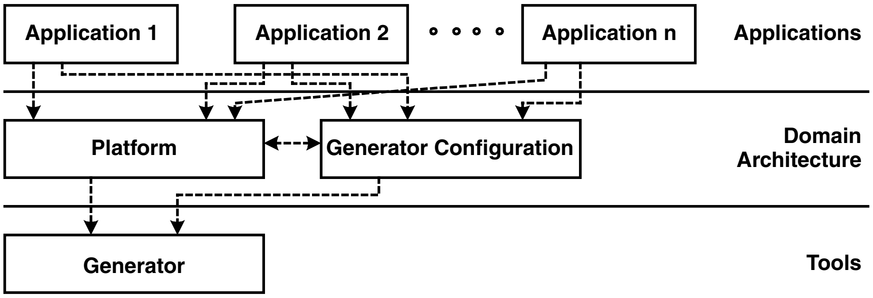


Figure 15.1 Projects and their dependencies

All the projects shown in Figure 15.1 are managed in the version management system. The tools and the domain architecture are checked in completely, but not the applications: the generated code that is created as the result of running the generator should not be checked in.

It is important to manage the dependencies of the various projects, including their versions. For an application project, it is essential to specify *on which version* of the domain architecture the project depends. If the underlying platform evolves, you might have to co-evolve the domain architecture, and maybe even the application projects. A framework metaphor is useful to clarify this: imagine the domain architecture as a framework. If you evolve the framework you have to adapt its client applications. These are the same dependencies, so the same methods are applied in the evolution of a domain architecture.

It is worth mentioning that this view can also cascade. For example, it makes sense to version the MDSD platform and the generator configuration in a domain architecture separately, especially when the platform is reused in other domain architectures. On top of this, one will want to version reusable transformation modules (cartridges) separately, and a powerful platform may decompose into a number of decoupled frameworks. In an even more advanced scenario, a functional/professional MDSD platform might have been created with the aid of an architecture-centric domain architecture. These dependencies must be recognized and considered, both in versioning and in the context of architectural dependency management.

15.3 The Structure of Application Projects

Figure 15.2 shows how an application project can be structured at the highest level and how the generator and compiler work on it.

The models of the application, as well as the manually-created code, are located in the application repository. The generator creates the generated code, including configuration files and so on, supported by the generator configuration. The latter is located in the domain architecture repository. The application is next generated with the help of the build script (in most cases this is also generated). This step uses the manually-created code of the application and the platform, the latter taken from the domain architecture repository.

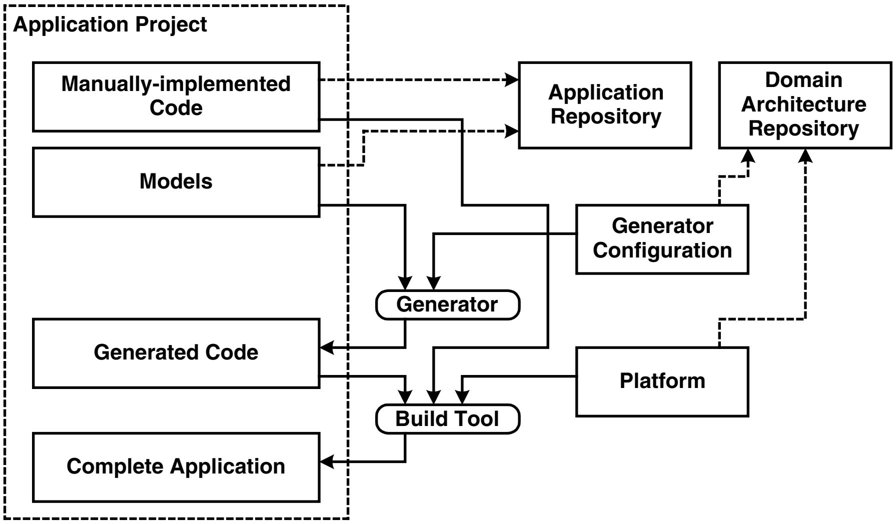


Figure 15.2 Projects, artifacts, and repositories

15.4 Version Management and Build Process for Mixed Files

A complete separation of generated and non-generated code is not always possible or sensible. Examples of this are:

- Custom code in J2EE deployment descriptors.
- Custom code in JSP files.
- Custom code in property files or other configuration files.

In general, these are locations where the target language does not provide sufficient delegation mechanisms.

In these cases one would as a rule work with mechanisms outlined in the case study in Chapter 3: protected regions that can be defined in the generator templates. As a consequence, markings are created in the generated code that are used by the generator during iterative regeneration, to find and preserve the manually-created code contained in it. These markings are hidden syntactically from the compiler or interpreter of the target language by labeling them as comments.

Obviously the use of such protected regions leads to files in which generated and non-generated code is mixed. The problem here is that these files can usually only be versioned as a whole. This results in redundant code being checked in, because the generated code (without the contents of the protected regions) is, after all, not source – the source would be the (partial) model from which the code was generated.

These redundancies can lead to inconsistencies during development in a team. The inconsistencies will become increasingly problematical as the team grows larger: for example, assume that developer A changes something in the model or the architecture while developer B is programming domain logic contained in a file whose generated portion is affected by A's changes. This situation can cause a problem or conflict when checking in either A or B, because the data is no longer up-to-date. The reason for this is a redundancy in the repository. The objective must be to avoid this redundancy and for example to manage the contents of the protected regions in isolation: the generator must allow this. On the other hand, an isolated protected region is usually of little help to the application developer, because they need the context of the generated code as guidance and to execute the compile-run cycle.

In this situation we need a procedure that avoids redundancy in the repository *and* offers the developer their familiar, file-oriented view. Figure 15.3 shows such an approach.

The real (that is, non-generated) application sources are managed in the application repository. Manually-created, application-specific source code that is *not* organized in protected regions is labeled *separate code* here. In addition, one can automatically – and frequently – create an *application image*, for example via CruiseControl [CRUI], that contains the generated as well as the manually-created code. Automated tests can then also be initiated on the server.

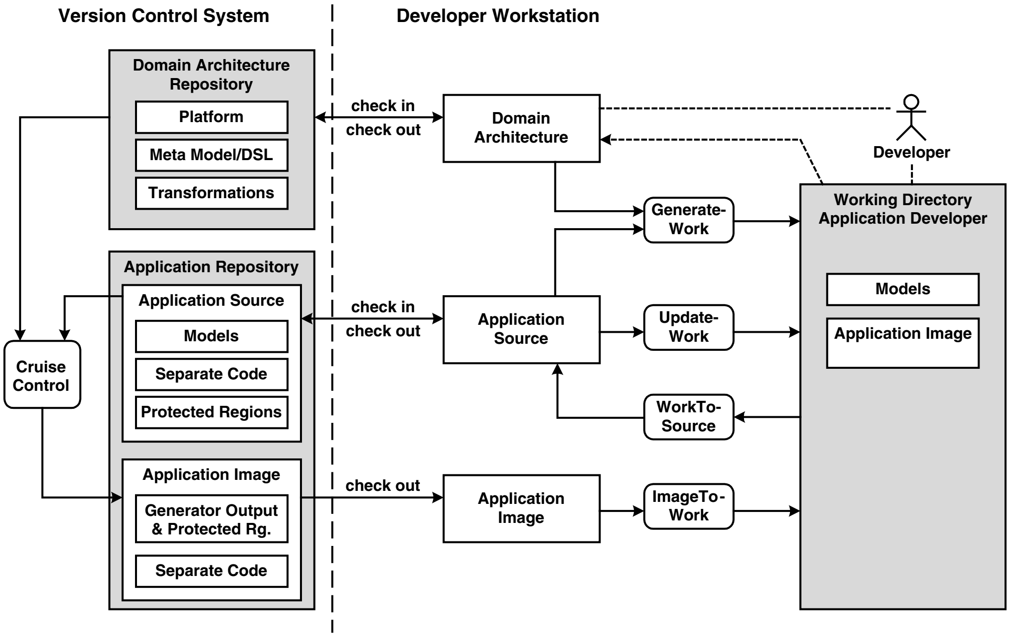


Figure 15.3 Version management and build process for mixed files

The developers can now use the well-known *check in* and *check out* processes of version management to achieve synchronization with the repositories or modules. The actual application development takes place in a separate *work directory*. The synchronization between this work directory and the (local) directories that are also controlled during versioning takes place for example via the following scripts or Ant tasks:

- *GenerateWork*. This script applies the domain architecture on the application model and integrates the checked-out handwritten code from separate files and protected regions with generated code through a generator run. In other words, it produces a complete source code image of the application in the developer's working directory from scratch.
- *UpdateWork*. Other than with *GenerateWork*, no generation takes place here. Only the handwritten code is updated
- *WorkToSource*. Changes to the working directory are reduced to changes of real sources (generated code is not source) and made available to the version management system, so that check-in can take place in a way that is compatible with the structures in the version control system.
- *ImageToWork*. The result of this script is not very different from *GenerateWork*. It refreshes the complete source code image in the working directory from the version control system, where it was previously built by a continuous integration server such as CruiseControl. This saves local processing power by delegating a complete build to the server side. Obviously only checked-in content is involved here, in contrast to *GenerateWork*.

Merge conflicts between developers are exclusively detected using application *sources* in the repository.

It should be emphasized again that a separation between generated and non-generated code is preferable, in our opinion, and should in any case be attempted.

15.5 Modeling in a Team and Versioning of Partial Models

Big systems must be partitioned. Their constituent parts or subsystems are developed more or less independently of each other. Interfaces define how the systems will interact. Regular integration steps bring the parts together. Such an approach is especially useful if the parts are developed in different locations or by different teams. Of course this primarily affects the development process and communication in the team, possibly also the system architecture. This section casts light on various aspects of this process in the context of versioning.

15.5.1 Partitioning vs. Subdomains

First it is important to point out the difference between partitioning and the use of subdomains (see Figure 15.4):

- (Technical) subdomains isolate various aspects of the whole system. Each subdomain has its own metamodel and a DSL. The different metamodels are conceptually unified via gateway metaclasses. In the context of an enterprise system, these could be, for example, the subdomains *business processes*, *persistence* and *GUIs*.

- In contrast, partitioning describes the definition of partial systems. For reasons of efficient project organization or complexity, a large number of technically-similar requirements is broken down into separate parts that can be integrated with interfaces.

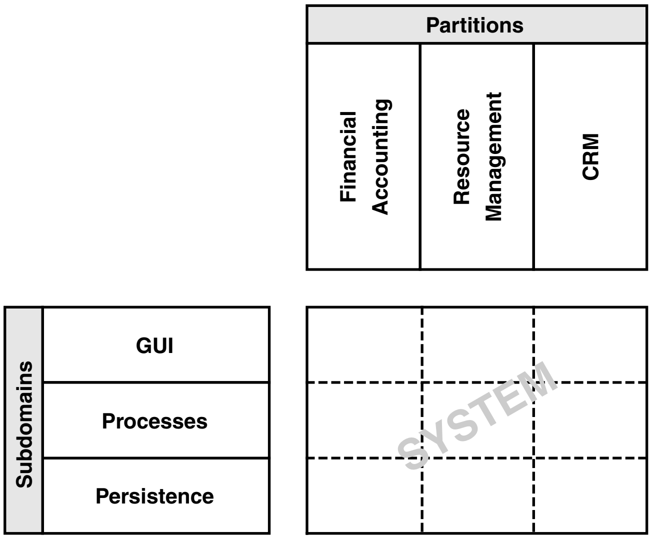


Figure 15.4 Technical subdomains contrasted with partitioning in a financial example

15.5.2 Various Generative Software Architectures

If different (versions of) generative architectures are used in different partitions of a project, the question arises of whether the generated artifacts will work together. In general, this means that an integration should take place at the level of the generated code. As an example, assume we work with different versions of generative infrastructures that all create parts of a comprehensive J2EE application. Since all generated artifacts must be J2EE-compliant in such a scenario, integration can take place at the level of the finished applications: it is not mandatory that all parts work with the same domain architecture. Such an approach is not ideal, of course, since overall system-wide constraints checking is not possible.

15.5.3 Evolution of the DSL

The DSL typically continues to be developed in the course of one (or more) projects. The knowledge and understanding of the domain grows and deepens, so the DSL will thus be extended. To make life simpler, one must make sure that the DSL remains backwards-compatible during its evolution.

The approach to versioning in Section 15.5.2 is one way of accomplishing that. Another option is to modify the generator configuration in such a way that it supports different versions of the DSL and metamodel. This should be considered particularly if you work with different versions of the DSL in parallel, for example when application developers switch to a new version of the domain architecture at the end of an iteration. Now a newer – in most cases more powerful – version of the domain architecture is offered, while the application models are still using the previous version. In this case, you must provide a migration path that requires as little change as possible on the model from the application developer's viewpoint. The new features of the DSL should be offered to the developers, not forced on them. Older features might also become deprecated: the generator could issue warnings when such features are used.

In practice, the support of domain architecture versions using generator configurations is not as complicated as it may sound at first. After all, the generator configuration is implemented by the developers of the domain architecture. They define the metamodel and the concrete syntax, as well as the code to be generated. They can, for example, place a version number in the models[1] that determines how the generator interprets the model or what code it generates. You can also implement implicit rules: if a certain attribute in the model does not exist, a specific default value is then used. Another example is the validation of attributes of entities. Let's assume you have a metamodel that contains the concept of an entity. An entity owns a number of attributes. The following listing shows an example model rendered in XML:

```
<Entity name="Person">
  <Attribute name="name" type="String" label="Name"/>
  <Attribute name="firstname" type="String"
             label="Firstname"/>
</Entity>
```

Typically you want to check the attributes for correctness. For this purpose, you can state various constraints. Initially, you might implement this as follows, by simply annotating a named constraint:

```
<Attribute name="name" type="String" label="Name"
           constraint="notNull"/>
```

This asserts that the attribute cannot be left empty. In the course of the project, you next learn that you need more than one constraint, which cannot be expressed adequately with XML attributes. Instead of changing everything, you can now allow the additional constraints as additional XML elements – the old attribute variant can remain:

```
<Attribute name="name" type="String" label="Name"
           constraint="notNull">
  <Constraint name="startsWithLetter"/>
</Attribute>
```

[1] A portable, tool-independent option of placing versioning information in the UML model allows the model elements to be enriched with the respective tagged values.

Now you may find out that you need an optional Boolean expression. This can for example be formulated using the target language:

```
<Attribute name="age" type="int" label="Age"
           constraint="notNull">
  <Constraint>
    1 <= age <= 110
  </Constraint>
</Attribute>
```

If this flexibility is still not enough, you can also state a class name in the target language while the respective class implements a validator interface defined by the platform:

```
<Attribute name="age" type="int" label="Age"
         constraintChecker="person.AgeChecker"/>
```

If you continue to develop your DSL in such a way that you keep the old features and add new ones, an evolution of the DSL is easily accomplished in everyday project work. Such mechanisms let you cover an astonishingly large number of cases. Of course this approach increases the complexity of the generator configuration. You should also make sure that outdated features are removed over time. A controlled use of *deprecated* lets you sort out old features as the project progresses. The generator can easily create a file that logs which features are still in use. This will help you to remove features from the generator configuration that are no longer used – a kind of garbage collection.

15.5.4 Partitioning and Integration

Assume that different teams need the same interfaces, perhaps because one team implements a component that uses code from another team. Figure 15.5 illustrates this:

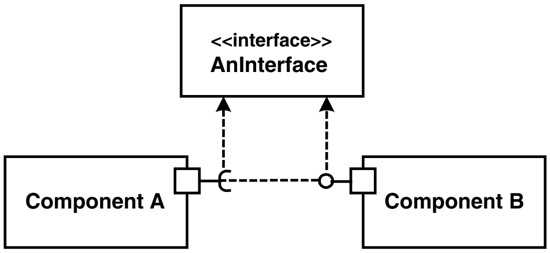


Figure 15.5 Access to shared model elements

When your work is model-driven, it is mandatory that at least the model of the interface is available in both models, as shown in Figure 15.6. However, this approach is not ideal, because information is duplicated in the two models, leading to consistency concerns. Depending on the tools, other options exist.

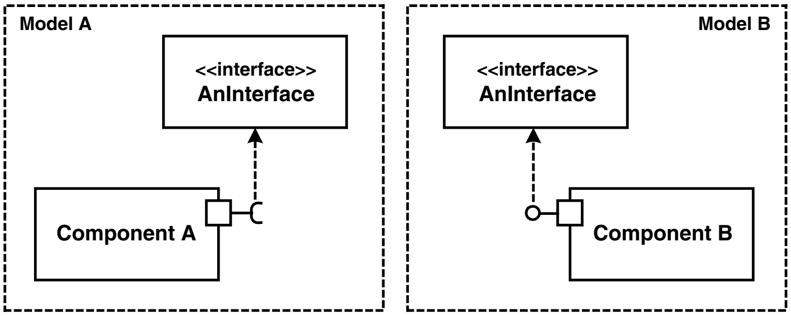


Figure 15.6 Duplication of shared model elements

Integration in the Model

If the modeling tools support it, you should make sure that the interface only exists in one place and is referenced from both models. From the generator's view, this results in one consistent model – see Figure 15.7:

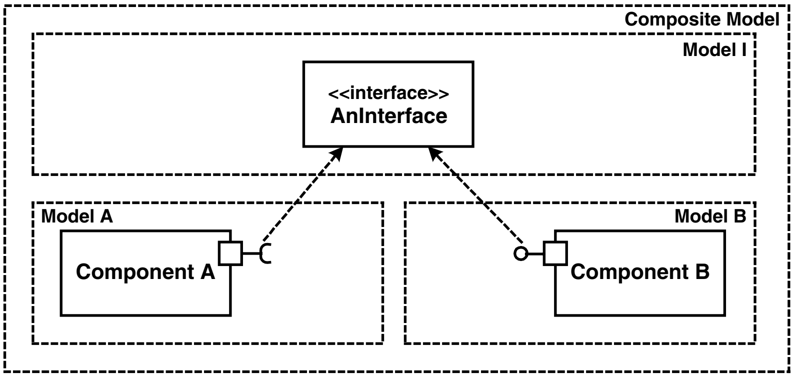


Figure 15.7 Sharing of the interface

Whether this approach can be realized or not depends on the modeling tools. Among UML tools, repository-based tools that support distributed modeling are ideal.

Integration in the Generator via Model Synchronization

If the modeling tool does not offer adequate integration options, integration can also take place at the generator level. The generator reads several input models that each contain specific model elements, as can be seen in Figure 15.8.

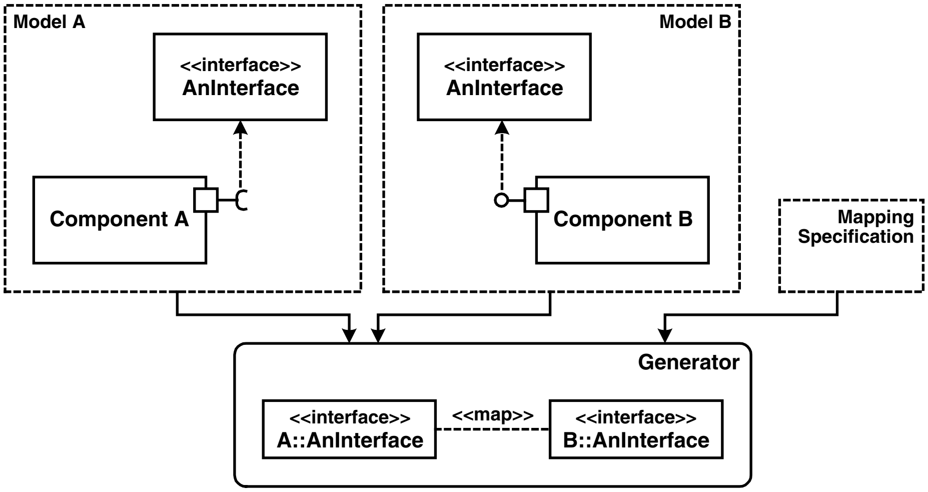


Figure 15.8 Duplication of shared model elements and resolution by the generator

In this case, it is the generator's task to solve the possible consistency problem:

- The mapping can either be carried out explicitly via a mapping specification, or simply based on identical names.
- Should both model elements (defined by the mapping as *identical*) differ content-wise, either an error can be reported or an adaption take place. Again, this can be automated to a certain extent.

Integration in the Generator via References (Proxy Elements)

A further integration alternative is the use of references, as explained in Section 8.3.4. Figure 15.9 once more illustrates the principle: the interface *AnInterface* is only present in one model. The other models merely contain interface *references* to the interface. Dereferencing can take place via names and is done automatically by the generator.

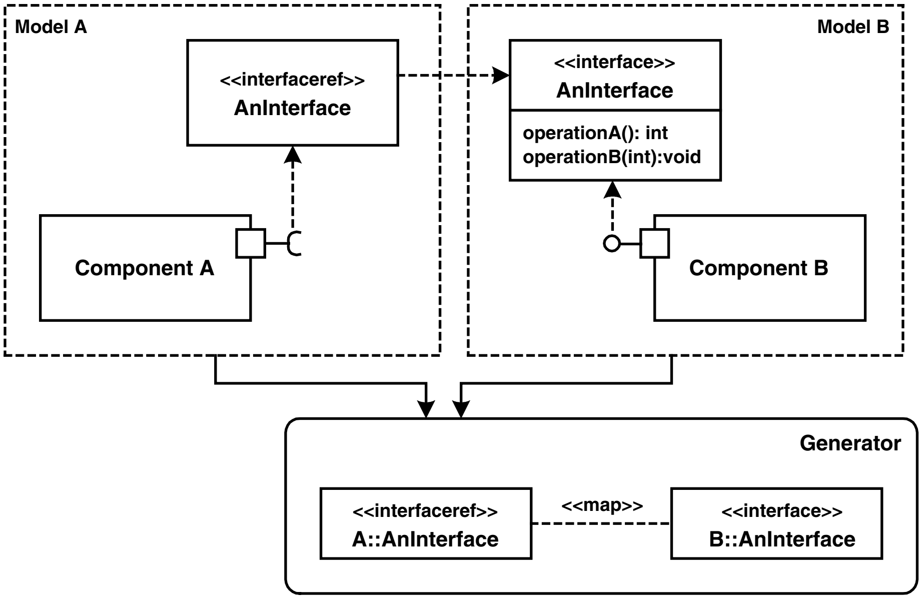


Figure 15.9 Application of reference model elements to realize commonly used model elements

16 Case Study: Embedded Component Infrastructures

16.1 Overview

This chapter contains a second, comprehensive case study, an example from the world of embedded systems. The chapter is also about the model-driven development of component infrastructures. It illustrates the following topics:

- Modeling via textual DSLs
- Use of different models for different subdomains
- Generation of infrastructure, not applications
- Product lines and feature models
- Use and adaptation of the openArchitectureWare generator
- 'Pseudo-declarative' programming of metamodel constraints
- Interceptors in the generator to separate various aspects in the metamodel
- Generation of build files
- Cascading domain architectures

The case study is based on our experience from various real-life projects.

Note that the explanations in this chapter presuppose that in principle you understand how the openArchitectureWare generator [OAW] works[1]. The case study in the first part of this book conveys important basics. Therefore, you should read it first. Also, since openArchitectureWare changes faster than this book, the examples shown here might not use the most up-to-date APIs.

[1] In the interim development of the concrete API of the generator has progressed. Details, as well as a tutorial, can be found at *http://www.openarchitectureware.org*

16.1.1 Introduction and Motivation

Embedded systems have rapidly increased in complexity in recent years:

- You can take photographs with cell phones, surf the Internet with them, as well as display video streams: it sometimes comes as a surprise to find that you can still use them to make telephone calls.
- Cars contain up to seventy small computers (ECUs – electronic control units) that control a large variety of the vehicle's technical aspects. These computers are connected to different bus systems and contain a lot of software.
- More and more appliances contain an increasing amount of software.

For a long time it was common practice in the world of embedded systems to adapt the software to the hardware platform manually – with obvious consequences for time-to-market and effort in general. This is becoming more and more difficult – a month scarcely passes without new cell phones entering the market. Similarly, the lifecycles of modern car models are growing shorter and shorter.

Because electronics and software pervade our everyday life, the quality and reliability of software must be maintained, if not improved. In the recent past, massive product quality problems for automobile and cell phone manufacturers have been reported in the press due to software and electronics.

The manual production of discrete embedded systems must therefore be replaced by the development of product lines and software systems families. The meaning of software architecture as a central building block in software development is assuming increasing significance in this context. Model-Driven Software Development as an implementation paradigm plays a pivotal role here.

Code generation per se is already widespread in the world of embedded system development. However, it is primarily used to generate application logic from models that are usually state charts (for discrete systems) or signal flow diagrams (for continuous systems). Yet the development of the technical infrastructure in which these functionalities are executed is increasingly often developed using MDSD. Attempts have been made to establish standards for this kind of middleware in various industries, for example [ASAR] in the field of automotive manufacture. A particularly promising approach is the combination of component infrastructures [Voe02] and MDSD.

16.1.2 Component Infrastructures

Component infrastructures as described in [VSW02] build on two central aspects:

- Components encapsulate a self-contained piece of application functionality. They provide this functionality via well-defined interfaces. In addition, the component describes which resources are needed for the execution of its functionality, such as other component interfaces.
- A container provides basic technical services to the component instances running in it. What these services are exactly depends on the domain. It is described further below for embedded systems.

An application thus consists of a number of component instances that are run in containers. The advantages are clear: foremost is the fact that the developer can concentrate on the implementation of the application logic while the container delivers the basic technical functionality. Depending on its degree of standardization, the container can be reused or purchased from third parties[2].

As we will see, the concrete requirements for containers depend strongly on the domain and environment. In the world of embedded systems, typical examples for container services are remote communication using various bus systems, dependency management, start-up coordination and lifecycle management, tasking and scheduling. These are realized together with the (real-time) operating system.

16.1.3 Component Infrastructure Requirements for Embedded Systems

A special challenge exists in the world of embedded systems: the scarceness of resources such as memory or electricity, processing power, and – of extreme importance – the unit price of the complete system. It is absolutely mandatory that such component infrastructures do not induce additional requirements relating to the scarce resources or reduced runtime performance. We are faced with interesting challenges here that can be tackled well with the help of MDSD.

Note that there are also embedded systems that are much bigger in nature, highly distributed, and much more complex, such as the DRE systems found in avionics. In this kind of system the generation of tailored infrastructures is not that critical. However, other aspects of system design – namely, distribution, configuration and distributed QoS – can be solved using model-driven techniques. However, this class of systems is not part of the discussion in this chapter

16.1.4 The Basic Approach

The basic approach is that the component container is generated. This allows containers to be specifically adapted and optimized for the target system. Via several models that describe different technical subdomains, the generator creates a container that precisely matches the platform and meets the requirements of both, components and hardware.

16.2 Product-Line Engineering

Before we actually show how the various artifacts are built, we want to use this case study to illustrate the basics of product-line engineering as described in Chapter 13.

The containers represent a software system family. The containers that are used with various systems have several commonalities, but also differ in specific respects. For example, they will only contain code for communication via a particular bus if the respective ECU is actually attached to this bus. The same is true for the communication paradigms. If communication takes place only via asynchronous events, the container doesn't need to contain code that brokers synchronous operation calls between different component instances via the bus.

[2] J2EE, COM+, or CCM are examples.

It makes sense to view this from the product-line engineering perspective. The argumentation is simple. If we only had a single container that could be used in all scenarios and systems, it would constitute a framework. As explained in Section 16.1.4, this approach is not practicable for performance and code size reasons.

If each container was a complete, unique entity, the concept of the container would be rendered absurd. For economic reasons, certain aspects must be similar between containers for different systems.

16.2.1 Domain Scoping

First, we must define for which scenarios the containers should be suitable. To approach this question systematically, we could use feature modeling for the entire domain of the embedded systems to determine which part of the whole domain we wish to address.

In this phase, this procedure often involves too much effort, which is why we use a more pragmatic approach here. The following aspects can be considered to help in defining the scope of the domain:

- *Operating system*. In the realm of embedded systems, it is not unusual to program directly to the hardware and not use an operating system. The features that an operating system offers for embedded systems (as well as their performance) vary considerably for each operating system. For our container, we assume that an operating system is available that at least handles scheduling. We consider memory protection, as well as other more complex features, to be optional.
- *Real-time capability*. The question of whether a system is able to perform in real-time[3] depends on various factors: in the case of distributed systems, it depends mostly on the underlying bus (FlexRay, for example, has real-time capability), on the operating system's scheduler, and the on application's functionality. We limit our examples to systems that must not meet any hard real-time requirements.
- *Safety concerns*. For many fields of application in a safety-critical environment, a large variety of certifications is required. In addition to certifying the software itself, the tools for building the software and the development process itself are often subject to certification. We also exclude such systems.
- *Dynamic vs. static configuration*. Component instances, their communication relationships, and various other aspects of the whole system can either be defined dynamically at runtime or statically at configuration time. For performance reasons, we only address the static case here, whereas only the lifecycle states are managed dynamically.

We therefore deliberately exclude certain scenarios from our domain. This helps us to keep focus during development of the system family. We focus the software system family on component containers with the following characteristics:

- No (hard) real-time capability
- Not safety-critical

[3] In this context, we understand the term *real-time capability* to imply that a deadline that is not met constitutes a system error (this is also called *hard real-time*).

- Statically configured
- Working on an operating system that at least handles scheduling

16.2.2 Variability Analysis and Domain Structuring

Since we are now familiar with the domain's boundaries, we must analyze the features of these systems to determine which of them are shared by all members of the system family, and which are variable. As a first step it is useful to divide the domain into subdomains, as described in Section 13.5:

- Communication via different bus systems and their high-performance implementation, and how to map those to the abstractions and mechanisms defined by the component model.
- Container services – additional services that the container can provide for the components at runtime.
- Deployment and consistency checks – to achieve optimal and consistent distribution of the components over the system nodes.
- Integration of scheduling with the operating system.

For each of these subdomains, one should perform a variability analysis. As examples, we will focus on two subdomains here: communication mechanisms and cross-cutting services of the container.

Communication Paradigms

Connectors serve to describe the communication relationship between two ports of components, as shown in Figure 16.1.

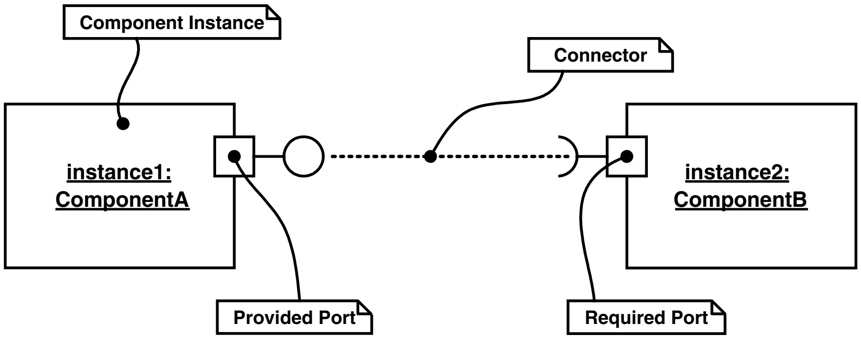


Figure 16.1 Component instances, ports and connectors

The paradigm that should be used for communication (operation call, message-oriented) is defined by the interface of both ports, and particularly by the connector. The feature model (see Section 13.5.3) in Figure 16.2 shows which communication options exist. You can see from this that two features must be determined first of all: technology and the paradigm.

Of course the technology depends on the constraints: one can only communicate via those bus systems that are connected to the respective hardware node. Moreover, one can only communicate locally if both instances are located in the same container (a container defines an address space).

The paradigm is somewhat more difficult. In principle, one can either communicate using a client/server or message-based method. In the case of client/server communication, one must decide whether one wishes to communicate synchronously or asynchronously. In the case of asynchronous communication, one must decide how one wishes to be informed about the call's result.

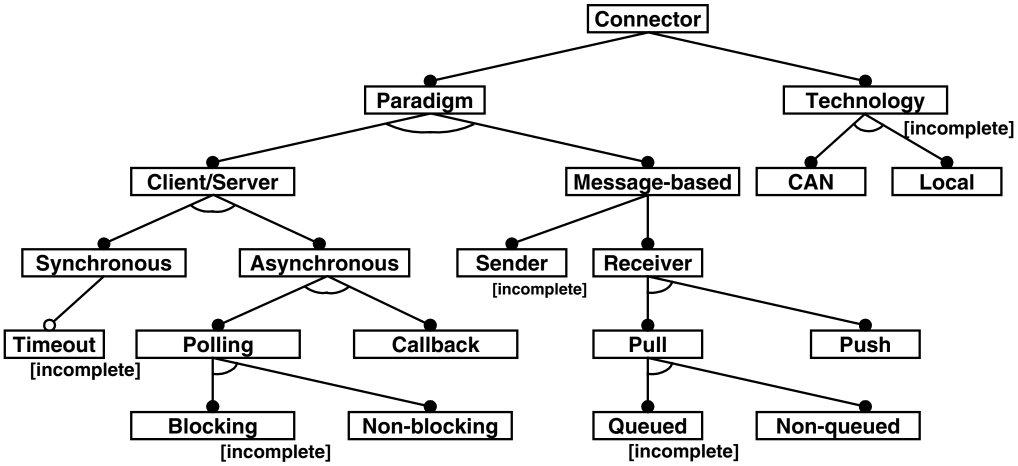


Figure 16.2 Feature model of the communication options

These and other decisions must also be made in the case of message-based communication.

It is important to determine when the developer defines each feature. This can, for example, be done using an annotation of the feature diagram, as shown in Figure 16.3.

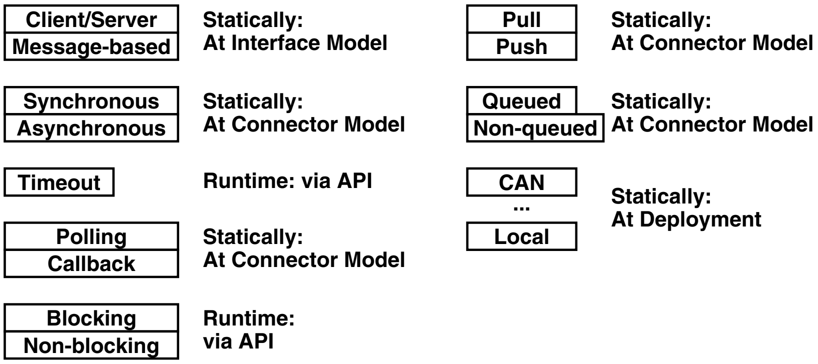

Client/Server
Message-based
Statically: At Interface Model
Synchronous
Asynchronous
Statically: At Connector Model
Timeout
Runtime: via API
Polling
Callback
Statically: At Connector Model
Blocking
Non-blocking
Runtime: via API
Pull
Push
Statically: At Connector Model
Queued
Non-queued
Statically: At Connector Model
CAN
...
Local
Statically: At Deployment

Figure 16.3 Binding times for different features

The reasons why a number of features must be defined statically are code size, memory consumption, deterministic timing characteristics, and performance.

The question of whether polling should be blocking or non-blocking must be decided statically on some operating systems, such as Osek, to configure the operating system in such a way that blocking is actually possible[4].

Container Services

The services offered by the container constitute a further important aspect of the software system family component containers – see Figure 16.4.

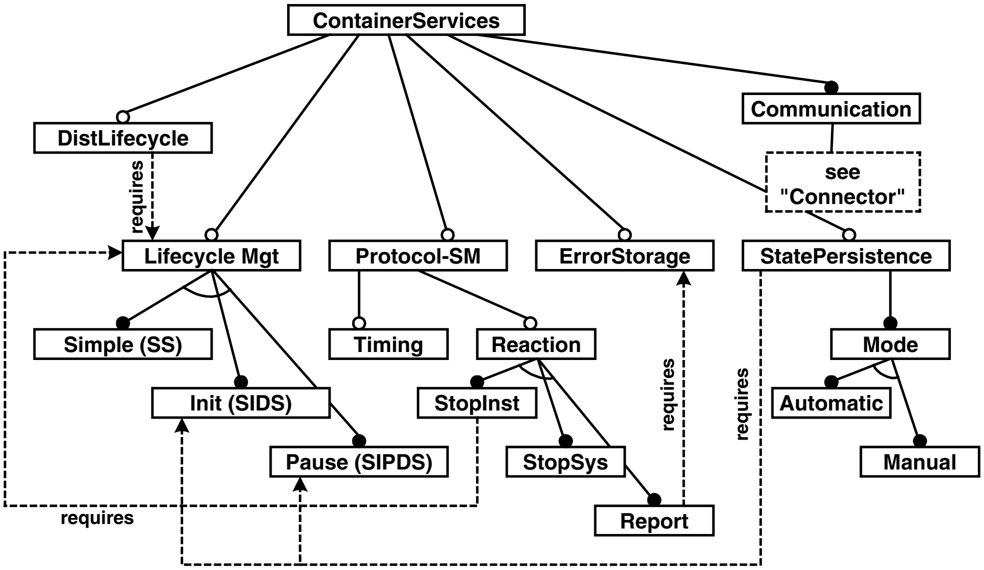


Figure 16.4 Feature model for the container's services

First, containers always offer communication services. The container can also take care of the component instances' lifecycle by controlling their activation and deactivation. We distinguish three modes:

- *Simple* only knows the events *Start* and *Stop*.
- *Init* also knows an initialization phase, that is *Start*, *Init*, *Deinit*, *Stop*.
- *Pause* knows a *pause* mode, for example for energy-saving.

[4] BCC versus ECC, for the Osek insiders among our readers.

The container can monitor protocol state machines. In this context, two aspects must be defined: first, should timing constraints also be monitored, and second, how should one respond to errors?

- *StopInst* stops the respective instance (for this purpose, lifecycle management is needed).
- *StopSys* stops the entire container.
- *Report* merely reports the error to a central error storage.

The container must have the *error storage* feature for the latter.

The container can also manage the state of the component instances. If this feature is activated, the container can store the component's state persistently either automatically or after the developer invokes a specific API operation. Lifecycle management must be activated for this.

Finally, the container can be involved in distributed – system-wide – lifecycle management. This requires lifecycle management in the container.

As a rule the respective features are bound statically, but there are differences in the details:

- The lifecycle mode must be known when the component implementation is developed, because one might want to react to lifecycle events in the implementation code.
- The question of whether protocol state machines should be monitored at runtime can be decided at the generation time of the container.

We address this issue again when we discuss the definition of the production plan.

16.2.3 Domain Design

During the domain design process, the target architecture is defined – that is, the common basic architecture of the different software system family's members is determined, as well as how the optional features integrate. The production process is also defined: this prescribes the route from the model to the final product.

Target Architecture

We cannot dwell on all the details of the target architecture here, as this would exceed the scope of our case study. Figure 16.5, however, provides an overview. The shaded items constitute the MDSD platform. The items in gray are generated, and the white parts represent the components' manual implementations.

We want to reinvent as little as possible, which is why we make extensive use of the operating system facilities. The lower dashed box comprises the operating system and some basic libraries and drivers. The upper dashed box makes up the container.

Note that the items displayed here do not represent all artifacts that are required or generated. For example, we need generated makefiles and generated configuration files, but these are not relevant for the target architecture of the completed system.

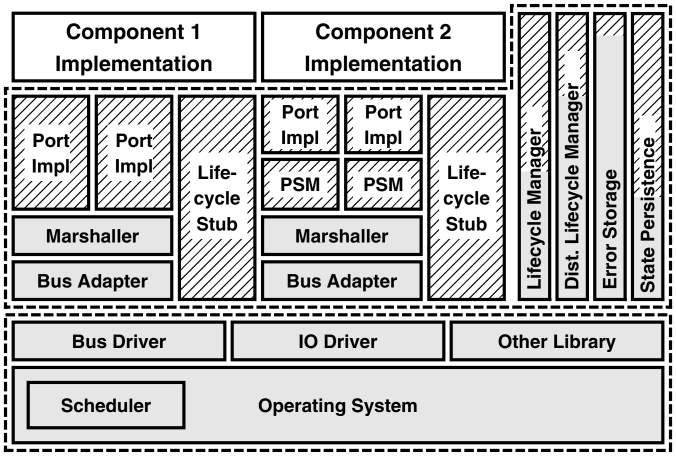


Figure 16.5 Family architecture of the example component infrastructure

The Production Process

The production process describes how one builds complete, executable systems, starting from the models. Figure 16.6 shows a simplified version of this process for the case study example. The steps are explained below, but we do not address the roles – the vertical divisions in the diagram – in this process in detail. In the illustration, the solid arrows denote a chronological order and the dashed arrows mark a dependency or a *uses* relationship.

Let's look at some specifics regarding this process: we'll clarify the details in the course of the case study:

1. In the first step the interfaces are modeled. Interfaces provide the basis for communication between the component instances.
2. In the second step the components and their ports are defined with the help of the interfaces.
3. From these two models, the generator can create the skeleton component code, such as for example C header files.
4. Based on the generated skeleton code, the developer can now develop the actual component implementation.
5. The developer is now able to define the complete system, that is, component instances, the connectors between them, as well as the distribution of instances to the hardware nodes (which must also be described here). The container service configuration model can now be established. This defines which of the additional services in Figure 16.4 the container must actually provide for the respective system.
6. This is the actual generation step. The generator creates the code that is needed to implement the required container functionality. To do this it needs access to all models that have been

defined up to this point. The generator also creates a configuration file for the respective operating system, as well as a makefile for building the image.

7. In this step, compiler, linker and make can create the program's image. This image consists of the container, the (generated) component skeleton code, the manually-created component implementation, and the libraries required from the platform.
8. In the last step, the operating system instance is created from the respective configuration file. This is done by a tool that is part of the operating system.

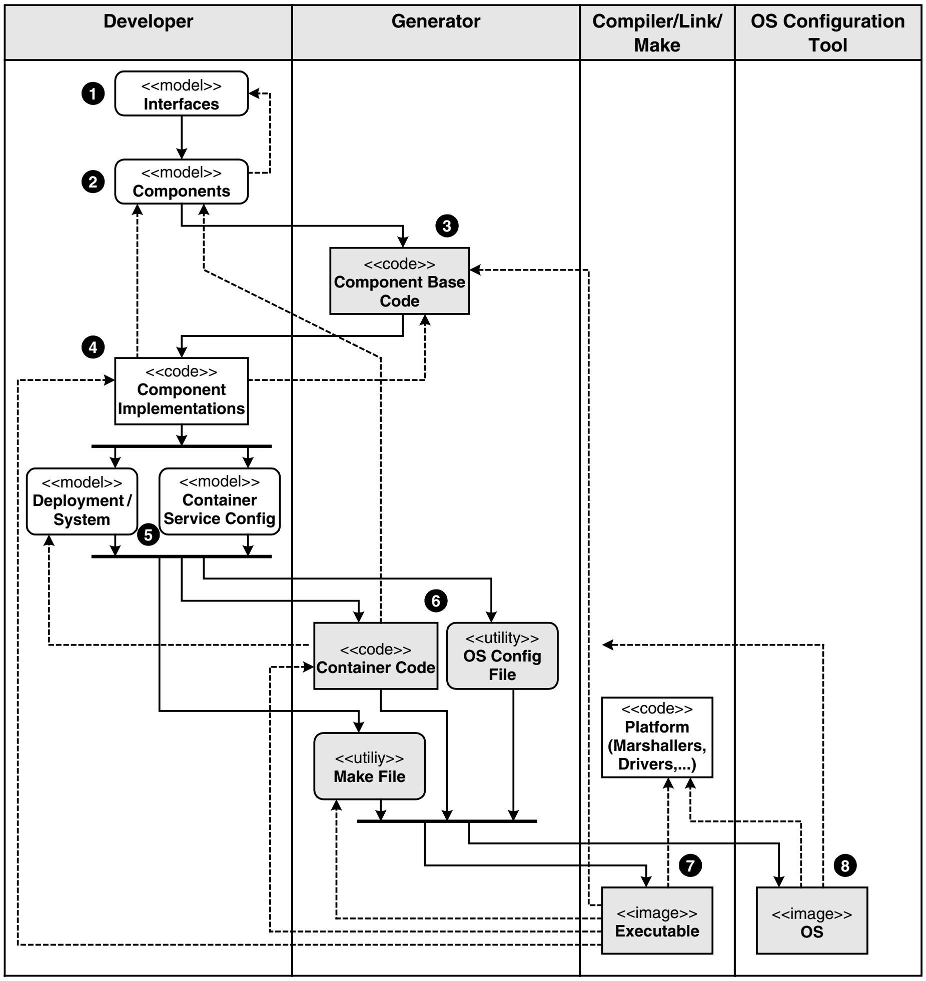

Developer
Generator
Compiler/Link/ Make
OS Configuration Tool
1
<<model>> Interfaces
2
<<model>> Components
3
<<code>> Component Base Code
4
<<code>> Component Implementations
<<model>> Deployment / System
<<model>> Container Service Config
5
6
<<code>> Container Code
<<utility>> OS Config File
<<utiliy>> Make File
<<code>> Platform (Marshallers, Drivers,...)
7
8
<<image>> Executable
<<image>> OS

Figure 16.6 Production process of the middleware system family

We can now transfer both operating system and application – that is, the container and components – to the target hardware and run them.

16.2.4 Domain Implementation

The last phase of product-line engineering is the implementation of the concepts we have already introduced. This phase constitutes the major part of our case study and is explained in the remainder of this chapter.

16.3 Modeling

This section describes what the models look like in the system's various subdomains. For this purpose, a custom DSL is used for each of the three subdomains. Like all DSLs, these too consist of:

- A metamodel that defines the abstract syntax and the static semantics.
- The concrete syntax, which defines the notation to render the models.
- The (dynamic) semantics, which define the meaning of the metamodel elements.

The metamodels are implemented in Java, as prescribed by the openArchitectureWare generator. They are coupled via gateway and proxy metaclasses. The concrete syntax of the DSL varies in all three cases. A UML profile is used here as well as a textual notation (inspired by CORBA IDL) and a special XML-DTD. In all three cases, the DSL's semantics are enforced via the transformations used by the generator to generate the implementation code.

16.3.1 Definition of Interfaces

How do we define a component? A component consists of a series of ports that either provide or use the services of an interface. Figure 16.7 shows the metamodel for interfaces.

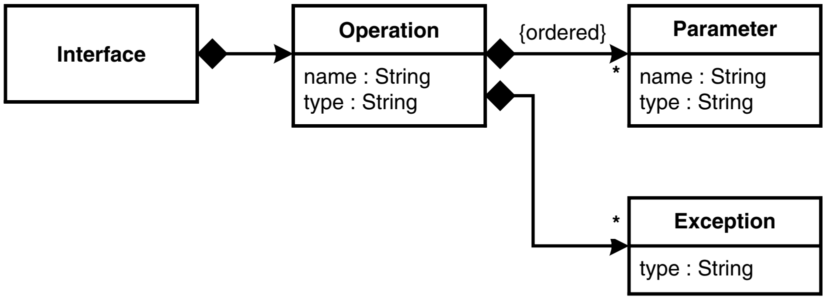


Figure 16.7 The metamodel for interfaces

The concrete syntax will be textual, as graphical notations are no more productive for this purpose. Let's begin with the definition of an interface:

```
interface Sensor {
  operation start():void;
  operation stop():void;
  operation measure():float;
}
interface Controller {
  operation reportProblem(Sensor s,
            String errorDesc ):void;
}
```

In addition to this syntactic definition of interfaces, protocol state machines can also be modeled, for example by using UML. These define in which order the operations of an interface may be invoked. From this state machine, code can be generated that monitors at runtime whether the constraints defined by the state machine are observed by clients. If this is not the case, an error can be stored in an error store that can be used subsequently for diagnostics. Figure 16.8 shows an example of a protocol state machine.

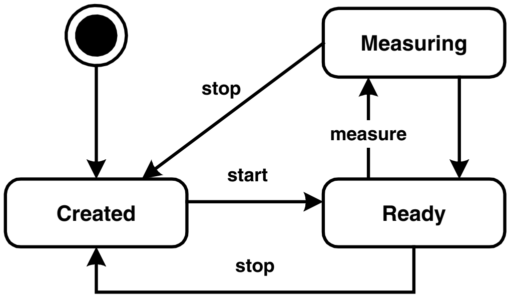


Figure 16.8 An example of a protocol state machine

The metamodel for this purpose corresponds with the well-known metamodel for state machines, consisting of states, transitions, guards, and events. The events in this case are operations that can be invoked on the interface. We address the question of how the interface finds its protocol state machine and how runtime monitoring code is generated later.

16.3.2 Definition of Components and Ports

Next we model components and their ports. The metamodel used here is shown in Figure 16.9.

Since the concrete syntax is going to be a UML profile, we use the UML metamodel as a basis. The metamodel element *InterfaceRef* is particularly interesting – it serves as a reference to interfaces defined in other models. Assignment of the reference to the model that defines it is based on name equivalence. Figure 16.10 represents the concrete syntax of an example component model based on UML.

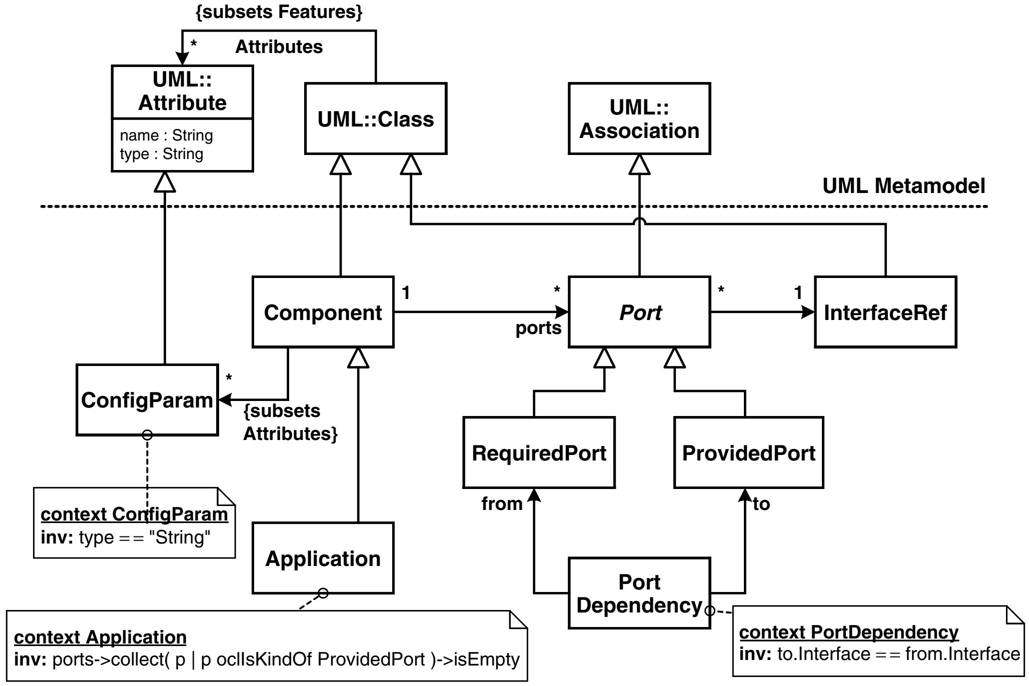


Figure 16.9 Metamodel for components, ports and interfaces as an extension of the UML metamodel

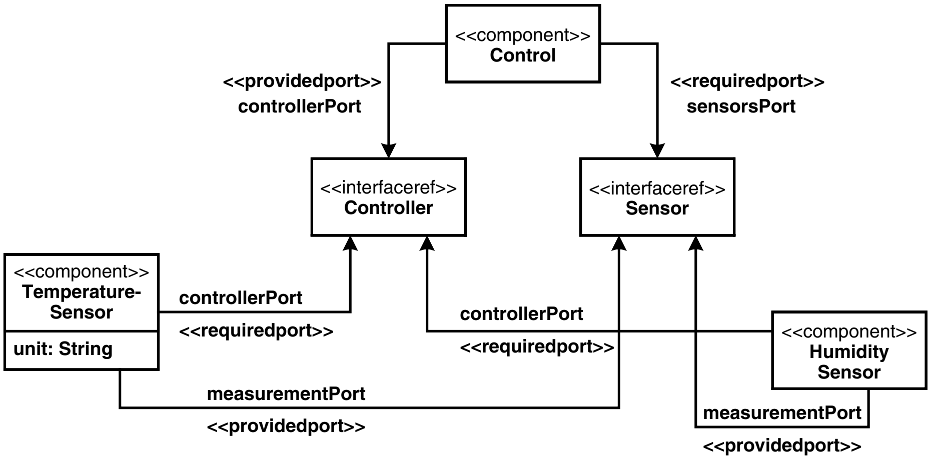


Figure 16.10 An example component model of a weather station

16.3.3 Definition of a System

A system consists of a number of nodes. One or more containers are deployed on each node. A container hosts one or more component instances. The ports of component instances are linked using *connectors*. As can be seen from the metamodel, there are different kinds of connectors. Figure 16.11 shows this metamodel.

Here, too, we work with metamodel references to reference elements that are defined in other models.

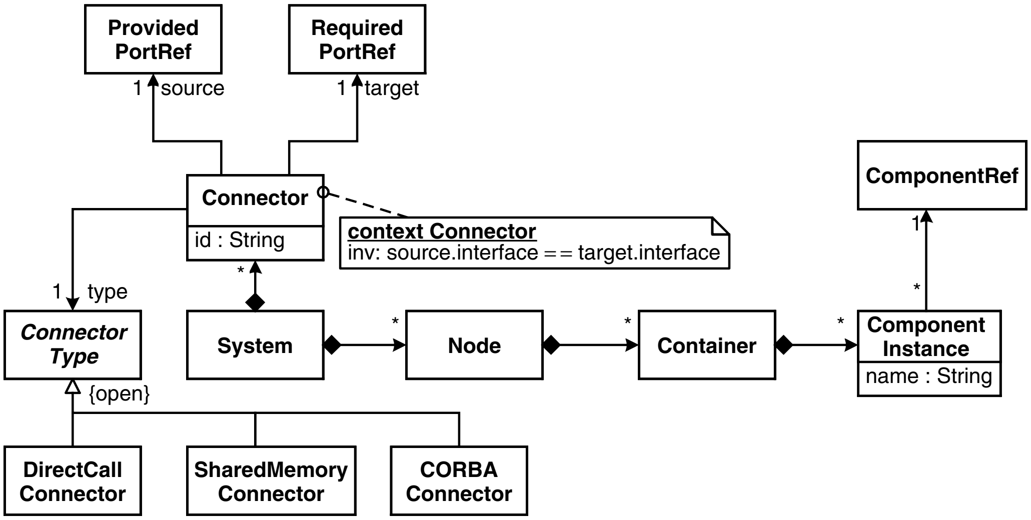


Figure 16.11 Metamodels for systems, containers and connectors.

For the system model, the concrete syntax is an XML schema that has been designed explicitly for this purpose. First we define the component instances, system nodes and containers.

```
<system name="weatherStation">
  <node name="main">
    <container name="main">
      <instance name="controller"
                type="Control"/>
    </container>
  </node>
  <node name="inside">
    <container name="sensorInside">
      <instance name="tempInside"
                type="TemperatureSensor">´
        <param name="unit" value="centigrade"/>
      </instance>
    </container>
  </node>
```

```
    <node name="outside">
      <container name="sensorsOutside">
        <instance name="tempOutside"
                  type="TemperatureSensor">´
          <param name="unit" value="centigrade"/>
        </instance>
        <instance name="humOutside"
                  type="HumiditySensor"/>
      </container>
    </node>
```

Next, we describe the connections between the component instances in the form of connectors.

```
    <!-- temperature sensor outside -->
    <connector name="toSensorTempOutside">
      <providedPort instance="tempOutside"
                    port="measurementPort">
      <requiredPort instance="controller"
                    port="sensorsPort">
    </connector>
    <connector name="fromSensorTempOutside">
      <providedPort instance="controller"
                    port="controllerPort">
      <requiredPort instance="tempOutside"
                    port="controllerPort>
    </connector>

    <!-- humidity sensor outside -->
    <connector name="toSensorHumOutside">
      <providedPort instance="humOutside"
                    port="measurementPort">
      <requiredPort instance="controller"
                    port="sensorsPort">
    </connector>
    <connector name="fromSensorHumOutside">
      <providedPort instance="controller"
                    port="controllerPort">
      <requiredPort instance="humOutside"
                    port="controllerPort>
    </connector>

    <!-- temperature sensor inside -->
    <connector name="toSensorTempInside">
      <providedPort instance="tempInside"
                    port="measurementPort">
      <requiredPort instance="controller"
                    port="sensorsPort">
    </connector>
    <connector name="fromSensorTempInside">
      <providedPort instance="controller"
                    port="controllerPort">
      <requiredPort instance="tempInside"
                    port="controllerPort>
    </connector>
  </system>
```

This defines the logical structure of the system. What is missing so far is the precise characterization of the connectors' features. Now the feature model defined in Section 16.2.2 comes into play. The following two configurations would be valid with regard to the feature model:

```
connector.technology=CAN
connector.paradigm=async-cs,polling,blocking

connector.technology=local
connector.paradigm=sync
```

Such configuration settings must be stated additionally for the connector definition mentioned above. In this context, a constraint emerges: a local connector can only be defined between two instances if these instances are active in the same container. The generator must check this and, if necessary, issue the respective error messages.

16.3.4 The Complete Model

The complete metamodel consists of a superset of metamodels from the various subdomains. Figure 16.12 shows the reference classes (proxy elements) that establish relationships between the different subdomains.

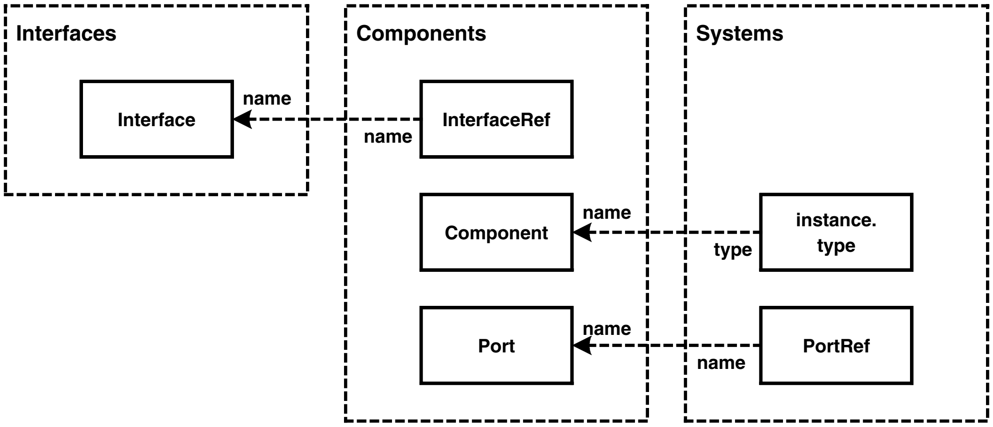


Figure 16.12 Relationships among the metamodels

16.3.5 Processing

Figure 16.13 shows how the different artifacts are processed by the generator.

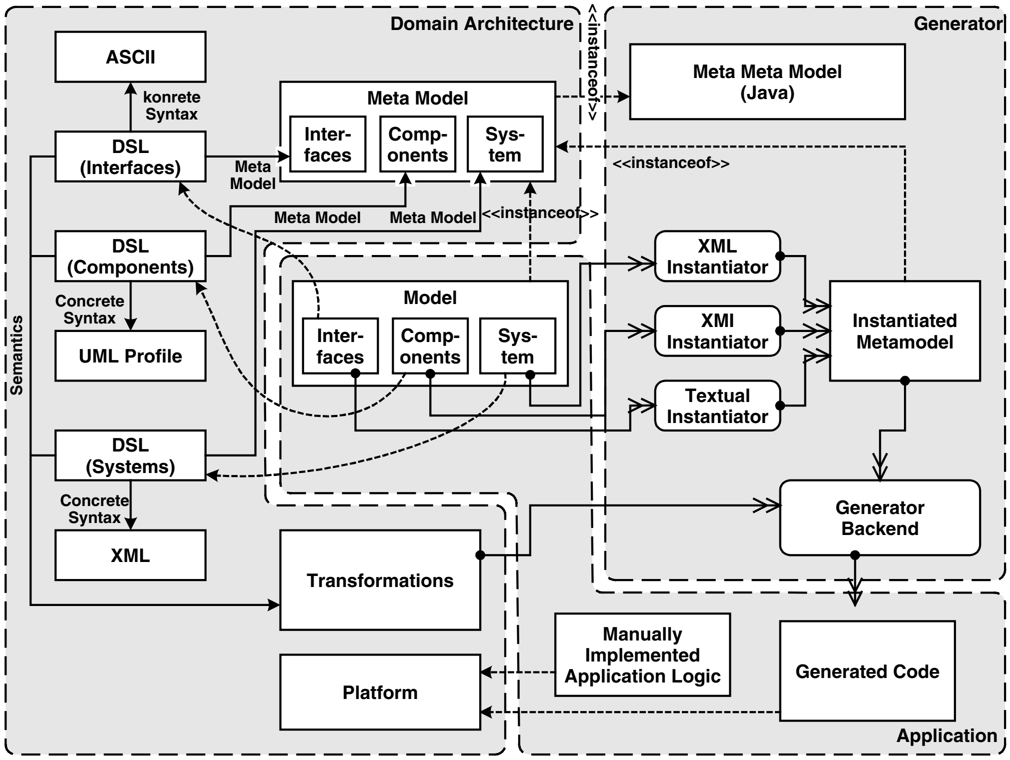


Figure 16.13 Overview of the generation process

16.4 Implementation of Components

This section shows how the concept of the component is mapped to programming languages and how the container interacts with the components. To simplify matters, we exclusively use Java. as an example implementation language. We are aware of course that for 'real' embedded systems a mapping to C would be more relevant. Mapping to Java is nevertheless easier to understand and is applied in practice, for example for cell phones.

Mapping to Java

A fairly direct mapping of the component is possible due to the fact that Java is an object-oriented language. Figure 16.4 gives an example:

- A (client/server) component interface is mapped to a Java interface.
- For each component, an abstract base class is generated from which the developer derives custom classes that implement operations offered by the components.

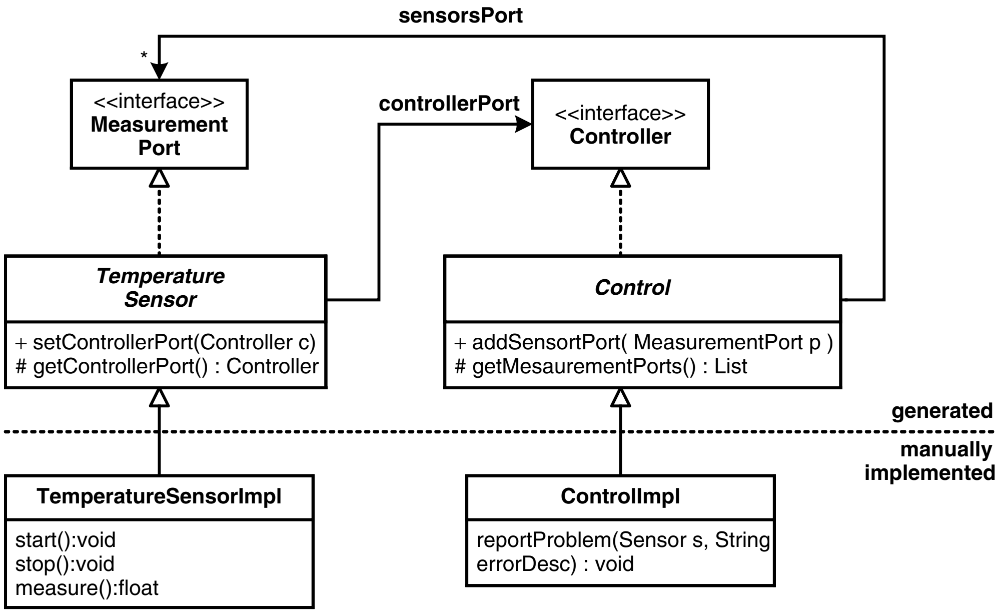


Figure 16.14 The class model of an implementation in Java (local)

The container is responsible for setting the ports' references correctly at system start-up. The developer can then access these ports from within the implementation code during implementation.

```
public class TemperatureSensorImpl
             extends TemperatureSensor {
  public void start() { … }
  public void stop() { … }
  public float measure() {
    float value = // use driver to measure
    if ( /* there is a problem */ ) {
      getControllerPort().reportProblem(this,
        "cannot measure…." );
    }
  }
}
```

Generated and non-generated code are completely separated. The example given above works if both components are instantiated in the same container – that is, in the same process, the same JVM. In the case of distributed components, proxies must be generated automatically, as shown in Figure 16.15.

These proxies are only generated if distributed communication actually takes place. The proxies then contain the code necessary to realize the selected communication paradigm: marshaling,

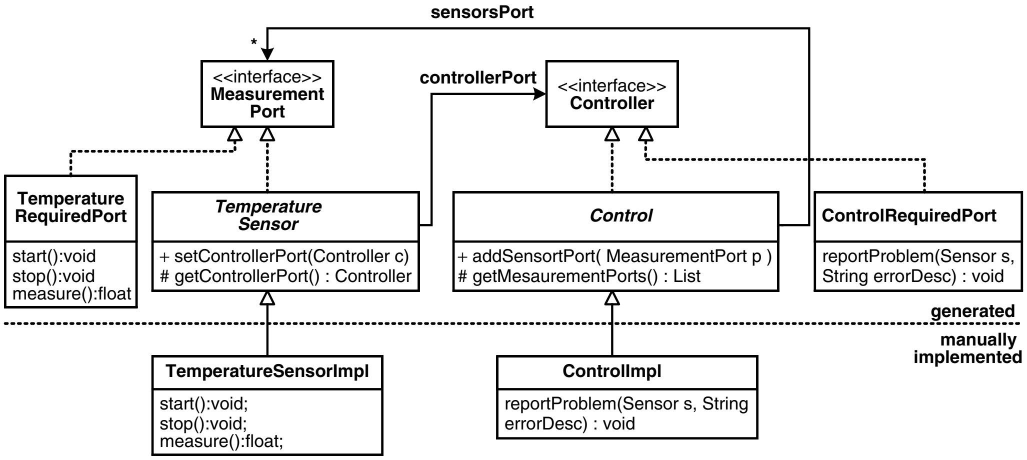


Figure 16.15 The class model of an implementation in Java (remote)

connection management, and so on. As much of this functionality as possible is implemented as part of the platform in the form of libraries.

Since the proxies implement the same interface as the component that provides the respective service, the component implementation does not change in the case of distributed communication – the container simply wires the proxy instead of the component.

16.5 Generator Adaptation

The generator must be adapted to be able to fulfill the following tasks:

- Parsing of the textual syntax
- Parsing of the XML configuration
- Merging of the different partial models and resolution of the references
- Validation of the complete model
- Generation of the different implementations

We discuss each of these requirements briefly in the following sections.

16.5.1 Parsing of the Textual Syntax

We use the parser generator *JavaCC* [JCC] to generate the parser. We don't go into details of this step here, just deal with integration into the generator. Another parser generator could also be used instead of *JavaCC*.

We begin with the parsing of interface definitions of the following type:

```
  interface Sensor {
   operation start():void;
   operation stop():void;
   operation measure(): float;
 }
```

The task of *JavaCC* is to parse the text and create the abstract syntax tree (AST). Ultimately this is simply an object tree, whereas for each syntax node a corresponding object is used. Different AST classes are used to represent the different syntax elements. These correspond with metaclasses in the context of the openArchitectureWare generator. Since the latter expects all metaclasses to inherit from the class *ModelElement*, we must make sure that the base class of the AST nodes used by *JavaCC* also inherits from *ModelElement*. All AST classes are therefore model elements.

We modify the class *SimpleNode*, which is provided with *JavaCC* and constitutes the base class for all AST nodes, as follows:

```
public class SimpleNode
   extends ModelElement implements Node {

  public SimpleNode(int i) {
    id = i;
    JCCHelper.getMetaEnvironment().
              addElement( this );
    setName("");
  }

  // rest as before…

}
```

SimpleNode extends the class *ModelElement* that is provided by the openArchitectureWare generator. The constructor adds the current element to the generator's *MetaEnvironment*. Without this line, the whole system couldn't work, because the generator wouldn't 'know' the object.

Let's assume that we have generated the parser (*InterfaceParser.java*) from a suitable JavaCC syntax definition file. We then have to implement a corresponding frontend for the generator. Generator frontends are used to read the concrete syntax of models and provide a set of *Model Elements* that are available in the generator at runtime and serve as the basis for generation. Frontends must implement the *InstantiatorInterface*. The frontend's implementation delegates the parsing to the parser generated by JavaCC. As a result, our frontend looks like this:

```
public class JCCInstantiator
   implements InstantiatorInterface {

  public ElementSet loadDesign(
            InstantiatorEnvironment env)
            throws ConfigurationException,
              InstantiatorException {
     JCCHelper.setMetaEnvironment(
               env.getMetaEnvironment() );
     ElementSet result = new ElementSet();
     String spec = // read specification from somewhere…
     StringBufferInputStream s =
               new StringBufferInputStream( spec );
     InterfaceParser p = new InterfaceParser(s);
     try {
        ASTStart start = p.Start();
        ASTClass cls = (ASTClass)start.Child().get(0);
        result.add( cls );
     } catch ( Exception ex ) {
        ex.printStackTrace();
     }
     return result;
  }
}
```

Two points should be observed from this:

- The current *MetaEnvironment* is saved to the *JCCHelper* in the first line of the operation – as shown above, the AST classes retrieve it from there in their constructor.
- Inside the *try* block, the *ASTClass* instance is retrieved. This is the only element that is written into the operation's *result*. In effect, the instantiator returns the interface declaration as the top-level element of the model.

The integration into the generator is now almost complete. We must merely tell the generator to use the instantiator instead of the standard XMI frontend. To do that, we write a generator plug-in that contributes the instantiator shown above. You can find full details of how to do that in the openArchitectureWare documentation.

16.5.2 Parsing the System Definition XML

To parse the XML configurations, we use a generic XML parser that creates openArchitectureWare models from any XML document. The idea behind this is to render the AST – the model structure – directly using the XML document. The parser then only has to create a Java object graph representing the same structure.

The parser expects metaclasses with same name as the respective XML elements to exist in a specific package (which can be freely chosen by the developer). Let's look at the following example:

```
<system name="weatherStation">
  <node name="main">
    <container name="main">
      <instance name="controller"
                type="Control"/>
    </container>
  </node>
</system>
```

On reading this XML file, the generator first tries to instantiate a class *System*. It then calls a method *setName()* on the respective object and supplies it with the argument *«weatherStation»*. The parser then instantiates a class *Node*, sets its name and tries to call the method *addNode()* on the previously generated system object. The newly-created *Node* object serves as an argument here.

With this mechanism, the parser can instantiate any XML structure, but only if metaclasses that match the XML structure are available. The openArchitectureWare generator already offers a suitable frontend that fulfills the tasks described here.

16.5.3 Parsing and Merging the Complete Model

The complete model only makes sense when the models of all three subdomains are considered together. The generator must parse all three models and merge them. The reference objects – for example the *InterfaceRefs* – must be linked to the referenced objects, the interface of the same name. Figure 16.12 illustrates these relationships.

Parsing the Various Models

To parse the different models, we have to contribute all the instantiators to the generator. To achieve this, we create a plug-in the contributes these instantiators.

```
package util;

public class ECModelReaderPlugin extends GeneratorPlugin {

   private String systemConfFile;
   private String interfaceFile;
   private String componentsFile;

   public void init() {
      systemConfFile = getProperty("EC.SYSTEM");
      interfaceFile = getProperty("EC.INTERFACE");
      componentsFile = getProperty("EC.COMPONENTS");
   }

   public List contributeInstantiators() {
     return makeList(
       // a frontend that reads the UML model
       new XMIInstantiator( componentsFile ),
       // a frontend that reads the XML system spec
       // use ecMetamodel as package prefix when
       // attempting to load metamodel classes
       new XMLInstantiator( systemFile, "ecMetamodel" ),
       // a frontend that reads the textual spec
       // for the interfaces
       new JCCInstantiator( interfaceFile )
     );
   }
}
```

To get the generator to use the plug-in we have to configure it in the Ant file that actually starts the generator.

Merging of the Models

openArchitectureWare automatically ensures that the model elements from the various parsed models are available in a single model as instances of the respective Java metaclasses. After the models have been parsed, we have different instances of the class *Interface* and the class *InterfaceRef*. The following example shows an excerpt of the definition of *InterfaceRef* that makes sure that *InterfaceRefs* find the corresponding *Interface*:

```
package cmMetamodel;

public class InterfaceRef extends Class {

  private Interface referencedInterface = null;

  public Interface Interface() {
    if ( referencedInterface == null ) {
      String myName = Name().toString();
      referencedInterface =
                  (Interface)MMUtil.findByName( this,
                  Interface.class, myName,
                  "Cannot find interface named "+myName );
    }
    return referencedInterface;
  }
}
```

This code needs to be explained. The operation *Interface()* returns the interface that the corresponding *InterfaceRef* references. As we have already explained, the connection is established via the name. The operation *MMUtil.findByName()* does all the work for us: it searches for all instances of the class *Interface* in the model to which the current object *(this)* also belongs. If exactly one is found, it is returned. If none is found, an error with the specified error message is generated.

Note that *InterfaceRef* is a model element that is originally defined in the UML-based component model, whereas the interface is defined in the textual interface model. In the generator's model representation, these differences are no longer relevant, because it exclusively operates on the instantiated metamodel (the abstract syntax).

With the help of this approach, all references are resolved – including the connection between an interface and its optional protocol state machine.

Partitioning of Models

We have now answered the question of how different subdomains (interfaces, components, deployment) of the system can be represented as separated models using different concrete syntax, and later be merged in the generator. The partitioning of DSLs and models is a related topic, which was addressed in Chapter 15. For this purpose, the same referencing and integration mechanisms can be used at the generator level if the modeling tool does not offer sufficient support for distributed modeling.

16.5.4 Pseudo-declarative Metamodel Implementation

Constraint Checks

Resolving references is only one of many validation steps. If the referenced model element is not available, an error message must be issued. In the context of non-trivial MDSD systems, a wide variety of such constraints exist that must be checked by the generator.

openArchitectureWare offers a separate step in the generator workflow to cope with this. After the complete model has been parsed and before code generation begins, the operation *CheckConstraint* is called on all model elements. The developer can place the metaclass-specific constraints (model invariants) inside this operation.

In our view it is essential that we are able to formulate these constraints as expressively and briefly as possible. A syntax based on declarative languages such as OCL is a great help here. openArchitectureWare includes a number of helper functions in the form of the classes *Checks*, *Filters*, and *MMUtil*.We show a few examples below.

One model constraint states that components are not allowed to define their own operations. Also, superclasses or implemented interfaces are not allowed:

```
public class Component extends Class {

  public String CheckConstraints() {
     Checks.assertEmpty( this, Operation(),
        "must not have attributes." );
    Checks.assertEmpty( this, Generalization(),
        "must not have superclasses or subclasses." );
    Checks.assertEmpty( this, Realization(),
        "must not implement any interface." );
```

A further requirement is that the ports of a component must have unique names.

```
    Checks.assertUniqueNames( this, Port(),
        "a component's ports must have unique names." );
  }
  // more …
}
```

Note that the *Port() operation* returns the set of all defined ports of a component.

Now consider the constraint validation of interfaces. Let's suppose we only support the primitive types *int* and *long*. In this case we would need to check whether the parameter's type is in fact only one of those two values:

```
public class ASTParameter
       extends SimpleNode { // SimpleNode extends
                            // ModelElement
    private String type;
    private String name;
    private static String[] types = new String[]
                                    {"int", "long"};

    public void CheckConstraints() {
      Checks.assertOneOf( this, type, types,
             "Type must be one of int, long" );
    }

    public ASTParameter(int id) {
        super(id);
    }

    public void setType(String type) {
        this.type = type;
    }

    // set name implemented in ModelElement
}
```

As this example shows, the differences in the concrete syntax (UML, textual, XML) are completely neutralized – we can check constraints in the same way for textual models.

Filtering

On the metamodel level, a port is a UML association that points to an *InterfaceRef*. The set of all ports can be created via respective filtering of all associations of the component[5]. Here is the implementation of the operation *Port()*:

```
public class Component extends Class {

  // as before

  public ElementSet Port() {
    return Filters.filter( this, AssociationEnd(),
      new AssocEndOppositeClassTypeFilter(
          InterfaceRef.class ),
      new AssocEndAssocMapper() );
  }
}
```

[5] A component can also have other kinds of associations, but we don't go into that here.

To understand what happens here, it is helpful to examine the structure of the relevant part of the metamodel. Since we realize the metamodel as an extension of the UML metamodel, we can also find constructs of the UML metamodel in it. Specifically, an association in the model is represented as an instance of an *AssociationEnd*, an *Association*, and another *AssociationEnd.* We realize the ports as subclasses of *Association*. Figure 16.16 shows the relevant part of the metamodel.

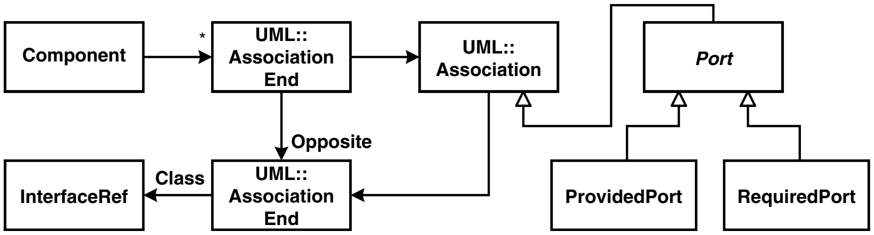


Figure 16.16 Metamodel for ports as an extension of associations

Based on this metamodel, the code should become understandable: the *filter() operation* is invoked, which expects four arguments:

- As for all helper functions, the current model element is supplied. This is important to enable the correct context to be reported in error messages.
- The set (more precisely: the *ElementSet*) of the model elements that is to be filtered is provided. These are the corresponding *AssociationEnds*, that is, the association ends that end at the current element (here the component).
- The filter object. In our example, this is an *AssocEndAssocTypeFilter*. It assumes that an *AssociationEnd* is provided as the input to its filter() operation, then it gets the association and filters it based on the specified type.
- A *mapper* is supplied that maps each association end to its corresponding association object, since we wish to get the ports (as subclass of *Association*) as the result of the operation, not the association ends.

You can implement powerful filters ‘pseudo-declaratively’ with quite little effort if you use the constructs introduced here. The alternative would be nested *for* loops.

16.6 Code Generation

Code generation uses the template language *Xpand*, which is part of the openArchitectureWare described in earlier chapters. In this section we avoid details of the template language’s syntax, but point out important characteristics and introduce some useful techniques for metamodel implementation that significantly simplify template programming.

16.6.1 References

Merging the different models is accomplished via references. After the model has been merged, a component, a port, and the corresponding interface look for example like Figure 16.17:

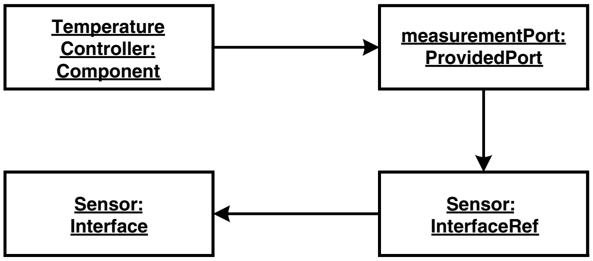

Temperature Controller: Component
measurementPort: ProvidedPort
Sensor: InterfaceRef
Sensor: Interface

Figure 16.17 A merged model (*InterfaceRef* references *Interface*)

From the templates' perspective, however, it is irrelevant that the interface is integrated via a reference – the conceptual metamodel says that a port is associated with an interface. To allow this abstraction from the template's perspective, we simply introduce an operation *Interface()* to the class *Port* that implicitly de-references the reference:

```
public abstract class Port extends Association {

  public IntefaceRef InterfaceRef() {
    // return the model element on the 'target' side
    // of this association; remember ports are modeled
    // as associations that point from component to
    // interface reference.
  }

  public Interface Interface() {
    return InterfaceRef().Interface();
  }
}
```

Thus an *Xpand* template can be written as follows:

```
«DEFINE Something FOR Component»
  public class «Name» implements
  «FOREACH ProvidedPort AS pp EXPAND»
    «Interface.Name»
  «ENDFOREACH»
  {
    // more stuff to come
  }
«ENDDEFINE»
```

In distributed application development it often happens that specific model constructs either describe an element directly, or a reference to it. In the former case, one could imagine the following: assume interfaces could also be defined in the UML model. It is then possible that a specific interface is either defined in the same model as the component that uses it, or that it is located in a different model. In the first case, one can work entirely without references and let the *Port* point directly to the *Interface*. In the second case, one would have to work with a reference of the same name, as shown above. As a consequence, a port can either point directly to an interface, or to a reference that itself points to an interface. Figure 16.18 illustrates this:

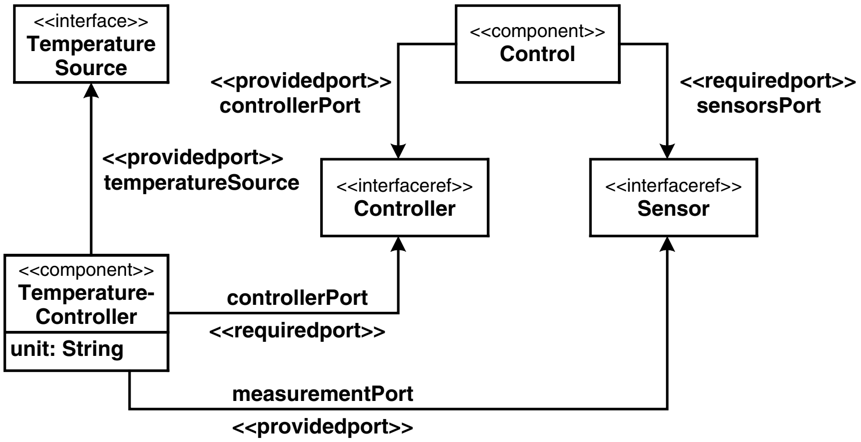


Figure 16.18 An example component model with *InterfaceRefs*

Thus a port is either associated with an *Interface*, or with an *InterfaceRef*. To solve this problem elegantly, we can build on the tried and tested principles of object-orientation and implement the Proxy pattern on the metalevel, as shown in Figure 16.19.

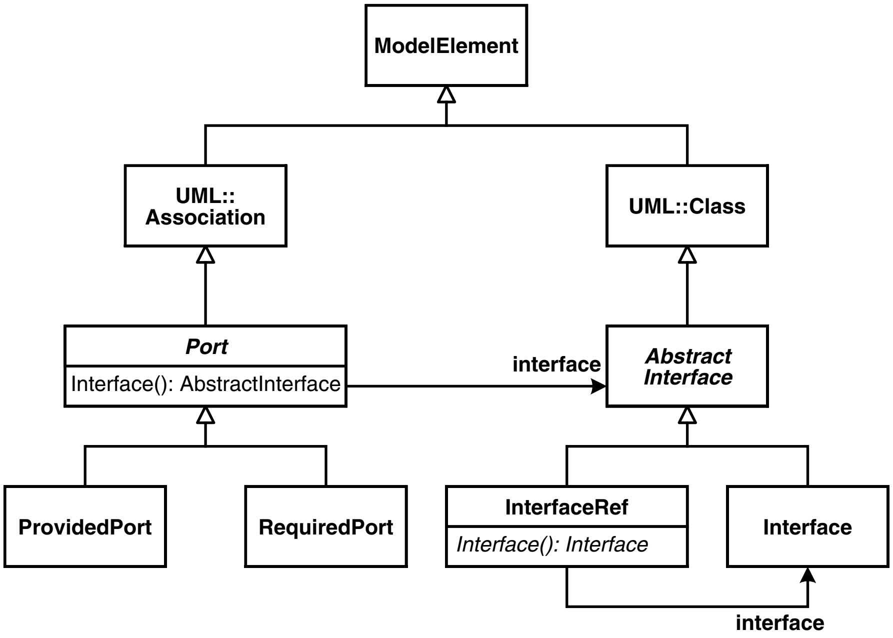


Figure 16.19 The Proxy pattern as basis for implementation of the *InterfaceRefs*

The operation *Interface()* of the port is interesting. Here we distinguish between a reference and an interface.

```
public abstract class Port extends Association {

  protected AbstractInterface AbstractInterface() {
    // return the "other" end's class of the
    // association
  }

  public Inteface Interface () {
    AbstractInterface f = AbstractInterface();
    if ( f instanceof InterfaceRef ) {
      return ((InterfaceRef)f).Interface();
    }
    return (Interface)f;
  }

}
```

The *Xpand* template from our previous example can be used 'as is'. The *Interface()* operation of the port always returns the interface and – if necessary – de-references the reference internally.

A further option for implementing reference proxys is to let the reference object (the proxy) implement all operations in such a way that they are delegated to the referenced object. If this is the case, a proxy can be used exactly like the object referenced by it. An explicit distinction – the class *Port()* in the operation *Interface()* – is then obsolete. However, when using this approach, you must make sure that all (metamodel-relevant) operations are actually delegated. If the referenced metaclass evolves, adaptation of the proxys can easily be forgotten. To address this problem, you can generate the reference proxys and the metamodel implementation automatically. The openArchitectureWare framework already includes such a meta-cartridge.

16.6.2 Polymorphism

The generator supports polymorphism at the template level: if several template definitions that are valid for different metaclasses exist in a template file, the generator always expands exactly that template with the type declaration that best matches the dynamic type of the model element. We can illustrate this with a simple example: in addition to configuration parameters, components can also have normal attributes that describe the component's state. Figure 16.20 shows an example and its representation in the metamodel.

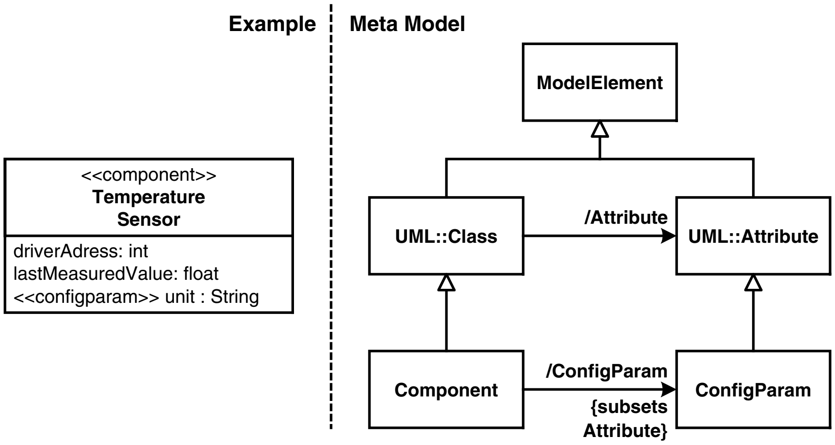


Figure 16.20 Configuration parameter: model and metamodel

Thus the set of all attributes contains normal attributes as well as the *ConfigParams.* The following *Xpand* template can thus be written:

```
«DEFINE Something FOR Component»
  public class «Name» {
    «EXPAND AttrDef FOREACH Attribute»
  }
«ENDDEFINE»

«DEFINE AttrDef FOR Attribute»
  private «Type» «Name»;
«ENDDEFINE»

«DEFINE AttrDef FOR ConfigParam»
  private String «Name»; // config param
  public void configure«UpperCaseName»( String value ) {
    this.«Name» = value;
  }
«ENDDEFINE»
```

In the third line this template iterates over all attributes and expands the template *AttrDef* for each attribute. If the current attribute is indeed a *ConfigParam*, the specific template is run. In this manner one can avoid the typical *if-isInstanceOf* cascades: the templates become easier to maintain as a result.

16.6.3 Separation of Concerns in the Metamodel

Utilities

The metamodel represents concepts of the problem space – so far, so good. However, often aspects that have nothing to do with the problem space that shows up in the metaclasses. Above, we used the property *UpperCaseName* from a template, for example. This means that the following operation must be present in the metaclass we are currently dealing with (in the example given above, *ConfigParam*):

```
public String UpperCaseName() {
  String n = // format Name().toString() appropriately
  return n;
}
```

We want to use this *UpperCaseName* property not only for *ConfigParams*, but for all model elements that have names. Further useful utilities also exist, for example the current date, the value of an incremental counter, and so on. It is not particularly elegant to implement all these operations in the metamodel, so that it is available everywhere – that is, in *ModelElement*. It would be better if one could ‘patch in’ these utilities on demand.

Precisely this is possible using *invokers*. If an non-existent property is called from a template, a previously-registered invoker – an implementation of the Interceptor pattern from [POSA2] – is called. The invoker now has the chance of returning a value for the respective property. The following invoker provides the *UpperCaseName*:

```
package genfwutil.mmutil.invoker;

public class UtilInvoker
       implements PropertyInvoker {

  public Object handleInvocation( Element element,
         Syntax syntax, Element name, String propertyName)
         throws EvaluationException {
    if ( propertyName.equals("UpperCaseName")) {
      String n = element.getNameProperty();
      return n.substring(0,1).toUpperCase()+
             n.substring(1);
    }
    throw new DesignError( "Property "+propertyName+
                           " not found on "+element );
  }
}
```

The generator calls the operation *handleInvocation()* on the invoker. This operation has various parameters, including the current model element (*element*) on which the property (*propertyName*) is to be called. If a property is required that cannot be provided by the invoker, it throws a *DesignError*.

To get this invoker to work we must make it known to the generation process. We do that – as always – by contributing it as part of a plug-in, together with the *CounterInvoker*, which provides a counter in the templates using the properties *CounterReset*, *CounterInc*, and *CounterValue*:

```
package util;

public class ECCodeGenPlugin extends GeneratorPlugin {

  public List contributeInvokers() {
   return makeList(
    new UtilInvoker(),
    new CounterInvoker()
   );
  }
}
```

Different Target Languages

A similar problem occurs if one needs to generate code for several target languages, as in the example here. We need utility functions in the metamodel for each of the target languages. In the case of Java, for example, *PackageDirectory*, which returns the relative directory for the *Package* of the current model element.

```
«DEFINE JavaImpl FOR Component»
 «FILE PackageDirectory"/"Name".java"»
 package «JavaPackage»
 public class «Name» {
  …
 }
«ENDDEFINE»
```

One can easily imagine that the metamodel implementation is quickly ‘polluted’ by all kinds of utility properties for the different target languages. A better structured solution must therefore be found. This solution is available in the form of a special invoker, the *SubpackageInvoker*. This invoker tries to instantiate another class and invoke the respective property via reflection. Let’s assume we have the following metaclass:

```
package ecMetamodel;
public class Component extends Class {
 // some Operations
}
```

If we now wish to implement operations that are specific to the Java platform, we can implement these in another class:

```
package ecMetamodel.java;
public class Component_Java implements StatefulDelegate {

  private Component component;

  public void setModelElement(ModelElement me) {
    this.component = (Component)me;
  }

  public String JavaPackage() {
    …
  }

  public String PackageDirectory() {
    …
  }
}
```

Since these operations are not implemented in the context of the metaclass, *this* does not point to the current model element, but to the helper object instead. The current model element is therefore provided to the helper object through the *setModelElement()* operation. The developer can now implement properties as if they were defined directly in the metaclass, but they must (explicitly) access the model element via *component* instead of *this*.

To actually use these helper classes, the *SubpackageInvoker* needs to be contributed in the same way as the *CounterInvoker* and *UtilInvoker* were registered in the examples on page 335.

```
public class ECCodeGenPlugin extends GeneratorPlugin {

   public List contributeInvokers() {
     return makeList(
       new UtilInvoker(),
       new CounterInvoker(),
       new SubpackageInvoker( "java", "Java" )
     );
   }
}
```

The *SubpackageInvoker* configured above tries to instantiate the helper class *p.q.java.C_Java* for a metaclass *p.q.C* and invoke the property operation on it. This makes it possible to delegate various aspects in the metamodel to separate classes and explicitly ‘contribute’ them to the model using the invoker.

16.6.4 Generation of Build Files

In the context of code generation, not only Java, C or C++ source code is created. The configuration files that configure the operating system platform must also be created. In most cases, real-time operating systems offer a large variety of configuration options to make the system as small and specific as possible for its respective application. Furthermore, makefiles and Ant files are created that compile, link, and package the generated code with the handwritten code, depending on the respective platform's requirements.

The following example shows the creation of *Jar* files. A custom *Jar* file is created for each container defined in the system, provided that we are dealing with a Java container:

```
«DEFINE BuildFile FOR System»
    ...
  «FOREACH Container AS c EXPAND»
    «IF c.LanguageID == "java"»
      <jar jarfile="«c.Name».jar">
        <fileset dir="${APP.BUILD}">
          «FOREACH c.UsedComponent AS comp EXPAND»
           <include name="generated.comp.«comp.Name»/**/*.class"/>
           <include name="manual.comp.«comp.Name»/**/*.class"/>
            «FOREACH comp.UsedInterface AS i EXPAND»
             <include name="generated.interfaces.«i.Name»/**/*.class"/>
            «ENDFOREACH»
          «ENDFOREACH»
        </fileset>
      </jar>
    «ENDIF»
  «ENDFOREACH»
    ...
«ENDDEFINE»
```

As you can see in the template excerpt, all interfaces that are used by the respective components are included as well.

16.6.5 Use of AspectJ

This section illustrates a possible integration of AspectJ in the MDSD production process, addressing the relationship between MDSD and AOP.

We use tracing as an example here. This is actually the standard example for AOP, yet its use in the world of embedded systems is quite realistic, because debugging on the target platform is often not easy. It is also important for us to be able to switch off any overhead created by tracing, if necessary, to save resources in real-life applications.

The idea is to define for which container tracing code shall be generated during system configuration:

```
<node name="outside">
  <container name="sensorsOutside" tracing="app">
      …
  </container>
</node>
```

The XML attribute can have the following values:

- *no*: tracing is switched off.
- *app*: only the application logic operations of the components are traced.
- *all*: all operations in the container are traced.

We also assume that only such code is generated that cannot be written more sensibly manually. We define an abstract aspect that contains the trace functionality as part of the platform. This aspect contains an abstract pointcut that we will concretize subsequently using code generation.

```
package aspects;

public abstract aspect TracingAspect {

   abstract pointcut relevantOperationExecution();

   before(): relevantOperationExecution() {
         // use some more sophisticated logging,
         // in practice
       System.out.println( thisJointPoint.toString() );
   }

}
```

This aspect creates trace information in all locations that are identified by the pointcut *relevantOperationExecution*. It is still defined as being abstract. During generation, we will inherit from it and define the pointcut. The following template shows how this could be done:

```
«DEFINE LoggingAspect FOR System»
    ...
  «FOREACH Container AS c EXPAND»
    «IF c.Tracing == "app"»
      «FILE "aspects/"c.Name"Tracing"»
      package aspects;
      public aspect «c.Name»Trace extends TracingAspect {
        pointcut relevantOperationExecution() :
          «FOREACH c.UsedComponent AS comp
                  EXPAND USING SEPARATOR "||"»
           execution( * manual.comp.«comp.Name»..*.*(..) )
          «ENDFOREACH»
        ;
      }
      «ENDFILE»
    «ENDIF»
  «ENDFOREACH»
    ...
«ENDDEFINE»
```

In consequence, the following code would be generated for the outer container:

```
package aspects;
public aspect sensorsOutsideTrace extends TracingAspect {
  pointcut relevantOperationExecution() :
    execution( * manual.comp.temperatureSensor..*.*(..) )
    ||
    execution( * manual.comp.humiditySensor..*.*(..) )
    ;
}
```

If we now use the AspectJ compiler instead of *javac*, this aspect is added automatically. It generates trace outputs before executing all operations of the code in the implementation package. *AspectJ* also allows for more expressive tracing, for example logging of the calling location.

The connection of MDSD generators and AOP can – as this example indicates – yield powerful solutions. Details about the interaction of MDSD and AOP can also be found in [Voe04].

17 Case Study: An Enterprise System

17.1 Overview

This chapter serves as an extended example for some of the advanced ideas in MDSD. This includes cascaded MDSD (Section 8.2.8), the integration of MDSD and CBD as explained Section 7.8, as well as the architectural process described in Section 13.4. This chapter is structured along the lines of that architectural process.

The example describes a fairly typical enterprise system that contains various subsystems such as customer management, billing, and catalogs. In addition to managing the data using a database, input forms and the like, we also have to manage the associated long-running business processes.

17.2 Phase 1: Elaboration

17.2.1 Technology-Independent Architecture

We decide that our example system will be built from components. Each component can provide a number of interfaces. It can also use a number of interfaces provided by other components. Communication is synchronous and is also restricted to be local: no remoting is supported on this level. We design components to be stateless.

Data types can either be simple types (*string*, *int*, *Boolean*) or complex. A complex data type is basically like a struct, in that it has named and typed attributes. There are two kinds of complex data types:

- Persistent entities that have a well-defined identity and can thus be searched and have relationships to other entities.
- Data transfer objects that have no identity and are not persistent.

In addition to components we also explicitly support business processes. These are considered to be expressible as state machines. Components can trigger the state machine by supplying events to them. In turn, other components can be triggered by the state machine, resulting in the invocation of specific operations defined by one of their interfaces. Communication between processes is asynchronous. Remote communication is supported.

17.2.2 Programming Model

The programming model uses a simple dependency injection approach, as exemplified by the Spring framework, to define component dependencies. An external XML file takes care of the configuration of the instances. The following fragment of code shows the implementation of a simple component. Note how we use Java 5 annotations. Component implementation classes are marked up with the *@component* annotation, whereas the setters for the resources use the *@resource* tag. The setters and annotations together make dependencies explicitly visible as part of the class signature, and don't hide them in implementation code.

```
public @component class ExampleComponent
  implements HelloWorld { // provides HelloWorld

  private IConsole console;

  public @resource void setConsole( IConsole c ) {
    this.console = c;            // setter for console
  }                              // component

  public void sayHello( String s ) {
    console.write( s );
  }
}
```

Processes described by state machines are implemented within a special kind of components, the *process components*. The state machine itself is implemented using the State pattern [GHJ+94]. To make the state machine triggers accessible for external clients, process components provide an interface that contains a void operations for each of the state machine's triggers (which can easily be sent asynchronously). They also define interfaces with the actions that those components can implement that want to be notified of state changes. These are also implemented as void methods, for the same reason. The following code shows the skeleton of a component that hosts a state machine. It has two triggers T1 and T2 and calls a single action on a resource component. It also has one guard that needs to be evaluated.

```
public @process class SomeProcess
                implements ISomeProcessTrigger {

  private IHelloWorld resource;

  public @resource void setResource( IHelloWorld w ) {
    this.resource = w;
  }
```

```
 public @trigger void T1( int procID ) {
    SomeProcessInstance i = loadProcess( procID );
    if ( guardG1() ) {
      // advance to another state…
    }
  }

  public @trigger void T2( int procID ) {
    SomeProcessInstance i = loadProcess( procID );
    // …
    resource.sayHello( "hello" );
  }
}
```

Since components are stateless, the process component shown above does not actually represent a specific instance of the state machine. Rather, it is an engine that can ‘advance’ any of the process instances. The actual process instance is loaded by the process component when a trigger is received. To identify the process instance on which we want to apply the trigger, the trigger operations contain a unique process id.

17.2.3 Technology Mapping

We want to keep the infrastructure for running the application itself as simple as possible. The Spring framework will be used as long as no advanced load balancing, management, or transaction policies are required. We will try to keep the technology mapping well separated from the application logic, so that we can easily move to a different technology platform should the need arise.

The following is the Spring configuration file for this simple example. It instantiates three Beans and ‘wires’ their resources accordingly.

```
<beans>
  <bean id="proc" class="somePackage.SomeProcess">
    <property name="resource">
       <ref bean="hello"/>
    </property>
  </bean>
  <bean id="hello"
        class="somePackage.ExampleComponent">
    <property name="console">
      <ref bean="cons"/>
    </property>
  </bean>
  <bean id="cons" class="someframework.StdOutConsole">
</beans>
```

Once a more sophisticated platform becomes necessary, we will implement stateless session EJBs to run the components inside a J2EE application server. The necessary code to wrap our components inside EJBs is easy to write: for each Bean, we write a remote/local interface, an implementation class that wraps our own implementation, and a deployment descriptor.

Persistence for the data entities is implemented using Hibernate. Persistent long-duration business processes are implemented along the same lines: for each process we implement an entity that contains all the data that represents the current state of the respective process: the id of the process instance, the identifier of it's current state, as well as the values of the context attributes.

We will use Web Services for remote communication between business processes. Since we transport rather simple trigger events implemented as asynchronous *oneway* methods, the mapping to the technology is trivial. We generate a WSDL file from business interfaces such as *IHelloWorld*, as well as the necessary endpoint implementation. We don't implement all the technology ourselves, of course, we use one of the many available Web Service frameworks.

17.2.4 Mock Platform

Since we are already using Spring as the technology mapping, we use that same platform to run the application components locally for test purposes. Stubbing out parts is easy based on Spring's XML configuration file. To make testing as easy and fast as possible, we use the Hypersonic in-memory database. Whenever we run a test, the schema is created anew – Hibernate can do this for us with one line of code. We discuss the mock platform further later in this chapter.

17.2.5 Vertical Prototype

The vertical prototype includes parts of the customer and billing systems. These parts of the system require both kinds of interactions: for creating an invoice, the billing system uses normal interfaces to query the customer subsystem for customer details. The invoicing process, including payment receipt and optional reminder management, is based on a long-duration process.

After implementing the vertical prototype, we execute a load test. This unearths two problems:

- For short-duration processes, the repeated loading and saving of persistent process state is a problem, so we add a caching layer.
- Second, Web Service-based communication with process components is a problem. We change communication to CORBA for remote cases that are inside the company – the external processes remain based on Web Services.

Note that for both changes, the application code does not have to be changed. We only need to change the adapters that map the logical communication to Web Services to be able to use CORBA.

17.3 Phase 2: Iterate

Spring is not just used as the MOCK PLATFORM, but also as the production environment. However, as a consequence of some new requirements, this has become infeasible. Spring does not easily support two important features: dynamic installation/de-installation of components, and isolations of components from each other, specifically with regard to using different classloaders. Both of these problems arise as a consequence the additional non-functional requirement that several versions of the same component have to run in one system to allow the evolution of the system over time.

As a consequence, the Eclipse platform is chosen as the new execution framework. The PROGRAMMING MODEL does not change, but the TECHNOLOGY MAPPING, however, has to be adapted.

17.4 Phase 3: Automate

17.4.1 Architecture Metamodel

A simplified metamodel for the system is shown below, which is rendered as a MOF model.

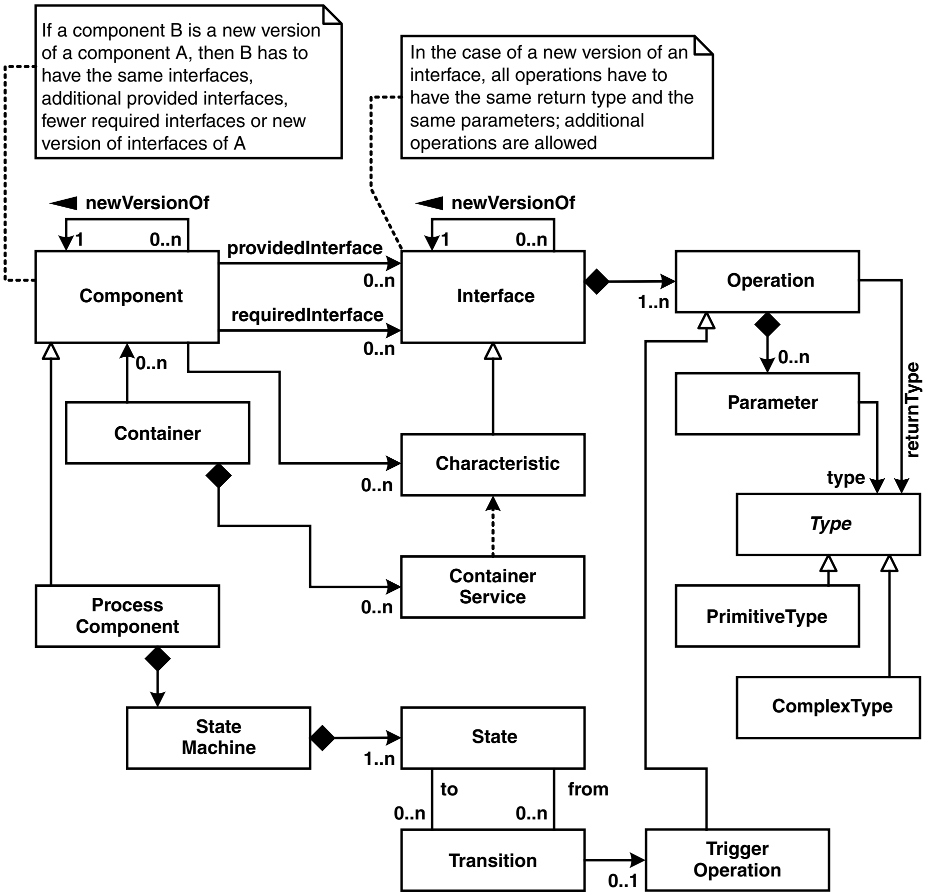

If a component B is a new version of a component A, then B has to have the same interfaces, additional provided interfaces, fewer required interfaces or new version of interfaces of A
In the case of a new version of an interface, all operations have to have the same return type and the same parameters; additional operations are allowed
◀ newVersionOf
1
0..n
◀ newVersionOf
1
0..n
Component
providedInterface
0..n
requiredInterface
0..n
Interface
1..n
Operation
0..n
Parameter
returnType
type
Type
PrimitiveType
ComplexType
0..n
Container
Characteristic
0..n
Container Service
0..n
Process Component
State Machine
State
1..n
to
from
0..n
0..n
Transition
0..1
Trigger Operation

Figure 17.1 The metamodel of our example system rendered in MOF

It is interesting to see that even the component container that hosts the components is modular with respect to its services. Characteristics (special kinds of interfaces) are used to mark up components with respect to the services they require. A container service such as persistence or lifecycle will take care of components that provide the respective characteristics interface. An example is shown in the following fragment of source code:

```
public @component class ComponentWithState
                implements IPersistentCharacteristics {

  private ComponentWithState_State state;

  // required by IPersistentCharacteristics
  public IEntity getPersistentState() {
    return state;
  }

  // required by IPersistentCharacteristics
  public void setPersistentState( IEntity state ) {
    this.state = state;
  }
}
```

17.4.2 Glue Code Generation

Our scenario has several useful locations for glue code generation:

- We generate the Hibernate mapping files from the entities.
- We generate the Web Service and CORBA adapters based on the interfaces and data types that are used for communication. The generator uses reflection to obtain the necessary type information.
- Finally, we generate the process interfaces from the state machine implementations.

In the PROGRAMMING MODEL we use Java 5 annotations to mark up those aspects that cannot be derived by using reflection alone. Annotations can help a code generator to know what to generate without making the programming model overly ugly.

17.4.3 DSL-based Programming Model

Components and Interfaces

There are several places where using a DSL makes a lot of sense, such as components, interfaces, and dependencies. Describing this aspect in a model has two benefits: first, the GLUE CODE GENERATION can use a more semantically rich model as its input, and second, the model allows for very powerful MODEL-BASED ARCHITECTURE VALIDATION, as described in Section 17.4.4.

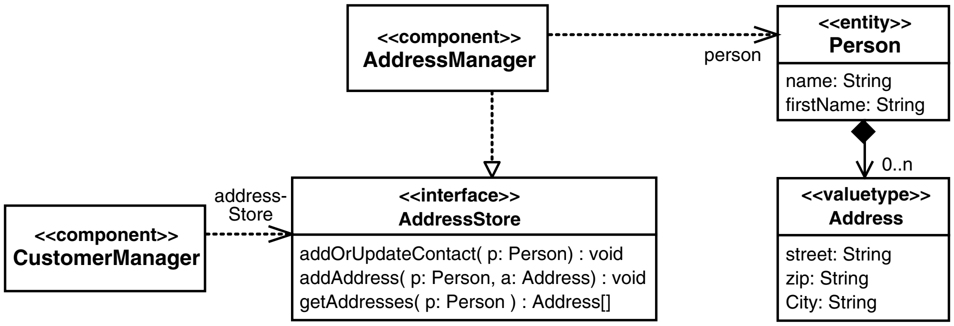


Figure 17.2 A very simple example component model

Figure 17.2 contains a logical model of two components, a shared interface as well as two data structures. From these diagrams, we can generate various things. From the components and their interfaces, we can generate a skeleton component implementation class, as well as all the necessary Java interfaces. Developers simply inherit from the generated skeleton and implement the operations defined by the provided interfaces. The following illustration shows this.

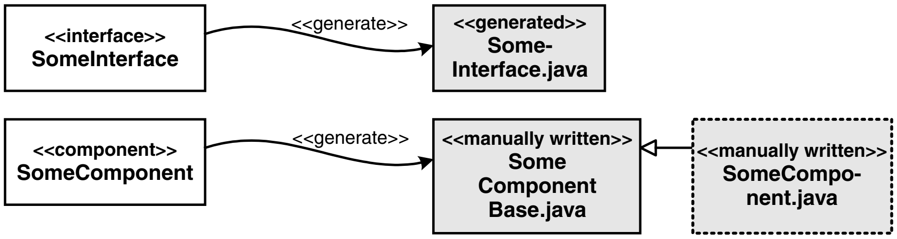


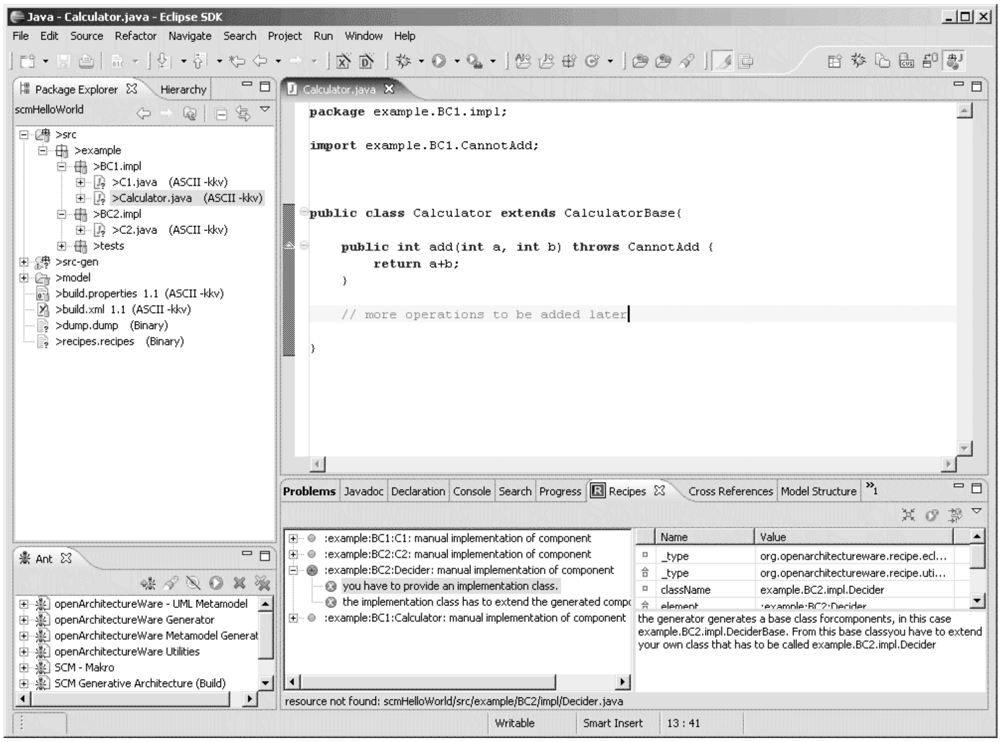
Figure 17.3 Code generated from interfaces and components

The programming model we describe requires developers to write implementation classes that extend the generated base classes. The problem is that this approach lays the burden on developers to do the right thing: if they forget to provide an implementation class, the system will not compile, or will fail strangely at runtime. To minimize these problems, we used a recipe framework (see Section 11.1.4) to further guide the developers after the code generator has done its work:

- We instantiate two checks for each component in the model. One checks the existence of a class that fits the required naming pattern: for a component *X* in the model, there has to be a class called *X* in the code, in the correct package. The other check verifies that this class actually extends the correct (generated) base class: for a component *X* in the model, the implementation class *X* has to extend *XBase*.
- These checks are stored in a file that accompanies the generated code.

- In the IDE, the IDE part of the recipe framework loads this file and evaluates the checks against the code base. Whenever the workspace contents change, the checks are re-evaluated. The view provides developers with an elegant way of viewing outstanding tasks that are required to make the software structurally complete.

Figure 17.4 shows how the IDE – in this case Eclipse with the openArchitectureWare Recipe Framework – renders the checks. The bottom-center pane shows three checks that have passed, because the component implementation class is available, and one check, for the Decider component, that has failed.


Java - Calculator.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Hierarchy
scmHelloWorld
Calculator.java
Problems Javadoc Declaration Console Search Progress Recipes Cross References Model Structure
Ant
Name
Value
Writable
Smart Insert
13 : 41

```
>src
  >example
    >BC1.impl
      >C1.java   (ASCII -kkv)
      >Calculator.java   (ASCII -kkv)
    >BC2.impl
      >C2.java   (ASCII -kkv)
    >tests
>src-gen
>model
>build.properties 1.1 (ASCII -kkv)
>build.xml 1.1 (ASCII -kkv)
>dump.dump   (Binary)
>recipes.recipes   (Binary)
```

```
package example.BC1.impl;

import example.BC1.CannotAdd;



public class Calculator extends CalculatorBase{

    public int add(int a, int b) throws CannotAdd {
        return a+b;
    }

    // more operations to be added later

}
```

```
openArchitectureWare - UML Metamodel
openArchitectureWare Generator
openArchitectureWare Metamodel Generat
openArchitectureWare Utilities
SCM - Makro
SCM Generative Architecture (Build)
```

```
:example:BC1:C1: manual implementation of component
:example:BC2:C2: manual implementation of component
:example:BC2:Decider: manual implementation of component
    you have to provide an implementation class.
    the implementation class has to extend the generated compo
:example:BC1:Calculator: manual implementation of component
```

Name	Value
_type	org.openarchitectureware.recipe.ecl...
_type	org.openarchitectureware.recipe.uti...
className	example.BC2.impl.Decider
element	:example:BC2:Decider

the generator generates a base class forcomponents, in this case example.BC2.impl.DeciderBase. From this base classyou have to extend your own class that has to be called example.BC2.impl.Decider

resource not found: scmHelloWorld/src/example/BC2/impl/Decider.java

Figure 17.4 The user interface of the openArchitectureWare Recipe Framework in Eclipse

Entities

Handling entities is a bit more interesting, as Figure 17.5 shows. First we generate the respective Java Bean (*SomeEntity.java*), including its Hibernate mapping file *SomeEntity.xbm.xml*. In addition to that, we want to have data access object (DAO) components for each of the entities. The DAO components provide operations to create, read, update, and delete instances of the respective entity. Instead of directly generating the code for these components from the entity, we use a model-to-model transformation to create model elements that resemble the DAO component and its interface. After that transformation, the model contains an additional interface and an additional component. These are treated just as any other interface/component – that is, the existing code generation template generates the Java interface as well as the implementation skeleton class from them. We don't have to write new templates!

What is still missing, however, is the implementation for the DAO components. Implementation code generally is written manually by developers into an implementation class that extends the generated implementation skeleton class. In the case of DAOs, however, we can also generate the implementation for their operations – these simply create, read, update, and delete instances of the respective entity – a couple of lines of (Hibernate) code.

So we now create an additional template that generates the implementation for the DAO components following the same rule as do the developers when they create their implementation class: the implementation class extends the generated implementation skeleton.

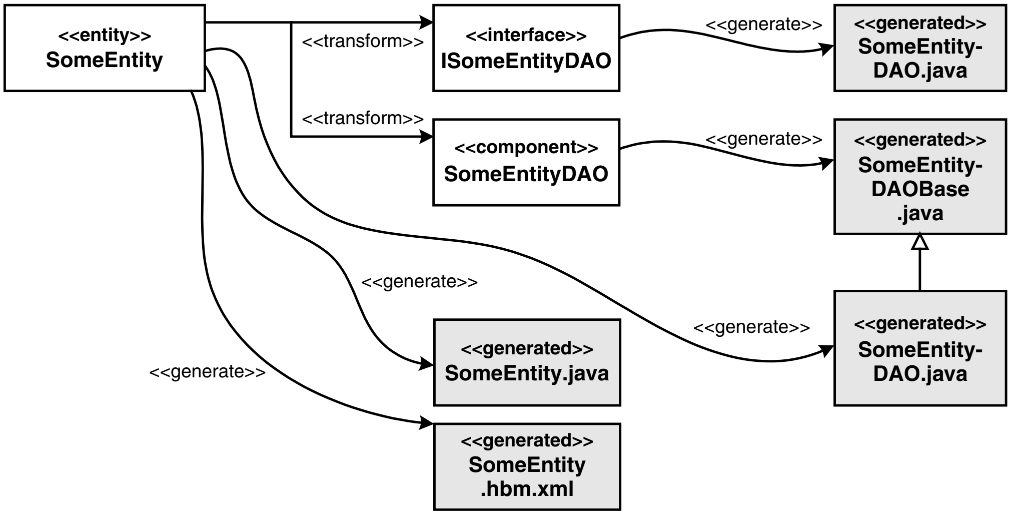


Figure 17.5 Handling entities

This approach has a number of advantages: we have to write fewer templates, we can reuse already-tested templates, and the DAOs and their interfaces show up as model elements in the model, not just as 'dumb' code files. This is important for the next aspect of the programming model, system modeling.

System Modeling

We now create our own models of how the system is composed and how it is deployed. This allows us to generate many more useful artifacts. Let's start with the composition model. We define various named configurations in this model. Each of these configurations, *customerStuff* and *addressStuff*, contains a number of component instances and their wiring. The test configuration is special, in that it does not define its own instances, but rather combines the two other configurations into a single one for testing. It is also interesting to note that we can create instances of the DAO components that we created from the entities. As we didn't merely create code for these components, but instead created real model elements by using a model-to-model transformation, we can now 'grab' this component and define instances of it. We couldn't have done that if we'd generated code directly from the entities.

```
<configurations>

  <configuration name="addressStuff">
    <instance name="am" type="AddressManager">
      <wire name="personDAO" target="personDAO"/>
    </instance>
    <instance name="personDAO" type="PersonDAO"/>
  </configuration>

  <configuration name="customerStuff">
    <instance name="cm" type="CustomerManager">
      <wire name="addressStore" target=":addressStuff:am"/>
    </instance>
  </configuration>

  <configuration name="test" includes="addressStuff, customerStuff"/>

</configurations>
```

The third model describes the system(s) onto which we deploy the configurations defined in the composition model.

```
<systems>

  <system name="production">
    <node name="server" type="spring" configuration="addressStuff"/>
    <node name="client" type="eclipse" configuration="customerStuff"/>
  <system>

  <system name="test">
    <node name="test" type="spring" configuration="test"/>
  <system>

</systems>
```

Here we define a system called *production* that consists of two nodes: one plays the role of the server, the other plays the role of the client. The *server* hosts the *addressStuff* configuration, the *client* hosts the *customerStuff*. Note that we also define the types of the respective nodes (*spring* for the *server*, *eclipse* for the *client*). From these two models, we can generate:

- A Spring configuration file for the server.
- The plug-ins needed for the client.
- The build files that create the necessary deployment artifacts.

- The remote communication infrastructure (CORBA, Web Services).
- Build files that assemble the necessary deployment artifacts.

If we actually generate the *test* system, then we get only a single Spring node onto which all component instances are deployed for unit testing. No remoting infrastructure will be generated. This is because we deploy the *test* configuration (which contains all instances) onto the single node in the system model.

Test Support

In the real-life case, we provided much more support for testing. For example, for each configuration, we generated a test case base class that would include all the set-up information to build the system as specified in the composition and system models. Developers could extend from these generated skeleton tests to implement test functionality.

Another really interesting way to support testing in this context is the use of mock objects. We integrated the EasyMock framework [EASY] in a very gentle way: the only thing we had to do to get a mock object for a specific component instance, instead of an instance of the implementation class, was to add the *mock="true"* attribute in the composition file:

```
<configurations>

  <configuration name="addressStuff">
    <instance name="am" type="AddressManager" mock="true">
      <wire name="personDAO" target="personDAO"/>
    </instance>
 </configuration>

 <!-- rest as before -->

</configurations>
```

As a consequence of the *mock* tag being set to *true* for the *am* instance, the context for the generated test now contains an EasyMock mock control object that can be used for testing. The following code fragment shows, in a simple test, case how these mocks can be used:

```
public class AddressManagerTest extends TestSystemTest {
  public void testAddressManager() {
    Address a = new Address( "Ziegelaecker 11", "89520", "Heidenheim" );
    setupMock(a);
    Person p = new Person();
    Address[] addresses = context().getAm().getAddresses(p);
    assertEquals( 1, addresses.length );
    assertEquals( a, addresses[0] );
  }

  private void setupMock(Address a) {
    context().getMockControlAm().reset();
```

```
      context().getAm().getAddresses(null);
      context().getAm().setReturnValue( new Address[]{a} );
      context().getMockControlAm().replay();
    }
  }
```

We also provided specific test support for the process components. For example, you could write assertions that checked that, after triggering the process component in a particular way, a specific action was called.

As a consequence of these test support features, it was possible to combine MDSD with test-driven development. Typically, developers would model components and interfaces, as well as a simple test configuration and system. From that, everything was generated except for the business logic and the test methods. Developers would then continue to implement the tests, often using the mock facilities. Finally, the implementation code was added until the tests were satisfied. Processes were developed in a similar way, building on the specific test support provided for process components.

Process Components

To complete the picture of model-to-model transformations and cascaded MDSD, let's look at how we work with process components and their machines.

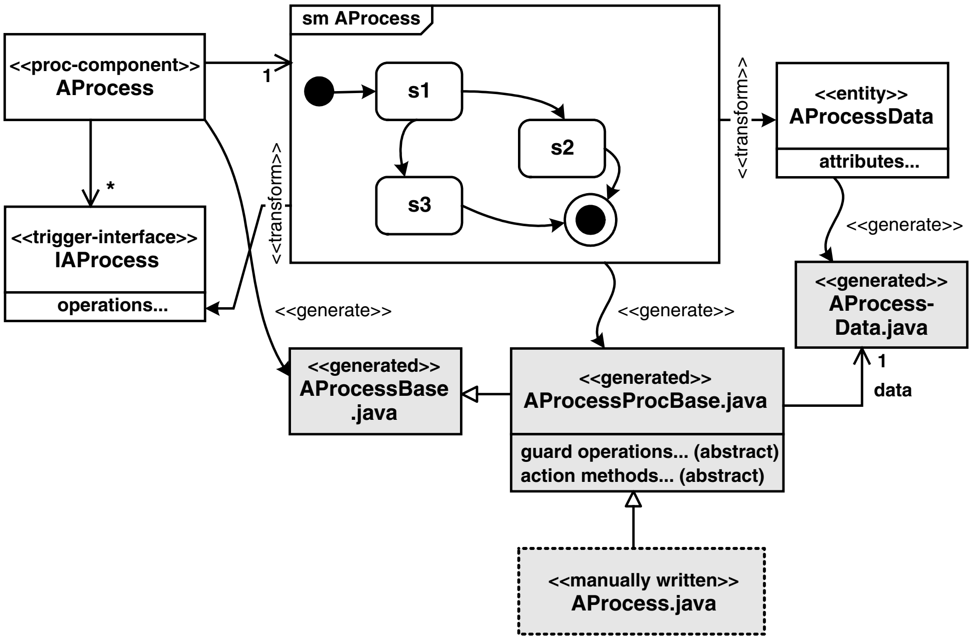


Figure 17.6 Handling process components

In this model we will create a process component called *AProcess*. This component provides an interface *IAProcess*. We will not model any operations in the interface – it's empty. We also associate the process component's state machine (*smAProcess*) with the component. Its state chart contains states, transitions, triggers, actions, and guards, just like any other state chart. The following process starts when the generator is run:

- From the triggers in the state chart, we add the necessary trigger operations into the empty interface using a model-to-model transformation.
- Using another model-to-model transformation, we create an entity that contains all the data necessary to describe the process instance described by the respective state chart.
- The mechanics that handle entities now 'grab' the entity and create the DAO component, its interface, the Java Bean, and the Hibernate mapping file. This is the identical process that was defined in the section on handling entities (page 348). No specific transformation or template has to be written.
- The interface for the process component is handled just as any other interface: a Java interface is generated from it.
- The component is handled like any other component: an implementation skeleton class is generated.
- We now need to provide an additional template that is specific to process components. Just as the implementation for the DAO components can be generated automatically, we can now generate an implementation of the process component (*AProcessProcBase.java*) that executes the process' state machine: we use a big *switch* statement for this. As the rules prescribe, this class extends the generated implementation skeleton class. However, since we have to add business logic to the action methods as well as to the guard operation, this generated class is still not complete: developers have to extend it and overwrite the guard and action methods. Again, the recipe framework is used to guide developers.

Again we have made heavy use of model-to-model transformations. While this approach might initially seem quite intricate and complex, it proved to be very useful in real life, because very little transformation code has to be developed. If we change the persistence mechanism from Hibernate to something else, the persistent process implementations are changed automatically, too.

The cascading of several levels of model-to-model transformations on top of each other allowed us to end up with a DSL for modeling business processes that was quite appealing to the business analysts that had to define the processes. The mechanics of integrating these project team members was as follows:

- The analysts initially created a state chart that described the business process intuitively, just as they were used to doing using state charts.
- The analysts were then joined by a developer. Together they marked up the intuitive state chart into one that was formal enough to serve as an input for code completion. This involved the application of stereotypes, formulating guards in a structured manner, as well as checking the chart for completeness. The code generator's verification facilities were used to check the state chart for completeness with respect to code generation.
- Further changes to the state chart were made mostly by the analysts by working directly on the formalized version. In some cases a developer was also involved.

This approach resulted in a much improved integration of analysts and developers, making the process of analysis a great deal more ‘tangible’ than before.

17.4.4 Model-Based Architecture Validation

Since the system will be built by a large number of developers, architectural constraint checking is essential. A number of basic model checks are done, to check for example that for all triggers in processes there is a component that calls the trigger. Other checks include dependency management. It is easy to detect circular dependencies among components. Components are assigned to layers (*app*, *service*, *base*) and dependencies are only allowed in certain directions. The programming model, based on dependency injection and inversion of control, combined with the fact that the component signature is generated from the model, prevents developers from creating dependencies to components that are not described in the model. Invalid dependencies in the model can also be detected easily.

Another really important aspect in our example system is evolution of interfaces. Consider the following diagram:

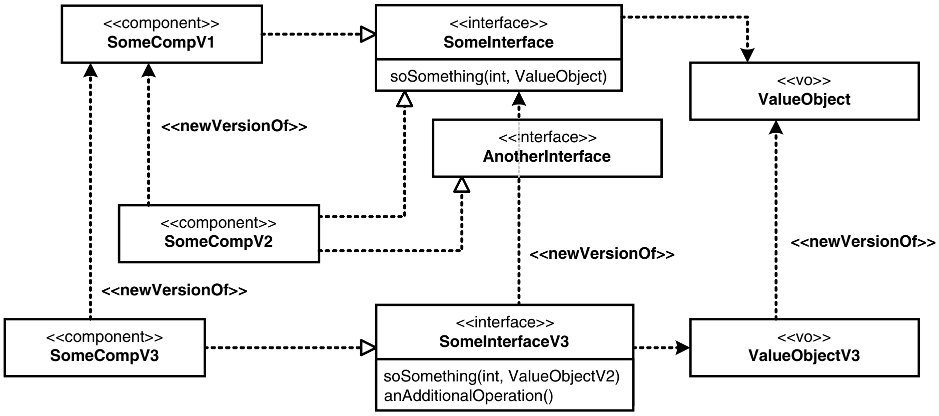


Figure 17.7 Component versioning example

Note how this diagram makes new versions of things explicit. This is essential to check and enforce compatibility rules that make sure that a client that expects *SomeInterface* can also deal with a new version, for example *SomeInterfaceV3*. The generated implementation of *SomeInterfaceV3* inherits from *SomeInterface*. This makes the interface types compatible. The generator also makes sure that a new version of an interface has the same operations (possibly plus additional ones). An interface can refine an operation by using a new version of a value object, the new version of which inherits from the old version. The verification phase of the generator therefore enforces rules that *make sure* that new versions of components and interfaces are always compatible with previous versions.

17.5 Discussion

This chapter illustrates a number of important concepts in the context of MDSD: we want to reemphasize some of them in this section. First of all, we show how cascading MDSD (see Section 8.2.8) can be used to cascade more abstract DSLs onto foundations laid by lower levels – in our case we cascade entities onto the basic components, and state machines onto components and entities. We also illustrate the role of model-to-model transformations in this context:

- The transformations in the actual project were implemented using ‘plain old Java code’, rather than some new ‘fancy’ model-to-model transformation language. We found that, while better languages might make the transformations more concise, the available tooling actually worked acceptably well. We see no pressing need for improvement in that area.
- Second, the transformation happened internally to the generator: the input model, which was based on UML and XML, was not changed to reflect the result of the transformation. We used generator-internal JUnit tests to verify that the transformations worked as they were expected to.

Finally, we hope that our explanations about testing support convince you that MDSD and agile software development – and test-driven development in general – are compatible and can be used together well.

Part IV
Management

This part of the book examines Model-Driven Software Development from the perspective of management. We understand management in general as the definition and adherence to strategies, goals, and measures in specific areas (besides the communicative aspects of management). We especially want to reach IT, architecture, and project managers.

Our elaborations cover economical aspects in the context of MDSD, as well as organizational constraints. We will also address outsourcing and offshoring. This part of the book concludes with a discussion of adoption strategies for MDSD that support the introduction of this paradigm into projects or companies.

18 Decision Support

With Jorn Bettin

This chapter explains the economic advantages and the investments that come with Model-Driven Software Development (MDSD) in general and architecture-centric MDSD in particular. It also attempts to answer both typical and critical questions.

You can find an overview of the motivation for, and basic principles of, MDSD in Chapter 2, Sections 2.1 to 2.3. We recommend you read those sections before this chapter.

18.1 Business Potential

MDSD combines the scalable aspects of agile approaches with other techniques to prevent quality and maintainability problems in large systems, as well as with techniques for the automation of recurring development steps in software development.

The potential of MDSD is based on a few basic principles:

- Formalization and condensation of software designs via the creation of modeling languages that are oriented more towards the problem space than the solution space.
- Abstraction from the level of expressiveness of today’s programming languages and platforms.
- Use of generators to automate repetitive activities.
- Separation of concerns, which to a large extent enables separate processing and evolution of functional and technical code.
- Reusability across project boundaries through the formation of software system families and product lines (see Section 4.1.4).

Based on these basic principles, a considerable number of features of the software systems developed in this manner can be derived and mapped to economic benefits as the following table demonstrates.

	Economic Benefits			
MDSD properties	Faster implementation of new business requirements.	Lower introductory costs of new technologies.	Reduced costs during the entire product lifecycle.	Strategic business advantage.
Use of expert knowledge	To realize new business requirements, domain knowledge rather than technical knowledge is needed.	Technology experts embed their knowledge into MDSD platforms and transformations so that it is available for application development.	In application development, fewer technology experts are needed.	Business knowledge is largely described in consistent models and thus rendered machine-readable. Technology knowledge is captured in the platforms and the transformations.
Automation in application development	Software production lines shorten the development time.	Implementation technologies can be adapted via automation at lower cost.	Less time and staff are needed overall.	Shorter time-to-market.
Securing application quality		Clear and formal separation of infrastructure code and functional code makes a technology change easier.	Significantly improved maintainability of architecture and technology aspects – beyond the boundaries of single software systems. Automation reduces potential for errors.	Reduction of maintenance cost, improvement of customer satisfaction
Extensive decoupling of technologies	Technology changes are largely limited to the platform and transformations. They do not affect the application models.	Each new technology mapping is only implemented once and centrally.	Less time and staff are needed overall.	New technologies that are relevant for the business sector can be used early on.
Use of formal application models	Changes and extensions are realized via the model. The existing production line is used without changes.	Application models are usually technology-independent and thus quite robust regarding technology changes.	Isolation of technology drift. The application's lifecycle is better decoupled from the technology's lifecycle.	A consistent and largely technology-independent software specification reduces the dependence on technologies and manufacturers.

Table 18-1 The economic benefits of MDSD

We now take a detailed look at the positive effects that MDSD can have on software projects.

18.2 Automation and Reuse

In a classical OO development process [JBR99] a design model is created incrementally and iteratively via a step-wise refinement of a part of the model that is established by projection to a few use cases. This model is refined to the point that it can more or less be transformed directly into an implementation.

Such a model can also be extracted automatically from an implementation by reverse engineering. We call this abstraction level an *implementation model*, because it contains all signature details of the implementation (but usually no more than this).

The diagram in Figure 18.1 shows the idealized development process of a development increment[1] from its analysis to its implementation. In time, both the level of understanding and the level of detail of the increment increase. At the start, more understanding is gained: towards the end, more details are worked out. A GUI design in the form of a sketch or a slide contains almost the same information as a completed implementation of the GUI using a JSP page or something similar. This means that the process of implementing the increment is primarily a task that requires tedious work and increases the increment's level of detail, yet it hardly improves the level of understanding. Nevertheless, the work is necessary in order to convert the essential information into a computer-readable form.

Whether this happens iteratively or not is of little relevance in this context. The overall effort basically consists of the progress in both previously-mentioned dimensions – that is, it corresponds to the area underneath the curve.

The disadvantages of an implementation model can be summarized as follows:

- The implementation model provides a poor overview, because essential information gets lost among the wealth of details.
- Such a model is relatively poorly suited when new team members need to be trained.
- The route from the results of the analysis to the implementation model is very long and there are no defined waypoints. Intermediate results yielded by different developers tend to be diverse. In their entirety, they hardly ever constitute a usable, complete model.
- Design changes are preferably carried out in the source code. Afterwards, the static implementation model is made consistent using reverse engineering. More abstract intermediate results are seldom maintained, particularly because it often unclear which effects a change of the implementation model may or should have on an intermediate result. The consequence is that only the static implementation model is up-to-date – if that. The dynamic aspects are neglected or even removed from the model because they no longer fit in. In most cases no consistent documentation exists at the end of the project that is any more abstract than the source code itself.

Figure 18.1 can be transferred to agile processes if a sufficiently abstract viewpoint is taken. The implementation model exists typically only in a virtual form – that is, in the form of source code. This avoids some disadvantages, of course, partly because specific artifacts are intentionally

[1] In the special case of a waterfall model, only one global 'increment' exists.

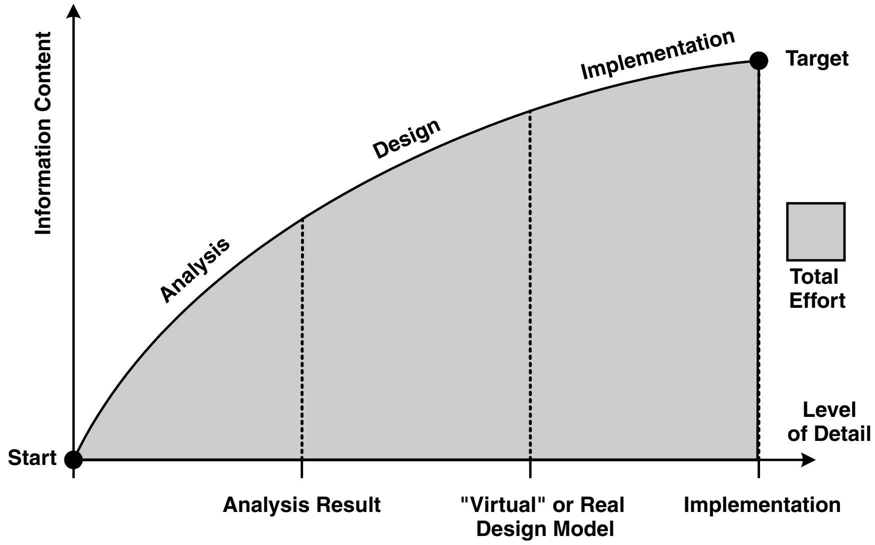


Figure 18.1 The effort required in traditional software development

omitted. However, as in a heavyweight process, the necessary level of understanding must be gained during development. The same is true for the functional requirements (analysis results) that are contained in the software or an increment. In other words: depending on the process, the milestones in the diagram may have other names or may not exist explicitly, yet the shape of the curve is basically the same.

For the sake of this discussion, we reduce the overall effort of software development to the factors of *information gain* and *level of detail*, and consciously ignore setbacks in the level of understanding that are brought about by changing requirements or new insights. We don't do this because these effects are irrelevant, but because they can be examined separately and independently of the issues described here.

We now introduce the essential potentials of MDSD, *automation* and *reuse*, to this discussion.

Here we clearly see the effects of abstraction, the modeling language's shift towards the problem space:

- Very compact models are created with a unified and formally-defined abstraction level that is much higher than that of an implementation model. For example, point A' is clearly to the left of the implementation model A, meaning that it is less cluttered with details.
- Those formal, MDSD models described here (at point A') contain significantly more information than the corresponding implementation models, due to the formalized semantics, particularly since not only signatures, but also partial implementations, are defined. For example, point A' is above implementation model A, meaning that its information content is greater.
- This is exactly where the potential for automation lies: a generator cannot obtain any information, but it can easily improve the level of detail. Thus the effect of automation in the

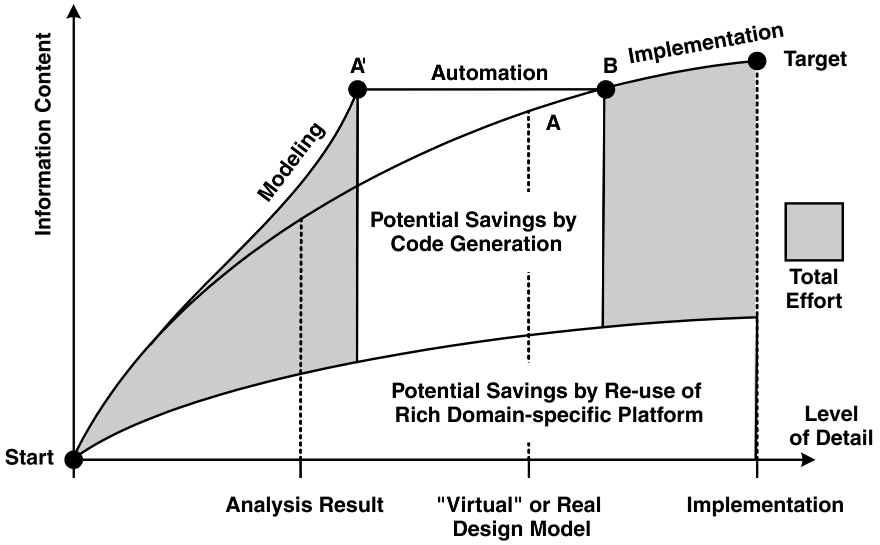


Figure 18.2 The effort in MDSD with partial manual coding

> diagram is strictly horizontal and leads to point B without further effort[2]. Depending on the domain and the concrete approach (see Chapter 4), point B is closer to, or further from, the target, determining the effort for the remaining (manual) coding.

In an extreme case, the model specifies the complete semantics of the (partial) application, so that no manual coding is necessary at all (see Figure 18.3). The circumstances in which this is possible or useful are discussed later, in Section 18.6.2.

A generator cannot help in increasing the level of understanding. All information must therefore already be present in the model, expressed using the domain-specific modeling language (DSL) – see Sections 2.3 and 4.1.1. Referring to Figure 18.2, the curve rises more steeply close to point A' than to point A. This means you have to put more work into understanding the domain - which has the positive side effect that it forces you to actually think about the domain.

Here, too, we are confronted with an extreme case: if the modeling language is very close to the problem space, the analysis result can be cast directly into a formal model – that is, point A' moves closer to the analysis result while containing the same amount of information, or it assumes the same position, thus proportionally increasing the automation potential. On the other hand, this also means that the analysis will have to be more formal (see Figure 18.3).

Taking an automotive engineering metaphor as an example: at your local car dealership, you fill in an order form that lists the vehicle type and any special features you want. The characteristics on the order form constitute the domain-specific language. You don't have to worry about implementation details, such as engine construction, when you order your car. The factory produces the desired product (your car) based on the domain model (your order form), using prefabricated components. Obviously, the engineering achievement lies in the

[2] We don't take the effort required for the creation of a generator into account here.

effort of building the production line and the mapping of your functional order onto this production line. The production of the item itself is automated.

The second important aspect about increased efficiency is *reusability* in the form of a domain-specific platform consisting of components, frameworks and so on. Here the effort is simply shifted from application development to the creation of the platform (see Figures 18.2 and 18.3), which is reusable. Such reusable artifacts can be used beneficially in almost all other development processes. However, their creation is usually not explicitly included in the methodology: in MDSD the platform complements both the generator and the DSL.

Now we can clearly see the potential for savings that is gained by automation in combination with a domain-specific platform. Figure 18.3 shows the extreme case. From the application development perspective it obviously offers maximum efficiency, but on the other hand a powerful domain-specific software production line must be established. However, to be able to leverage the advantages of MDSD to the full, some investments are necessary that must be weighed against the advantages. We discuss this topic later.

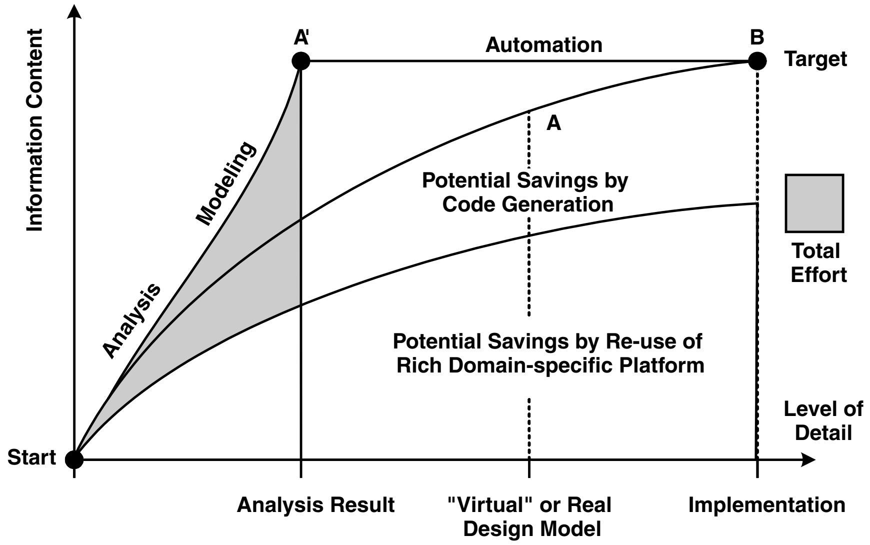


Figure 18.3 The effort in MDSD without any manual coding

In general the following advantages are gained compared to a standard development process:

- The modeling (design) phase ends much earlier.
- Only a fraction of the effort needed for creating an implementation model is required: the key to this is the domain-specific modeling language.

The implementation effort can be reduced considerably by using a generator that ‘understands’ the domain-specific modeling language. In the case of an architecture-centric approach (Section 2.5), a considerable amount of ‘classical’ programming remains, yet developers can focus on coding the functional requirements. The architectural infrastructure code is generated (Chapter 3) and also serves as an implementation guideline. A formalized, generative software architecture that includes a programming model emerges.

18.3 Quality

Until now we have focused entirely on how to increase efficiency in software development, but quality plays an equally important role, particularly because of its long-term effect. This section examines some of the factors that affect software quality and their connection with MDSD.

18.3.1 Well-defined Architecture

Code generation can only be used sensibly if the mapping of specific concepts in the model to the implementation code is defined in a systematic manner –that is, based on clearly-defined rules. It is mandatory to define these rules first. To reduce the demand for code generation, we recommend working with a set of rules that is as small and well-defined as possible. A small set of well-defined concepts and implementation idioms are the hallmark of a good architecture. We conclude that – in the context of MDSD – both the platform architecture and the realization of the application functionality on the platform must be well planned. MDSD demands a concise, well-defined architecture. This results in permanent consistency between model and implementation in spite of the model's high abstraction level. Suitable generation techniques guarantee this consistency in strongly iterative development (see Chapter 3).

18.3.2 Preserved Expert Knowledge

Modeling language, generator, and platform constitute a reusable software production line for a specific domain. Today's technical platforms, such as J2EE, .NET, or CORBA, offer a large number of basic services for their respective application fields – J2EE, for example, for big enterprise applications. Yet *using* these services – and hence the platform itself – efficiently is not easy. One must adhere to specific patterns or idioms to exploit the full potential of the platform: the platform must be used 'correctly«. All the patterns and idioms are documented and well-known in principle, of course, yet it is a big problem in larger projects to ensure that the team members, whose qualifications typically vary considerably, consequently apply the right patterns. MDSD can help here, because defined domain concepts are *automatically* implemented on the platform in *always the same manner*. Ergo, the transformations are design knowledge that has been rendered machine-processable. They constitute an inherent value, as they preserve the knowledge of how to use the platform effectively in the context of the respective domain.

18.3.3 A Stringent Programming Model

In today's practice it is not reasonable to generate 100% of the application code. As a rule, a skeleton is generated into which the developer adds handwritten application code. The platform architecture as well as the generated application skeleton determine where and how manually-created code has to be integrated and the manner in which this code is structured. This equips developers with a safeguard that makes it unlikely that they will create ‘big ball of mud’ systems.

18.3.4 Up-to-date and Usable Documentation

In the context of ‘classical’ software development, the application logic, mixed with technical code, is typically programmed against a specific platform in a 3GL language. This is also done, although to a lesser extent, when dealing with technologies such as EJB, where the application server takes over specific technical services such as transactions or security. Since many concepts are not visible at this abstraction level, the software must be documented using various means. In time-critical projects it is often impossible to keep this documentation synchronized with the code, and it is therefore usually the first victim if time starts to run out.

In the context of MDSD, the situation is altogether different: the various artifacts constitute a formal documentation of specific aspects of the system:

- The domain-specific model offers a superb overview, because recurring implementation schemas and details are factored from the model. It is therefore much more compact than an implementation model.
- The DSL and the transformations document the use of the platform and the models’ semantics.
- The models document the application structure and logic in a form that can be understood by domain experts. Besides the model itself, user manuals and other documentation can be generated from the application models and their descriptions. Generative creation guarantees that the documentation stays consistent with the code base beyond the first release.

These artifacts are always synchronized with the actual application, because they serve as the source for the application’s generation.

Of course MDSD also has a need for informal documentation. The DSL and its semantics must be documented to allow its efficient use in projects. However, this kind of documentation is only required once for each DSL, not for each application (see Section 18.4.).

18.3.5 The Quality of Generated Code

In general, generated code has a rather dubious reputation: barely readable, not documented, and with poor performance. However, this is an unfounded prejudice in the context of MDSD, because the quality of the generated code depends directly on the transformations (for example the templates). The latter are typically derived from a reference implementation (see Section 13.2.2). This means that the properties mentioned above, such as readability and performance, are propagated from the reference implementation into the generated applications.

This is also the reason why the reference implementation specifically should be developed and maintained with care. As long as transformations are created with sufficient care, the generated code’s quality will be no worse than that of manually-created code. On the contrary, generated code is usually more systematic and consistent than manually-created code, because repetitive aspects are always generated by the same generator, so they always look and work the same way.

For example, with today's generators it is no problem to indent the code sensibly. Comments can also be generated easily. One should pay attention to these things – the developers will be grateful.

18.3.6 Test Effort and Possible Sources of Errors

When MDSD is used a number of aspects have positive effects on the error rate of the software produced. This can be seen as early as your first MDSD project, but you will feel the effects mostly in subsequent projects and during maintenance:

- MDSD helps to automate tests via the use of generative methods for test script and test data generation (see Chapter 14).
- The generalization of schematic code in the form of transformations (or templates) enables a new approach to validation of the application architecture for to scalability, performance, and coverage of unusual application constructs. With little effort, a test application can be generated that is much more versatile than the manually-developed reference implementation, and which covers the extreme cases of the architecture's applicability. Thus weaknesses in the architecture can be detected early and can be cured by refining the reference implementation, followed by refining the templates. This positively affects the risk profile of projects and demonstrates that MDSD can sensibly be used in risky projects.
- Extensive tests with the generative test application eliminate an entire class of tests that would otherwise have needed to be tested with every single generated application.
- Since the code that can be generated typically makes up more than 50% of the whole code volume, potential error sources are significantly reduced through consequent automation. In the long run, the qualitative advantages of MDSD are as important as increased productivity.
- The generation of user manuals and other documentation based on descriptions and characteristics of application model elements is possible. Generative creation guarantees that the documentation stays consistent with the code base beyond the first release.

18.4 Reuse

Besides the potential for automation and quality improvement, a domain architecture also has a very high potential for reuse. Keep in mind that it is a software production line consisting of a domain-specific modeling language[3] (the DSL), a domain-specific platform, as well as a set of transformations that enable the step from model to runnable software, either completely or partially automated (see Section 4.1.4).

Such a production line is then reusable in similar software systems (applications) with the same properties. The definition of 'similarity' and its related potential for reusability is derived from these decisions:

- Definition of the domain
- Definition of the domain's implementation platform

[3] Compare for example the architecture-centric UML profile in Chapter 3's case study.

This makes MDSD interesting primarily in the world of software system families. In this context, a software system family is a set of applications based on the same domain architecture. This means that:

- They run on the same platform.
- They use the same DSL for specification of the applications – that is, of the ‘family member«.
- A common set of transformations exists that transfers the models into executable code based on the platform.

As an example, our first case study (see Chapter 3) covers the domain of architecture for business software with the MDSD platform J2EE/Struts. The modeling language described there is reusable for all software systems that use the features provided by this language, such as layering. The generative software architecture of the case study maps the modeling language to the platform. The transformations are therefore reusable for all software systems that have the same architectural features and will be implemented on the same platform.

Due to its level of abstraction, the reusability of DSLs typically is even greater than that of transformations and platforms. In the context of a business, it can for example be sensible to describe the business architecture via respective high-quality DSLs and map them to relevant platforms using transformations (generators).

In the case of a clearly-defined functional/professional domain such as software for insurance applications, a functional/professional DSL can be even more effective, for example to support the model-driven, software-technical realization of insurance products or tariffs.

A combination – or more precisely, a cascade – of functional/professional and technical MDSD domains is particularly effective: in most cases, the platform of a functional domain architecture can be very well realized with the help of an architecture-centric domain architecture (see Section 7.6). In this way the advantages of MDSD can be leveraged in both functional and technical dimensions.

Either way, the more applications or parts of applications that are created in such software production lines, the faster one will profit from their creation, and the greater will be the benefit.

18.5 Portability, Changeability

Fan-out is an important benefit of MDSD. This term describes the fact that a number of less abstract artifacts can be generated from a single model. The fan-out has two dimensions:

- On one hand, several artifacts can be generated from a single model in the course of a project. For example, relational database schemas, XML schemas, as well as serializers, can all be generated from a UML-based data model. In projects in which all of these different artifacts must be kept mutually consistent, this is a considerable advantage.
- On the other hand, different implementations can be generated from a single application model over time, making migration to newer versions of the technical platform, or to a totally new platform, easier.

An MDSD approach generally allows fast modification of the developed application(s). Since many artifacts are automatically created from one specification, changes to the specification will of course affect the whole system. In this way, the agility of project management increases.

18.6 Investment and Possible Benefits

We have seen the potential and usefulness of MSDS. Unfortunately, even in IT, there is no such thing as free lunch: to be able to enjoy the advantages, one must invest in training and infrastructure – of a technical as well as an organizational nature, depending on the desired degree of sophistication. This section introduces some experiences and conveys some condensed information from real-life projects to give you a clearer idea of the costs, the usefulness, and the break-even points in MDSD.

18.6.1 Architecture-centric MDSD

We recommend you first approach MDSD via architecture-centric MDSD, since this requires the smallest investment, while the effort of its introduction can pay off in the course of even a six-month project. Architecture-centric MDSD does not presuppose a functional/professional domain-specific platform, and is basically limited to the generation of repetitive code that is typically needed for use in commercial and Open Source frameworks or infrastructures.

The investment consists primarily of the training needed for handling an MDSD generator and its respective template language, as well as for the definition of a suitable DSL for modeling.

Effects on the Code Volume

To conduct a quantitative analysis, we took sample data from a real-life MDSD project at the time of its acceptance. This project was a strategic, Web-based back-office application for a big financial service provider. Table 18-2 shows in round numbers the effects of architecture-centric MDSD on the source code volume.

Amount of source code [kB]	**Traditional development**	**MDSD**
Source code reference implementation	1.000	1.000
handwritten code	18.800	2.200
Models		3.400
Transformations		200
Total	**19.800**	**7.800**

Table 18-2 Code-volume ratio in architecture-centric MDSD

To represent the influence of models on code volume, we selected the number of kilobytes required to store the all required UML files. We also assumed that a manually-created reference implementation would be developed to validate the application architecture.

Our concrete example for architecture-centric MDSD shows that the volume of code that needs to be maintained manually is reduced to 34% of the code volume that would have to be maintained in non-model driven development scenarios, including transformation sources! This number may appear to be very low, but in our experience it has proven to be representative. Tool manufacturers quite often publish data reflecting the percentage of generated code. However, these numbers are meaningless if it is unclear whether the necessary model and transformation sources, as shown in Table 18-2, have been included in the calculation. In our example, model and transformation sources make up more than 50% of the source code that needs to be maintained. Figure 18.4 provides a graphical view of the data from our example.

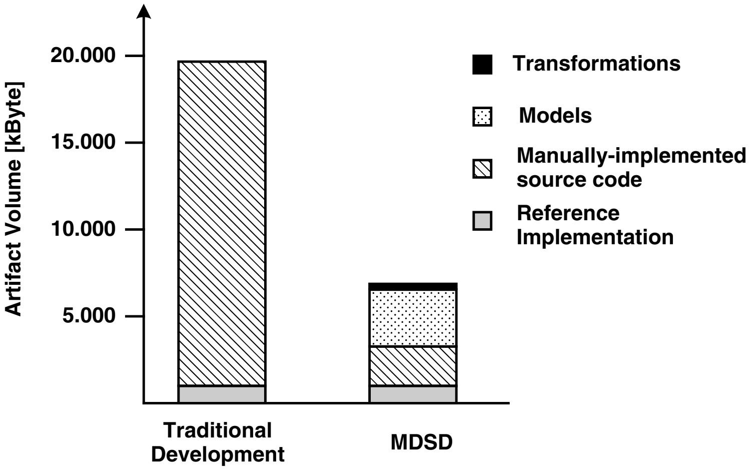


Figure 18.4 Code-volume distribution

Of course, other than in manual implementation, the volume of sources to be compiled remains unchanged if MDSD is used, as shown in Figure 18.5. The difference between traditional development and MDSD is that generated code doesn't constitute a source, but an effort-neutral intermediate result.

If we ignore model and transformation sources, as well as the reference implementation, we get a ratio of 88% generated code and 12% manually-created code. In our view, however, such figures are 'window dressing'. In MDSD, the models have the same value as normal source code, and the reference implementation should always be maintained in MDSD, because it is the basis of all refinements and extensions of the architecture.

Viewed from this perspective, the ratio of generated code and manually-created code is 72% to 28%. These figures are still impressive enough to make it clear that architecture-centric MDSD can pay off even if only a single application is developed with a given infrastructure – particularly if one takes into account medium-term maintenance requirements. To prove that

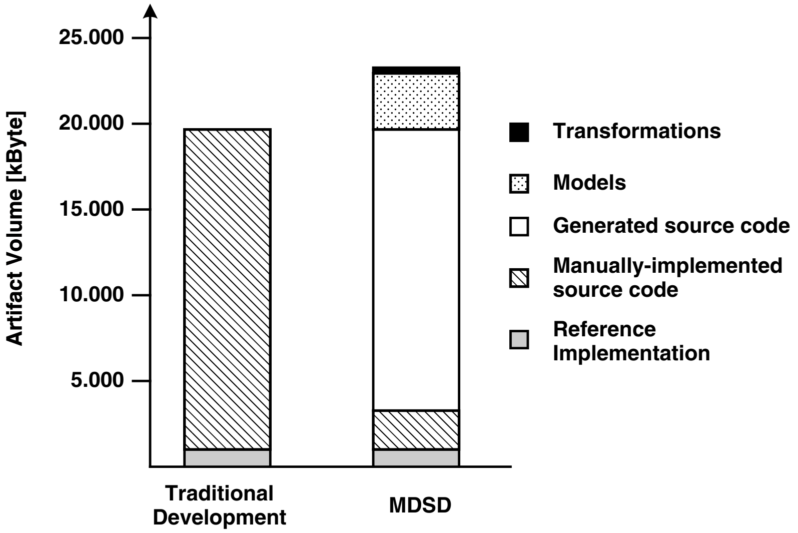


Figure 18.5 Ratio of generated 'sources' and manually created sources

our statement is true, we must consider the entire effort for the project's realization, not just the programming effort.

Effects on Project Time and Effort

Besides implementation, the project time and effort consist of:

- Analysis and documentation of the requirements.
- Architecture and design work, definition of the modeling language (for example UML profile), if applicable.
- Test generation and execution.
- Project management.

In addition, business process analysis preceding the project, required project documentation (user manuals, help menu texts), as well as production costs must be considered. MDSD has a positive influence on some of these activities, yet it is difficult to document this influence in general numbers. We therefore only examine in which respects MDSD affects the implementation effort, and how this again affects the total core activity efforts.

The figures given in the previous section are only useful for an assessment of the total work effort, because they do not reflect how much time – and thus money – is spent on the creation of models and transformations, as well as for the manual programming of reference implementations and application-specific source code. Likewise, the brainwork involved in creating the DSL is not considered here.

In a little practical experiment [Bet02] we examined how much time the creation and maintenance of models takes, based on data entry and mouse clicks. We then converted this information into the equivalent of source code lines using an approximate formula:

- If we define the total programming effort as being equivalent to the sum of all remaining manually-created source code lines, we get an interesting picture. In our experiment, MDSD causes a reduction of the programming effort to 48% of that of completely manual programming.
- If MDSD is not used and typical UML tools are applied instead to generate skeleton source code from the models, the programming effort increases to between 105% and 149% of that of manual programming completely without UML, depending on how many interaction diagrams the models contain (other than class diagrams, which don't contribute to skeleton generation).

These figures clearly demonstrate that MDSD is not the same as UML round-trip engineering. They also show why many software developers are skeptical regarding the use of UML tools.

In architecture-centric MDSD, the platform almost exclusively consists of external commercial and Open Source frameworks. The domain architecture that needs to be created consists primarily of a reference implementation and transformations derived from it (code generation templates).

The figures from Table 18-2's example (reference implementation circa 1,000 kB source code, transformation source code circa 200 kB) support our thesis that the derivation of transformations from a reference implementation takes only between 20% and 25% of the effort required for creating the actual reference implementation – especially if you keep in mind that most of the mental work has already been done for the reference implementation.

The effort required to create a reference implementation is not influenced by MDSD. For further discussions, we assume that the programming effort for the reference implementation constitutes 15% of a project's total programming effort. In the Table 18-2's example, the size of the reference implementation was only 5% of the entire application (measured in kB), so that even if one takes into account that the reference implementation is more difficult to program, 15% is a fairly conservative estimate.

Development effort [%]	**Traditional development**	**MDSD**
Transformations	–	4 (0,15*25)
Reference Implementation	15	15
Application models and code	85	41 (0,48*85)
Total	**100**	**60**

Table 18-3 Comparison of modeling and programming effort

Table 18-3 shows that architecture-centric MDSD can lower the programming effort by 40%, given the conditions described above. This figure confirms our practical experience as well, and is a good reference for calculating the costs of introducing MDSD. It is best however to use metrics from your own team to calculate how much programming effort is needed compared to the total project time and effort, and thus to calculate the potential of MDSD.

- If we assume that programming makes up 40% of the project activities, the potential for saving money can be up to 16%.
- The effort for the reference implementation and transformation development is not necessary for subsequent projects, so that the programming effort is reduced by 59% and potentially up to 24% of the costs can be avoided.
- A mature domain architecture increases the maintainability and changeability of all projects using MSDS, so that the maintenance cost is significantly reduced[4].

In real life, of course, the teams' learning effort, as well as the hiring of an MDSD expert as a coach for the first project, must also be considered.

We can make an empirical assessment: if an MDSD pilot project runs for six months or longer and the team consists of more than five people, the introduction of MDSD can pay off as early as during this first project, even if the whole production line must be built from scratch. Chapter 20 describes the adaptation strategies and prerequisites that should be observed.

18.6.2 Functional/Professional MDSD Domains

As you may already have inferred from the previous section, architecture-centric MDSD offers a simple and low-risk adoption path for MDSD. As the figures in our examples show, roughly half of the sources to be maintained consist of models and transformations: the other half consist of 'traditional' source code for the reference implementation and handwritten application logic.

If a mature implementation of architecture-centric MDSD is available, the remaining traditional source code cannot be reduced any further. Efficiency can only be increased by applying the MDSD paradigm not just to the *architectural/technical domain*, but also to *functional/professional domains*. This allows us to uncover *functional domain-specific* commonalities and variabilities of applications via domain analysis or product-line engineering (Section 13.5) and to increase the abstraction level of the application models even further: the models then describe domain-related problems and configurations, instead of architectural aspects, effecting a further reduction of the modeling effort and particularly the effort for manual coding.

To exemplify this it helps to look at the code volume distribution during development of a number of applications in the same domain. Figure 18.6 sketches the reduction of code volume via introduction of a functional domain architecture during the development of three applications, while assuming that the application models and application-specific, handwritten source code can be reduced by 50%. Note that Figure 18.6 is of a purely illustrative nature: the real-life facts very much depend on the domain and its complexity. The clearer can the domain's boundaries be defined, the more functionality can be integrated into the functional MDSD platform in a reusable form.

In practice, the development of a functional domain architecture is an incremental process based on architecture-centric MDSD.

The development effort for functional domain-specific languages (DSLs) and the corresponding frameworks should not be underestimated. The successful development of such languages

[4] It is difficult to express this effect in figures.

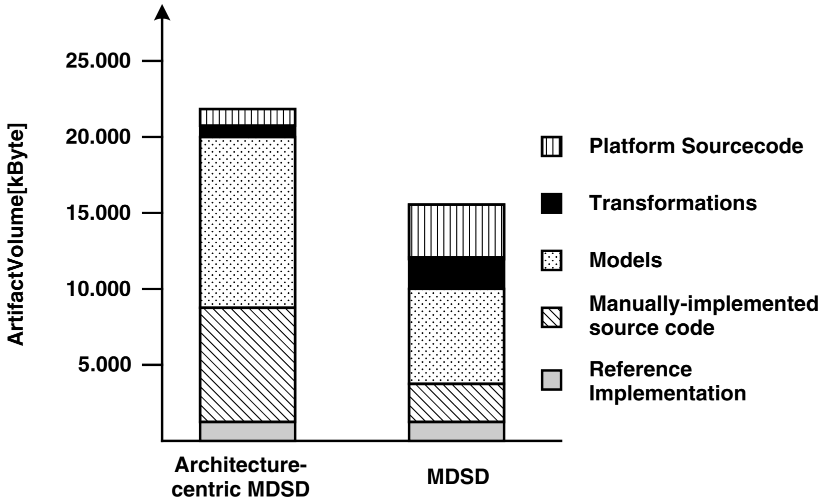


Figure 18.6 Code-volume distribution during introduction of a functional domain architecture

requires significant experience in the respective domain and should not be attempted in your first MDSD project.

There are also scenarios in which functional frameworks already exist that can be used without the help of model-driven generators, of course. In this case, MDSD can build on them as in the architecture-centric case – that is, the existing frameworks are regarded as the MDSD platform that defines the target architecture (see Chapter 7). A DSL is then derived from the configuration options. As in the architecture-centric case, productivity can be improved remarkably for relatively little investment.

18.7 Critical Questions

We have addressed some of the prejudices against MDSD in the preceding sections, as well as in Chapter 5. Some of these prejudices date back to the 1980s and 90s and stem from negative experiences with CASE tools, and are now projected onto model-driven approaches in general. Serious and important questions emerge: the answers to such questions are neither trivial nor easily found. Below we discuss answers to some of these questions:

- *What is new about MDSD?*

  Naturally, its individual concepts, for example the generation of source code from higher-level descriptions/models, are nothing new. Nevertheless, the simple code generators that every developer uses or writes at some point in their career are not comparable with the highly flexible MDA/MDSD tools of the latest generation, which allow you to define modeling languages and modular transformations both freely and in a relatively simple manner. MDSD is more than just a technique – it is mainly about the engineering principle.

- *If the approach is that brilliant, why isn't it more popular?*

  The required prerequisites, such as flexible tools and – most of all – a more comprehensive knowledge about the approach are not yet available. In the field of tools, the growing popularity of MDSD certainly plays a driving role. On the other hand, the integral architecture-centric MDSD view – and particularly its generative use – are already established among early adopters. The same is true for central, process-related MDSD concepts.

- *Does MDSD have negative effects on the performance or the readability of application source code?*

  In the model-driven approach, the necessary infrastructure and the corresponding design patterns are developed initially in the form of a small, manually-created reference implementation. This reference implementation may be small, but it is deep. It covers all the architecture's layers and is established in the first phases of a project by experienced team members. In the course of the project, the reference implementation is refined and optimized, so that its properties and improvements can be transferred to the entire source code via generative techniques. On average, the model-driven approach leads to a performance that is at least as high as if a traditional development approach is used. The same is true for source code readability (see Section 8.2.4).

- *Aren't today's UML-based IDEs with round-trip support much more mature than MDA/MDSD tools?*

  This may be correct to a certain extent, yet it is irrelevant, because these tools constitute a different class of tools. Due to their missing abstraction level, they do not offer the advantages of MDSD described above (see Section 18.2 and Chapter 5).

- *Doesn't MDSD create an unreasonably strong dependency on a specific tool or a certain technology combination?*

  Modular, automated transformations and the treatment of model transformations as first-class artifacts guarantee that tool dependencies remain local (see Section 11.2). Since MDA has not yet established a standard for transformation languages, one is bound to a specific transformation language by choosing an MDA/MDSD tool. However, MDSD tools can more and more be considered as everyday tools, especially if they are available as Open Source software. Since modern template-based generators have no restrictions over target languages, there is no limit to specific technology combinations for the application to be created.

- *Doesn't the model-driven approach imply a waterfall model, and doesn't it particularly conflict with agility? What distinguishes it from CASE?*

  The iterative, dual-track process of MDSD (see Section 13.2), in which the infrastructure is developed in parallel to the application(s), must be clearly distinguished from traditional waterfall methods that are based on a 'big design up-front' philosophy. MDSD is based on domain-specific platforms and is in strong opposition to the CASE approach that tries to anticipate all situations by using a universal language. Agile methods are particularly well-suited for the creation of MDSD reference implementations. MDSD also helps to scale agile methods, because the experience and expert knowledge gained can be made available to all developers in software form. MDSD fosters test-driven approaches (see Chapter 14).

- *How can one ensure that model and code do not diverge at some point?*

  In MDSD generation is based on the model, and generated code is consequently separated from non-generated code. Handwritten code is not obsolete. On the contrary, MDSD considers developers and generators to be complementary partners that must work together efficiently. The rules for this collaboration between developers and generators are not rigid, but are defined based on the actual requirements. The same applies to the rules for the developers' interaction in a team. However, the MDSD generator ensures that the developer does not cross the boundary set by the domain architecture, especially since manipulations of the generated code are undone during iterative regeneration.

- *How much time and effort is required for generator or transformation development? Is is worth the time and money?*

  Well-focused, yet mature generic tools are available, particularly in the Open Source field. These tools possess simple yet powerful template languages. Some also define powerful model-to-model transformation languages. One should build on these generators during transformation development.

  Modern template languages allow a very intuitive generalization of complex, handwritten code. The effort for the generalization or 'templating' of the source code is only 20% to 25% of the effort needed to code manually the reference implementation from which the templates are extracted. From a certain project size upward, and especially when similar subsequent projects are to be expected, the MDSD approach is much more efficient than traditional software development (see Section 18.6).

- *Doesn't it take longer to change an attribute type in the model, followed by regeneration, than to change the declaration in the code in a certain place manually?*

  If we have rich semantics, for example persistence of the attribute, changing one occurrence in the code is not enough; all occurrences in interfaces (getters/setters), serializers, and DDL scripts must be adapted consistently.

- *Isn't it faster to adapt changes due to maintenance by adapting the generated code manually instead of adapting the MDSD transformations (the generation rules)?*

  If the guidelines for MDSD are obeyed and generated code is kept strictly separate from non-generated code, changes to code templates are always faster than manual changes to numerous source files.

- *In time-critical phases of application development, one cannot always wait for the next release of the domain architecture. Manual interventions into the generated code are the consequence, and the MDSD approach fails because the generator can no longer be used. How can this be avoided?*

  For smaller patches it can make sense to create a specific, temporary branch of the domain architecture that is decommissioned during the domain architecture's next release. Should severe insufficiencies of the domain architecture emerge, one must fall back on manual programming, either temporarily or permanently. This is not accomplished by manipulating the generated code, but by opening the domain architecture in the relevant place: the transformations are changed in such a way that they generate only a (modifiable) default implementation. The opening-up of the domain architecture can – if applicable – also be made available as a temporary, project-specific branch.

- *Isn't it better to implement an object-oriented framework instead of using a generator? Do object-oriented generators and high-quality, object-oriented generated code exist?*

  Object-oriented frameworks and generators are an ideal match (see Chapter 3). The combination of both approaches is a real step forward.

  In MDSD, frameworks and generators complement each other in a similar way to the dynamic and structural aspects of an object-oriented model. Modern generators are typically realized in object-oriented languages and are in no way comparable to the simple utilities that every programmer has written at some point.

- *Model-Driven Software Development in larger teams inevitably means distributed modeling. Aren't the currently-available (UML) modeling tools a problem rather than a help?*

  Of course choosing the right modeling tool is essential to the success of MDSD in big projects. Yet there are also mature (UML) tools that support distributed modeling even in large teams. Independently of this approach, the simple partitioning of the application model into several loosely-coupled partial models can be helpful. Partitioning can take place either horizontally – alongside the architectural layers – or vertically – alongside loosely-coupled use case implementations. Partitioning must be allowed and supported by the DSL and the generator (see Chapter 15).

- *Isn't it much more time-consuming to work out an architecture explicitly?*

  Each application has an architecture, whether formally defined or not, but only if it is explicitly worked out (in whatever form) does it becomes a 'good' architecture. Quality attributes such as maintainability, scalability and so on can be created iteratively, but not accidentally.

- *Isn't handwritten code more reliable than generated code? Can one trust a generator in all situations, or won't there always be situations in which manual interventions are required?*

  It is wrong to assume that a couple of unpredictable exceptions of a design pattern render the use of generators impractical or uneconomic. It is not the goal of MDSD to eliminate the manual creation of source code altogether. On the contrary, MDSD offers pragmatic techniques for supplementing generated code with non-generated code in well-defined locations.

  As proof of the higher efficiency and lower error rates of handwritten code, strange compiler or assembler errors are sometimes quoted. The fact is overlooked here that this supposed weakness of generator technology would have prevented the development and usage of today's programming languages a long time ago. The opposite is true: practice shows that typically many errors are discovered during the derivation of templates from the reference implementation, due to the intense attention paid to implemented design patterns. Such errors would otherwise remain unnoticed and spread further via traditional copy and paste techniques.

  We admit that the borderline between generated code and manual implementation must be drawn with great care. It is a balancing act between automation and necessary degrees of freedom. Often it is the limited number of solution alternatives that will let you achieve your goal (see Chapter 7).

18.8 Conclusion

MDSD enables productivity and quality gains as soon as during your first model-driven project. Specifically, the implementation effort can be reduced by half compared to traditional manual programming. Considering the necessary training effort for the introduction of MDSD, real savings are to be expected from the second project onwards once the team is familiar with the MDSD paradigm, a concrete set of tools, and the methodology. The use of MDSD with functional/professional MDSD platforms should be the second step (*cascaded* MDSD, see Section 8.2.8), which is recommended particularly for the development of product lines (see Section 13.5).

18.9 Recommended Reading

Unfortunately the amount of publicly-accessible data on the efficiency of Model-Driven Software Development and product line development is rather small. This has several causes: on one hand, no-one will come up with the idea of executing a real-life project twice in parallel just to collect comparative metrics – and even this would be only of limited value, due to the dissimilar boundary conditions. Furthermore, positive effects on time and quality are obvious to everyone involved in the project, while interesting metrics relate more to the incremental refinement of the approaches than to a comparison with completely manual approaches. Last but not least, not all companies mention the option of using generative techniques and the resulting savings if the customer is only interested in the delivered, traditional source code or the finished application.

The sources [PLP], [Bet02] and [Bet04c] contain further data and economically-relevant statements regarding the MDSD development process.

19 Organizational Aspects

with Jorn Bettin

To use Model-Driven Software Development, it is necessary to distinguish between the development of concrete applications and the development of the domain architecture. The reasons for this distinction are practical: first, concrete projects must be finished at a certain point to meet deadlines, so possible technical problems in domain architecture development cannot be considered. Second, the domain architecture's quality is decisive for the success of MDSD. We advise against forcing compromises on the domain architecture's design when under pressure. Third, the knowledge needed for developing the domain architecture differs from the knowledge needed for application development. Thus the two aspects are separated. The goal of organizational best practices for MDSD is to create an environment that supports a useful assignment of tasks.

MDSD is scalable and well-suited for use in larger, distributed project environments. Distributed software development is not only an issue in very large projects, but increasingly also in the context of *offshoring*.

19.1 Assignment of Roles

This section addresses the MDSD-specific assignment of roles. The development process sketched in Chapter 13 serves as our guideline here.

19.1.1 Domain Architecture Development

For successful domain architecture development, both profound knowledge and experience of the domain are required. These are usually dispersed through a group of people. Some of the required roles do not differ from those in traditional software development projects, while other roles take on a special meaning in the context of MDSD. We will primarily discuss the latter, not to offer a kind of job description, but rather a representation of various activities that must be carried out in a real-world project. One person can assume more than one role in a project, of course, and similarly one role can be taken on by a number of people.

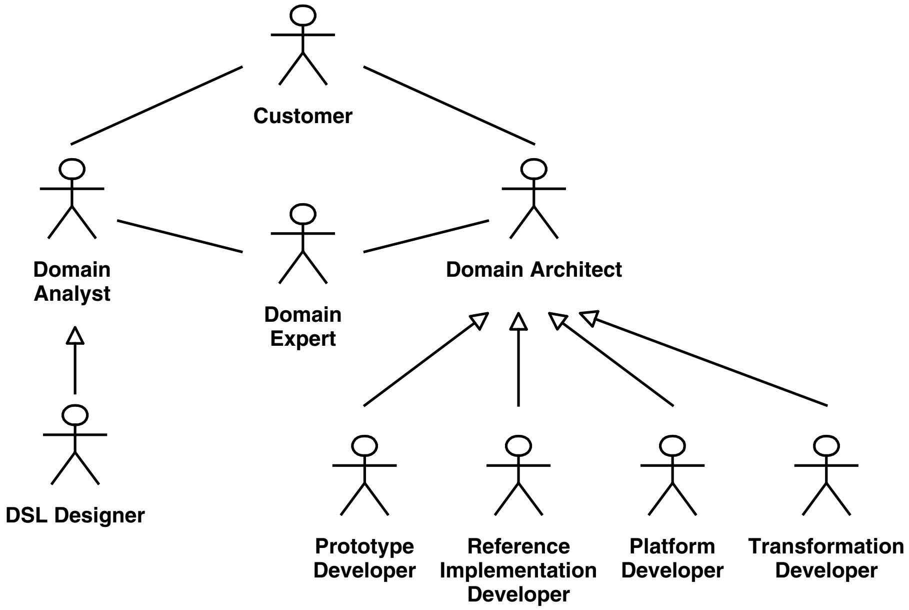


Figure 19.1 MDSD-specific roles in domain architecture development

Domain Experts

Expert knowledge about the MDSD domain is needed for the development of a good domain architecture (see Section 4.1.1) – that is, knowledge about the area that is to be supported by MDSD. We consider such people to be either domain experts who in the past have realized at least two applications in the domain, or potential users who are experts on parts of the domain. For example, for the development of an insurance application, jurists specializing in damage claims, actuaries, direct marketing experts, and so on can be consulted as domain experts. (Note that in architecture-centric MDSD, the domain experts are software architects who are experienced in the target architecture's technology – see Chapter 7).

Expert knowledge is relevant for traditional application development, but it is essential for developing a domain architecture. This is one reason why domain architecture development is rather badly suited for offshoring, at least during project bootstrapping.

Domain Analysts

In the section on product-line engineering (Section 13.5) we discussed the various aspects of domain analysis. The analysis of commonalities and variations within a domain are best worked out iteratively via a number of workshops with domain experts, product managers (if applicable), customers, requirements analysts, and domain architects.

The Language Designer

The language designer is a special type of domain analyst. On one hand they are responsible for the development of metamodels and UML profiles: on the other, for the development of other concrete kinds of syntax (see Section 4.1.1). In other words, they define domain-specific languages (DSLs). To this end, the language designer works closely with domain experts and other domain analysts. On the implementation side, the language designer is the decisive link with the domain architects, especially because the language designer also creates the reference models that match the reference implementation (see Section 13.2.2).

Domain Architects

Domain architects are responsible for the realization of domain architectures (see Part II and Chapter 13). This especially includes the definition of the target architecture (see Chapter 7) and the development of subsystem structures within the domain architecture. The latter is done in cooperation with the language designer. Specific flavors of domain architects can be distinguished, analogous to the various parts of a domain architecture:

- *Prototype developers*. Prototype developers create technology prototypes. In most cases, they are technology experts or work closely with such. Other than traditional development, MDSD allows for a very economic use of technology expertise – it can be transferred to application development via the generative approach, leading to a reduced need for coaching there.
- *Reference implementers*. Reference implementers map the DSL's constructs to the target architecture based on the prototypes. They use the reference models from domain analysis (see above) as their specification for this purpose.

  Prototypes have a rather experimental character, whereas reference implementations have a very formal character. This is because the transformations must be derived from them.
- *The platform developer*. We discussed the significance of domain-specific MDSD platforms in Sections 4.1.2 and 7.6. People with experience in framework development are particularly well-suited the role of platform developer. Platform developers must collaborate closely with reference implementers because the MDSD platform constitutes an important partition of the reference implementation. In other words, the reference implementation builds on the domain-specific MDSD platform and exemplifies its use.
- *Transformation developers*. Transformation developers must be able to work efficiently with generation or transformation tools. They derive code templates, for example, for the generation of source code from the reference implementation, or define model-to-model transformations. Template development is also closely linked to the development of the domain-specific platform. Its goal is to tune transformation and platform development in such a way that neither the internal complexity of the platform nor the transformations' complexity gets out of hand.

Coordinators

Domain analysis and domain architecture teams each need a coordinator who is responsible for the result. Depending on the abilities and the size of the team, these two roles can be taken on by the same person. The role of domain analysis coordinator is compatible with that of language designer.

Other Roles

Besides those already mentioned, the following roles are relevant for domain architecture development:

- *Customers*. Customers should be involved in the development of GUIs and application prototypes as much as possible. To make sure that practical usability criteria are not ignored, potential end users should also be available. Customers can also be domain experts in the sense explained above.
- *Product managers*. For the development of products and product lines, it is not only important to consider the wishes of single customers, but also the strategic focus of the product or the product lines. The product manager should have a good idea of the product's use by existing customers. Similarly, they should know the requirements of further potential customers in the target market.
- *Project manager*. As in traditional software development, a project manager is required in MDSD. In this context experience in iterative software development is pivotal.
- *Requirements analyst*. Like in traditional software development, analysts are needed to analyze and define customer requirements in the form of use cases and similar artifacts.
- *Test engineers*. Test engineers are responsible for developing test strategies that take into account the special potential of MDSD in this context (see Chapter 14).

19.1.2 Application Development

The roles in MDSD application development are identical to those in traditional software development if you ignore the following aspects, because these are delegated to the domain architecture development thread in MDSD:

- The development of frameworks – that is, the development of the MDSD platform.
- The development of application prototypes and reference implementations as well as experiments with new implementation technologies.
- Analysis of requirements that are relevant for the entire target market.

It is obvious that precisely those critical activities that often cause delays or loss of quality are delegated to domain architecture development, hence they are better decoupled from the time pressure of day-to-day development (see Section 13.3).

In addition to knowledge about traditional software development, the application developer needs a basic understanding of MDSD in general and the DSL of the domain architecture in particular.

19.2 Team Structure

Nothing would be gained by trying to impose a universal team structure. After all, different teams develop their own working style, which is customized to meet the individual team members' knowledge and temperament. Morale can be damaged if one tries to superimpose rigid (new) roles on people. The assignment of tasks should take place based on available knowledge and individual preferences. We value a suitable macrostructure of the team much more, for example the separation of domain architecture development and application development.

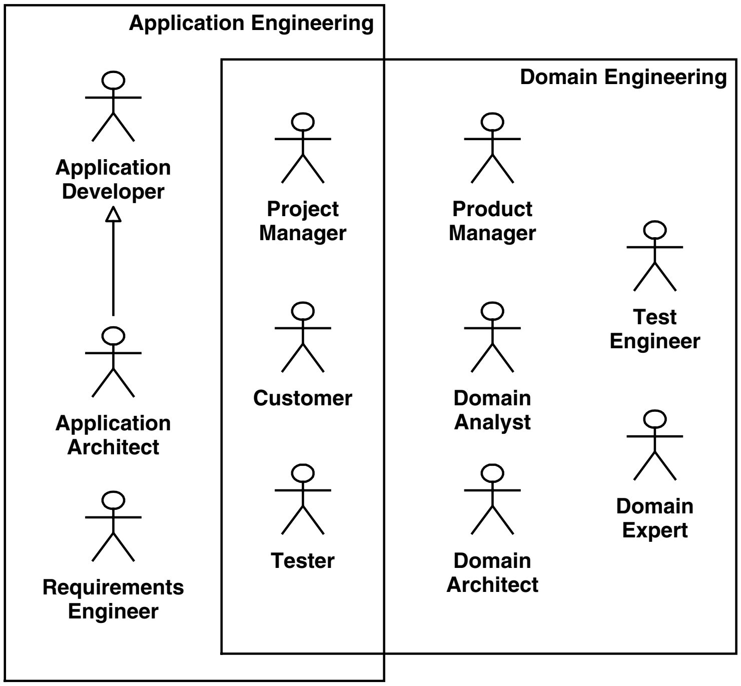


Figure 19.2 MDSD project team macrostructure

Figure 19.2 sums up the roles that are important in application development and in domain architecture development.

As you can see, there are roles that can be found in both development threads. If domain architecture development and application development take place in different physical locations, representatives of these roles must be present in both locations. The experts working in domain architecture development must convey their knowledge to application developers via workshops and training.

Moreover, the team structure should be aligned with the domain architecture's component structure, to support the scalability of domain architecture development. For example, if you have three important sections in the domain architecture, then you should probably have three groups of knowledge/experience in the team. A suitable distribution of responsibilities enables a top-down approach when outlining the domain architecture's design pattern. First, the interfaces are defined, so that subsequently the work on different subsystems and frameworks can be better parallelized.

19.2.1 Definition of Roles and Staffing Requirements

Filling actual roles depends on the project size, distribution of abilities in the team, and the focus of the MDSD domain. In architecture-centric MDSD, for example, the roles of domain analyst and domain architect can be taken on by the same people, because in this case the knowledge required for the MDSD domain is the domain architect's knowledge. In a very small, architecture-centric MDSD project, all the roles shown in Figure 19.1 can be filled by a single person, possibly even including the actual application development (thus a one-person project). In the context of a more comprehensive product family, the roles of domain analyst and domain architect particularly should be assumed by different people.

The scaling of the role assignment for domain architecture development compared to application development is of extreme importance: an MDSD project with, for example, more than a hundred application developers but only one transformation developer is bound to fail unless the domain architecture is already mature. Keep in mind that the domain architects have to coach the application developers, especially at the beginning of the project, and that feedback on the domain architecture is the consequence.

Of course we can't provide a simple rule of thumb, particularly since the bootstrapping phase of a domain architecture typically requires more staff than its iterative evolution. The number of application developers should increase proportionately, and only fan out after the bootstrapping phase. Figure 19.3 shows an idealized version of our experiences of role-oriented scaling over time.

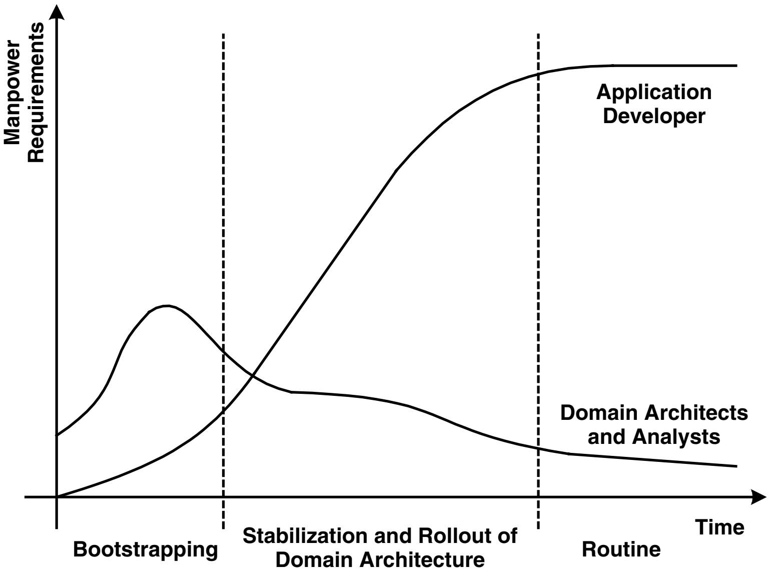


Figure 19.3 Temporal development of MDSD roles (idealized)

19.2.2 Cross-Cutting Teams

The same rules for effective communication between teams apply as for traditional software development on a larger scale: as soon as several developer teams work in parallel, compliance with standards and continuous improvement should be supported by establishing 'horizontal' groups to supplement the roles defined during the development process. As a guideline, we recommend that the following horizontal groups are created. (In this context, only the architecture group is mandatory. The other groups only become relevant if the teams consist of about fifteen or more.).

- The *architecture group* is responsible for the specifications of, and the priorities in, the domain architecture. This group typically consists of software architects from the application or product development teams and at least the coordinators of domain architecture development.
- The *quality assurance group* is responsible for the definition of and adherence to quality standards.
- The *product management group* is responsible for the quality of use cases and other requirements definitions, as well as for the priorities in product development. This group should include, for each product family: the product manager, domain experts, and the requirements analysts.
- The *project management group* is responsible for meeting deadlines and staying within the budget limits, for controlling, and for the definition of necessary project management standards.

19.2.3 Tasks of the Architecture Group

A cross-cutting architecture team is especially important if more than one application is being developed based on a single domain architecture. We cannot emphasize enough that the architecture group must see its role as that of a service provider and 'keeper of the software production line' to be successful. It is pivotal to avoid an 'ivory tower' architecture. The group's primary task is to control the evolution of the domain architecture(s) according to the project or product development requirements. The priorities are therefore not set by the domain architect, but by the architecture group. The domain architects must also make sure that their architecture will succeed in practice. It can be very useful to have them temporarily work as application developers or project architects.

The secondary task of the architecture group is to preserve the quality attributes of the domain architecture(s) – that is, to monitor performance, scalability, reusability, and maintainability. The architecture group must merge both objectives into a practicable synthesis.

We do *not* recommend that the architecture group be burdened with the operational development and evolution of domain architectures: the group usually won't have the time to do this. For this purpose, the organizational structure of a (partial) project with its own planning, iterations, and release cycles – similar to framework development – is much better suited. The architecture group should merely control the evolution of the domain architectures in terms of a cross-cutting representation of interests – that is, architectural features that are common to multiple products in the same domain.

This evolution typically has two dimensions – variant definition, and improvement:

- Variants emerge when a platform component is exchanged, such as for example the exchange of a data access layer from technology X to technology Y. If the domain architecture is well modularized, the transformations responsible for generating the data access layer will create a loosely-coupled module – often called a *cartridge* – within the domain architecture. You can now replace the X cartridge with a Y cartridge (which you may need to build first) without changing the interfaces to the business logic or the modeling language (DSL). What you get is a derivative of the domain architecture. To pursue this idea further, your main objective will be building blocks of cartridges that offer the maximum combinations while being minimally redundant. However, this goal will not be met by theorizing, but only through evolution.
- Improvements are brought about by feedback loops from the projects. For example, the correction of deficiencies such as too low an abstraction level in the DSL, or additional generative support for application fields that have been neglected or whose potential for automation has not been previously recognized. In most cases, improvements will result in new versions of existing cartridges. A transparent version and release management process should enable projects to decide for themselves whether they want to switch to a newer version or keep the old one (see Chapter 15 and Section 13.3).

In principle both dimensions of evolution (variant definition and improvement) can have negative effects on the concrete reuse of domain architectures: for example, one variant must be created first, or perhaps an improvement isn't backward-compatible. Yet it would be counterproductive to prevent evolution for this very reason, because it constitutes the ultimate source of innovation and added value. However, timely modularization on this level is important to the creation of building blocks for domain architectures. A production line for software systems based on modules is an investment that should be protected. These values, which extend well beyond the scope of individual projects, should be represented by the architecture group, and that group should control the evolution of domain architectures in the organization's best interest.

Our reflections are made, of course, from our perspective: they are certainly not relevant for your first MDSD project. Depending on the size of your organization, they are relevant for a planning interval of between two and five years. They are not hypothetical or purely theoretical, however, but are all based on actual experience.

19.3 Software Product Development Models

In the following sections we examine software development models from the perspective of a company that manufactures and distributes software products. We analyze the application field with a specific focus on MDSD. The same models can be used for the development of applications, but to keep things simple, we use product-oriented terminology.

19.3.1 Terminology

Several organizations can be involved in the development of a software product with respective customer-supplier relationships between the various parties – often referred to as *supply chains*. The term 'customer' is usually also used to describe the target market of the product. To avoid potential misunderstandings, we use the following terminology here:

- *Your company* is the software manufacturer who sells products to a certain marketplace.
- *Software service providers* are businesses whose services your company uses during product development. If necessary, we use the term 'software service provider' with a corresponding prefix, such as *on-shore*, *off-shore*, *external*, and so on.
- *Customers* are companies belonging to your products' target market.

In relation to *assets*, it is not enough to speak of 'software components«:

- *Software assets* are all value-added artifacts that play a role in software development: models, components, generators, languages, techniques, and process building blocks. Your organization can only develop one product, an entire product family, or a product line. A single product consists of a code base that is delivered to customers with a standard configuration. This configuration is either changed at installation time or later at the customer's site.
- A *product family* consists of a number of products that were developed on the same base – that is, they share at least one software asset, usually more. In the case of MDSD, this commonality is typically the domain architecture, so that the product family becomes a software system family (see Section 4.1.4). The elements of a product family are referred to as *family members*.
- A *product line* consists of a number of products with a common target market – that is, their arrangement is customer group-specific. An economically maintainable product line ideally consists of one product family. In practice, however, product structures often diverge from this ideal because software companies often grow through acquisition of other software manufacturers. The elements of a product line are *product line members*.
- *Product rationalization* is the consolidation and refactoring process in a product family.
- We speak of *software mass production* if product family members can be derived from customer specifications via a largely automated process and the efficiency of this procedure takes on the dimensions of mass production – this is the goal of MDSD in the context of functional domain-specific platforms.

19.3.2 In-house Development

The advantage of this model is the consolidation of domain knowledge and technical know-how – an ideal prerequisite for a good position in the market. Moreover, it can help to meet critical time-to-market requirements.

In-house development is without doubt ideally suited to the use of agile software development methods and the involvement of *on-site customers* in the development process. However, during the last few years this model has become less attractive due to economic pressure and competition from big manufacturers.

- *Use of MDSD*. With this development model, your organization can apply MDSD and benefit from the resulting cost advantages.
- *Risks*. The risk is high staff costs.
- *Applicability*. This approach can primarily be applied in scenarios in which the target market is local relative to your business, and where this proximity to the market can constitute a significant advantage for time-to-market.

This approach is not well-suited for a global target market, where proximity to the market cannot be achieved for all locations.

19.3.3 Classical Outsourcing

This model, in which software development is delegated to an external service provider, has been popular for a long time. It is attractive for businesses such as telecommunications service providers who need complex software to support their services.

It is interesting to examine what this development model, which is widespread in the production of custom-made software, looks like in the product development phase. It is an incentive for the external software service provider to develop a product (or a product family) that is universally applicable and thus provides a further source of income due to multiple sales. The external software service provider can either participate as early as during building of the prototype, or later in the product's evolution.

Typical service providers in this field are big multinational companies such as IBM, EDS, CSC, and others. These service providers market the development of products or product families under the label of *strategic outsourcing*.

- *Costs and advantages*. Your company can benefit from agreed cost limits for the development of certain software products, and the staff costs for your company's internal software development are reduced. Since the software service providers in this field are mostly big financially-strong companies, your company can use this model as a safeguard. Due to the physical proximity between your company and the software service provider, who is usually at least located in the same country, tried and trusted risk minimization strategies such as *on-site customer* [Bec00] can be applied without a problem.
- *Use of MDSD*. With this development model, the outsourcing service provider might benefit from MDSD.
- *Risks*. The software service provider often adopts staff (typically entire departments) from your company to obtain the necessary expertise. This means that a lot of knowledge leaves your company permanently. To counteract this risk, the outsourcing contract is usually limited to a period of five years or more.
- *Applicability*. If you use software yourself and at the same time wish to sell the software as a product, this model should only be considered if the software doesn't constitute the core of your business. You must take into consideration that your competition may buy and use your product.

19.3.4 Offshoring

In a world in which high software development costs are becoming a limiting factor to growth, offshoring approaches – the outsourcing of software development activities to low-cost locations – become increasingly attractive. Distributed software development becomes the rule. Those who have worked in software projects across time zones and cultural and language boundaries know that offshoring is not without risk. Even though one can save money for software development, time-to-market goals and a product that meets the market's requirements cannot always be guaranteed. To minimize the greatest risks, some offshore software service providers make offshore teams available with supportive resources at their customer's (onshore) location. This can help to overcome fundamental communication problems, but the efficiency of the whole product development process is nevertheless determined by the quality of the software development methodology that is used.

The software industry has learned the hard way that *big design up front* doesn't work – especially not if high quality is required and milestones are set. A team in one location that involves end users takes full advantages of the customer's presence on site.

So which risk reduction strategies are needed to compensate for a lack of communication with the customer in distributed software development? Your offshoring partner may have provided a variety of onshore resources on site to establish a connection with your product management, domain experts, and the offshore development center. To ensure the desired product quality, at least a few onshore team members must travel between the onshore and offshore locations. What effects does this therefore have on costs and time-to-market? We address these questions for MDSD in the sections that follow, and show that MDSD plays an important role in minimizing the risks in distributed software development.

19.3.5 Radical Offshoring

In *radical offshoring* software development is delegated as much as possible to low-cost locations. In the ideal case your company should own significant shares in the offshore software service provider, to ensure continuity of support.

This model is either implemented by establishing an offshore software company, or by searching for a suitable company that can perhaps even be acquired. The *start small* rule applies: first, one should conduct a small, non-critical project with an offshore service provider, followed by an evaluation of the project. Only then can a structure for future projects be established.

To successfully apply this model, a vast amount of expertise must be conveyed to the offshore software company.

- *Costs and advantages*. This approach requires an extensive exchange of staff and thus much traveling. Your domain experts and product manager must appoint the members of the offshore team and pass on the necessary knowledge. This model tries to maximize the advantages of low-cost locations, but these advantages must be weighed against high travel expenses and delayed time-to-market.
- *Use of MDSD*. If this development model is used, only your offshore partner will benefit from MDSD.

- *Risks*. The main risk of radical offshoring is the necessity to transfer considerable knowledge. Language barriers, particularly in the sector of industry-specific terminology, can be a further obstacle. Other than in normal outsourcing, radical offshoring does not offer the option of permanently relocating whole teams and their respective expertise to the offshore software company. The offshore team must be built from scratch, or it has to be assembled from an unknown pool of the offshore software company's resources.

  As soon as the necessary expertise has been conveyed, your business will depend heavily on the offshore software company. It can therefore make sense to secure ownership rights with regards to the offshore software company.

  Depending on where your target market is, time-to-market can become a problem. Furthermore, the implementation of risk minimization strategies like *on-site customer* [Bec00] can be difficult.

  The idea behind radical offshoring is the attempt to copy the simple structure of classical outsourcing. The problems caused by the physical distance between product management and software development can easily become a pitfall. All in all, this model is extremely risky.
- *Applicability.* This model can be applied if time-to-market is not critical, and particularly if your company already owns an experienced offshore software company. Nevertheless, the lack of technical expertise in the parent company – which might increase over time – means that the requirement for travel between locations will remain during the entire product lifecycle.

19.3.6 Controlled Offshoring

This approach means that only a part of the development process is outsourced to low-cost locations. We speak of *controlled* offshoring, not because other variants of offshoring are uncontrollabe, but because in controlled offshoring considerable domain expertise, technical expert knowledge and control regarding the company's intellectual property stay mostly in your business. This model offers the option of involving offshore software service providers without risking a monopoly-like dependency.

- *Costs and advantages*. In this model your company cooperates closely with customers to specify products, to develop product prototypes, reference implementations, and to design software architectures. This model requires technically well-versed onshore resources that should be available anyway if you have practiced in-house software development until now. At first glance, it may seem that this will lead to high costs, yet experience shows that the theoretical financial advantages of offshore prototype development are foiled by failure to communicate efficiently. The physical proximity of product management, customers, and important members of the software development team is maintained, which reduces the total risk compared to radical offshoring. If the relevant methods and tools are used, this model lowers the risk of critical dependency on a specific offshore software service provider.
- *Use of MDSD.* From your company's perspective, the advantages of MDSD can be fully leveraged when this offshore variant is used: since the domain architecture development – including domain analysis and language design – stays within your organization, MDSD

offers not only formally-defined (and hence comprehensible) software requirements in the form of models, but also a complete architectural framework for the product which the offshore software service provider can use. First, this guarantees that important architectural standards are adhered to. Second, the formal specifications allow use of a test-driven approach (see Chapters 13 and 14). MDSD allows a high degree of control regarding the product's technical quality, thus reducing both the offshoring risks and the need to travel. MDSD is thus sufficient – and probably even necessary– to realize controlled offshoring.

- *Risks*. Compared to radical offshoring, this approach leads to higher onshore costs – at least the ratio of onshore compared to offshore costs is better. The total costs can actually be lowered by using MDSD. As soon as a business has decided which type of offshoring strategy it wants to use, it runs the danger of overestimating the advantages of offshoring. The remaining manual development work should be reassessed to ensure that offshoring is indeed the best option, particularly if a high level of automation can be reached via MDSD.
- *Applicability.* This approach is particularly well-suited if your company is located in the same region as your target market. The staff savings when applying MDSD are especially positive if this means that the offshore team's size can be reduced, allowing the project management overhead to becomes less. The MDSD approach is easily scalable and suitable for the development of entire product families and product lines. However, domain architecture development should not be outsourced – this would mark the transition to radical offshoring, with an correspondingly high risk.

19.3.7 Component-wise Decision

Products or product families with a good design consist of relatively loosely-coupled components. The development of a new product, or the streamlining of existing products (see Section 19.3), are excellent opportunities for realizing a good component architecture [Bet04b].

Why is a clear-cut component architecture so important? First, such an architecture allows the application of a *divide and conquer* technique to break down the work on large systems into tangible and parallelized parts. Second, it supports strategic *build/buy/open source* decisions at the component level.

When designing a product family, it is important to decide on the software development model (in-house, classical outsourcing, radical offshoring, controlled offshoring) on a per-component basis, instead of globally. The same should be observed for the relevant supply chains.

In fact, most of today's software products aren't structurally designed particularly cleanly. They consist of closely-coupled components that are difficult to disentangle. The consequence is that most products are closely bound to a specific combination of implementation technologies. Changes to one of these technologies can result in a considerable maintenance effort. The concept of replaceable components therefore is fiction in the majority of today's products. The implementation of a clear-cut component architecture becomes appealing only when a business has recognized the full potential of the product family concept.

The sources [Bos00], [Coc01] and [Bet04b] contain further material regarding organizational topics in relation to MDSD.

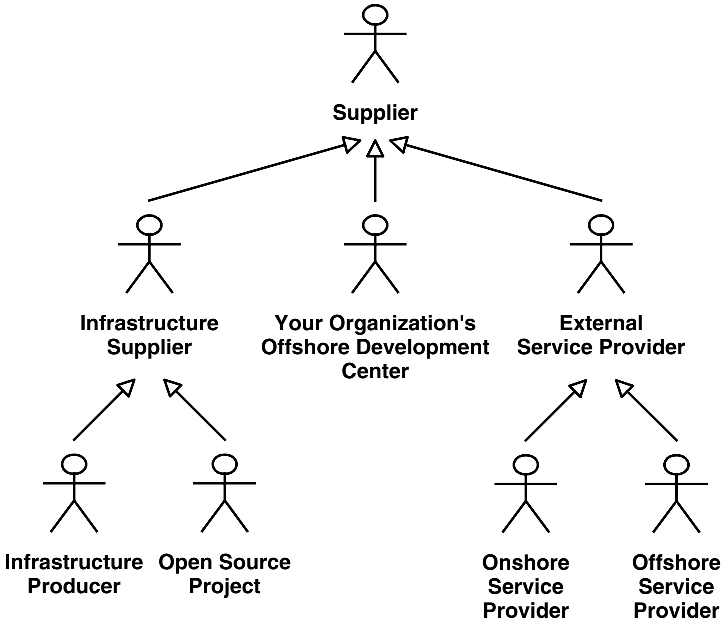

Supplier
Infrastructure Supplier
Your Organization's Offshore Development Center
External Service Provider
Infrastructure Producer
Open Source Project
Onshore Service Provider
Offshore Service Provider

Figure 19.4 Potential software suppliers

20 Adoption Strategies for MDSD

with Jorn Bettin

We hope we have made it clear the potential that Model-Driven Software Development has and which investments or organizational measures are useful or required in principle. We also hope that at this point you are wondering how and under what circumstances you can take steps towards MDSD in your own project or business, and how your adoption of MDSD might look. This chapter tried to answers these questions.

20.1 Prerequisites

Model-Driven Software Development in the context of this book can certainly be considered quite advanced in relation to software engineering. You may have to face much more profound changes that must be addressed before optimization in the form of MDSD will become useful:

- Has an iterative-incremental approach been established for the execution of the project – particularly in the field of requirements management? The required separation of domain architecture development and application development on the introduction of MDSD requires a functioning coordination of two partial projects: if you separate domain architecture development from application development, as we recommend, you need to coordinate these two aspects of development.
- Does the team have practical and useful architectural knowledge?

If the answer to either of these questions is ‘no«, we recommend that you address these issues prior to defining an MDSD pilot project, or at least while defining it – with external support if necessary.This should allow you to reduce avoidable friction, risks, and costs in project execution.

20.2 Getting Started – MDSD Piloting

An initial step in the direction of MDSD often consists of ‘proof of concept’ for the technical aspects of MDSD in the relevant context – for example the choice of tools (see Chapter 11). If this is successful, a pilot project can follow.

When we are asked to advise on the choice of a suitable pilot project, we are often confronted with statements such as “Project X would be well-suited, but our schedule is so tight that we don’t want to risk trying something new.” We can only respond that almost all projects run on a tight schedule and that MDSD is actually meant to alleviate exactly this situation. In other words, precisely these projects are ideal candidates, at least if the following conditions are met, or can be met:

- Ideally, the project is attractive enough to serve as a ‘seed’ for the use of MDSD.
- The project has sufficient versatility, duration, and enough schematic aspects to make automation worthwhile.
- The *start small* principle can be applied. Given the right conditions, even a technically complex and risky project can be tackled, as long as no more than about ten people overall participate in its development. The project shouldn’t be a one-man project, of course. A manageable team offers the best basis for successful knowledge transfer.
- MDSD must be accepted by the team, at least as an experiment. It is normal for a project team to require some convincing about its introduction. Developers especially must realize that they won’t lose anything by applying the method, but instead win something by getting rid of a lot of tedious copy and paste programming.
- Sufficient MDSD skill is available in the team, if necessary supplemented by external coaching.
- The software development environment should not be counterproductive for MDSD. This includes closed CASE tools, (UML) tools that require round-trip engineering (see Chapter 5), closed development environments with a source code repository of their own without an interface, as well as generative IDEs with proprietary models that can only be created interactively and thus cannot be generated via an MDSD transformation. Ideally, you should select suitable MDSD tools beforehand.

20.2.1 Risk Analysis

The additional risk that MDSD adds to a pilot project is small, because at any given time it is possible to give up formal modeling and code generation if you find that for any reason that it does not work for the project. The work that will have been invested in the architecture and systematics at that point will not have been in vain and is a benefit for the project.

As the MDA has not established a standardized transformation language at the time of writing, an MDSD project must use a proprietary approach/tool to create its transformations/generators. (There is QVT, but it is not yet generally adopted.) However, the code volume of these metaprograms is extremely small compared to the volume of generated code – see the evaluation in Chapter 18. Whether a tool is suboptimal or fundamentally unsuitable for MDSD will be realized relatively early in a pilot project, so that a possible migration of the transformation to another tool should not constitute a significant overhead.

You can achieve investment protection and independence of manufacturer if you use Open Source tools that have reached a high degree of maturity (see Chapter 11).

20.2.2 Project Initialization

The initialization of a pilot project should include the following:

- The team must familiarize itself with the basics of MDSD. This especially includes the process aspects described in Chapter 13, such as the separation of application and domain architecture development, as well as two-track iterative synchronization (Section 13.3). If necessary, get an MDSD coach.
- Start out with architecture-centric MDSD and let your team gain some experience with this MDSD variant first.
- The team should take on the roles described in Chapter 19 in a way suitable to the project's size. Pay special attention to dynamic balancing of staffing, as explained in Section 19.2.1 – plan the bootstrapping phase of the domain architecture and the accompanying work load for the respective roles. The team's technology experts should be able to make an assessment based on the concrete facts.

If a test run of the domain architecture's bootstrapping is not feasible, introduction can take place via a separate partial project. In this context, we can recommend the following for enterprise software development:

- Establish a small but powerful partial team for the creation of a reference implementation.
- At the same time, start to develop the actual application with a strong focus on the GUI, with the goal of providing your customer with a reasonable prototype (a first functional increment) to give a first impression of the actual application. Implement the required minimum of business logic first, and encapsulate it as well as possible in separate classes/modules. Replace the still-unavailable architectural infrastructure, such as database integration, with simple mock objects (see Chapter 14).
- The reference implementation should also cover architectural aspects such as integration with existing legacy applications (see Section 8.2.5).
- A continuous exchange of information in both directions between the reference and the application teams must be guaranteed: the reference implementation team must know the requirements and ideas of application development and respond to them, whereas the application developers must know the concepts and ideas embodied in the reference implementation.
- Synchronize application development as soon as an initial version of the architectural reference implementation is available. From this point onward, the construction and programming paradigms of the reference implementation are integrated into application development. Depending on the schedule and available capacities, a small subteam can apply refactorings to existing application parts, replacing the mock objects with the reference implementation's concepts. The feedback loop between the application development team and the smaller reference implementation team is still pivotal.
- At this point, the domain architects can begin to extract the formal aspects and start to build the domain architecture.

- As soon as an initial release of the domain architecture is available, application development is switched to MDSD and existing application parts – if applicable – are remodeled so that they are also ready for the transition.

20.3 MDSD Adaptation of Existing Systems

Often one not only wants to develop new systems using MDSD, but also to apply the advantages of MDSD to existing systems, maybe even modernize them by outfitting them with a new technological basis.

The procedures and best practices described in this book are also valid for this type of project, yet with a different focus and direction for specific steps during bootstrapping of the domain architecture:

- The existing application largely assumes the role of the prototype and the reference implementation. In the case of a modernization, the reference implementation shows the new target architecture.
- It is possible that the (MDSD) platform is not clearly recognizable or separated from the application. If necessary, a refactoring must take place.
- Additional (infrastructure) code may be created in the reference implementation to provide the legacy application generatively with standard interfaces or adapters later. It may be the case that *only* such adapters need to be supplemented generatively, while at the core the legacy application is not switched to MDSD.
- In most cases the legacy applications will not possess the clear structures that are necessary for a reference implementation, for example the derivation of transformation rules in the form of generation templates. It is necessary to analyze which building principles are valid and which are not. If necessary, some parts of the application must be altered into an ideal state via refactoring, as examples.
- Domain analysis and DSL construction must be adapted to match the valid structures of the legacy application or the newly introduced infrastructure code.
- To create a reference design/model, the legacy application or a part of it is remodeled using the DSL. It is then used to verify code generation.
- The partial aspects of the legacy application are remodeled via the DSL in the application development thread, followed by generation of the new implementation skeleton. If applicable, remaining code from the legacy application is integrated into this framework. In the case of a modernization, it is migrated.

As can be clearly seen from this list, severe restructuring efforts *might* be required, depending on how deeply the MDSD approach impacts the legacy application. The actual situation, and particularly the cost/benefit ratio, can only be determined for each scenario via careful analysis. Likewise, when legacy applications are adapted, the right balance between DSL, platform, generated code, and non-generated code is essential.

We need to warn explicitly against the idea that simple reverse engineering of a legacy application, although it without doubt also results in a model, could be an alternative to MDSD: you would get a partial code visualization of your former application, nothing more. The advantages of Model-Driven Software Development cannot be obtained in this way (see Section 18.1).

20.4 Classification of the Software Inventory

As soon as the first MDSD projects have been completed successfully, the sphere of MDSD can be extended from a single project to a group of projects/product developments. MDSD can thus become part of a company's IT strategy. This will lead to the concepts of *product-line engineering* (Section 13.5), *software system families* (Section 4.1.4), *product families*, and *product lines* (Section 19.3.1).

To adapt these concepts, it makes sense to classify the software inventory for their usability and the maintenance costs for each of the constituent parts. This step can be taken prior, during, or after MDSD piloting, but it should be considered a prerequisite for business-wide MDSD use.

If priorities in software maintenance and evolution are only determined by short-term goals, the intention of the original design can get lost over time. Apart from the project's actual time-pressure, an exclusive view of software from an accounting perspective favors neglect of software maintenance. This view can hardly be reconciled with the idea of building and maintaining a domain architecture incrementally, or with that of the targeted reuse of software assets within a product family.

Due to the application of domain-specific knowledge and a disciplined approach to the development of strategic software assets (models, components, frameworks, generators, languages, and building blocks) from parts of existing software, *the software's value increases*, contrary to conventional software maintenance. The value of strategic software assets is best used if a long-term investment strategy is pursued. This is necessary to avoid fatal compromises over the software architecture's quality. From this perspective, too, a conscious separation of domain architecture development and application development is advantageous: in application development, short-term, tactical decisions can help to reach time-to-market goals. In the context of domain architecture development, one can work on more general, longer-term solutions in parallel. These solutions will then be available from a specific release onwards. The MDSD approach offers significant advantages here, because later changes to the architecture can be integrated much more easily into the entire code base.

It doesn't make sense to consider all software components used in an application as strategic software assets of course. For investment planning in software, the following classification scheme is well-suited:

- *Strategic software assets* constitute the heart of your business. These are assets that are further developed to become both a human and machine readable knowledge base of your products and business processes.
- *Non-strategic software assets* are necessary infrastructures that are affected by changes in implementation technologies and which should take two to three years to depreciate.
- *Legacies* are software components or systems with maintenance costs too high to be further maintained.

The identification of strategic software assets is only possible if a clear business strategy exists, if profitable business processes are known within the company, and the company is able to articulate the software requirements. The strategic software assets of a company define its competitive edge. The functionality of non-strategic software assets is not unique and can be acquired from a number of manufacturers. Examples include operating systems, relational databases, and

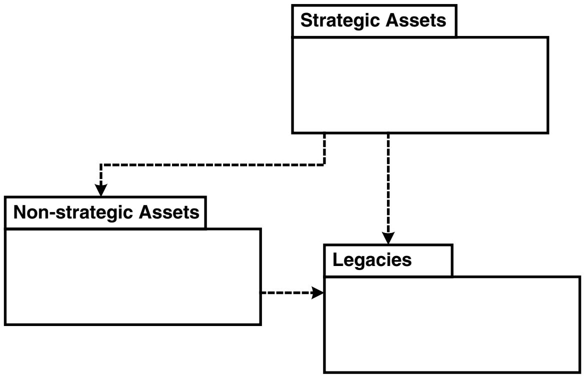


Figure 20.1 Classification of a software inventory

application servers. Strategic software assets typically build on non-strategic (infrastructure) assets. The latter must be identified and treated as such.

Model-Driven Software Development provides tools for the management of strategic software assets. Model-driven integration allows the use of commercial third-party tools to provide non-strategic software assets economically. The use of Open Source infrastructure as a public asset reduces the cost of creation and maintenance of strategic software assets.

20.5 Build, Buy, or Open Source

Software manufacturers are notorious for suffering from the *not-invented-here* syndrome. The choice of whether to build, buy, or use Open Source software should be based on economic long-term considerations:

- Don't just compare the short-term costs, but also the total cost of ownership, including maintenance and cost of capital.
- Will the development of an infrastructure pay if you consider the maintenance costs? Will you gain a competitive advantage – and if so, for how long?
- As soon as an infrastructure developed in-house is superseded by industry standards, the consequences of further maintenance should be compared to those of adopting the standard.

Nevertheless, you should not replace hard-earned, domain-specific platforms or MDSD transformations – which are strategic software assets – by less well thought-out standard tools without first pondering the consequences.

Commercial standard software from third-party manufacturers can largely be considered a non-strategic software asset. Commercial standard software typically is only partially based on open standards, and you cannot assume that the software product will be supported over an extended period of time. If you invest in third-party products, these should be seen as depreciating assets.

Be honest and don't declare all your legacy components to be strategic software assets. Identify your legacies and plan their replacement with a combination of strategic and non-strategic software assets. If you don't pay enough attention to the quality of strategic software assets, they will soon become legacies. You must therefore observe the Extreme Programming rule *refactor mercilessly* in regard to your domain architecture.

A sound mixture of strategic and non-strategic software assets might contain only a small core of strategic assets. The smaller this core is, the easier it is to evolve with the necessary care. Non-strategic software assets play an important role and can be seen as necessary ‘consumer goods«.

Proven standard software components and robust Open Source infrastructure components that are used by thousands of organizations should not be disregarded in software development. Solid Open Source software can even be considered a strategic (public) software asset. If applicable, use an Open Source infrastructure to avoid the risks that are inherent in dependencies on manufacturers. But do take a close look at the licensing model when evaluating Open Source software. Some licenses allow unproblematic use in terms of commercial product development, whereas other licenses only allow use if the end product is also subject to the same Open Source license.

20.6 The Design of a Supply Chain

When you have classified your software and know which software assets are strategic or non-strategic, you can start to design a supply chain for future product families.

To this end, we have applied the classification listed above – strategic, non-strategic, legacy – to the most important MDSD artifacts (Section 4.1.4):

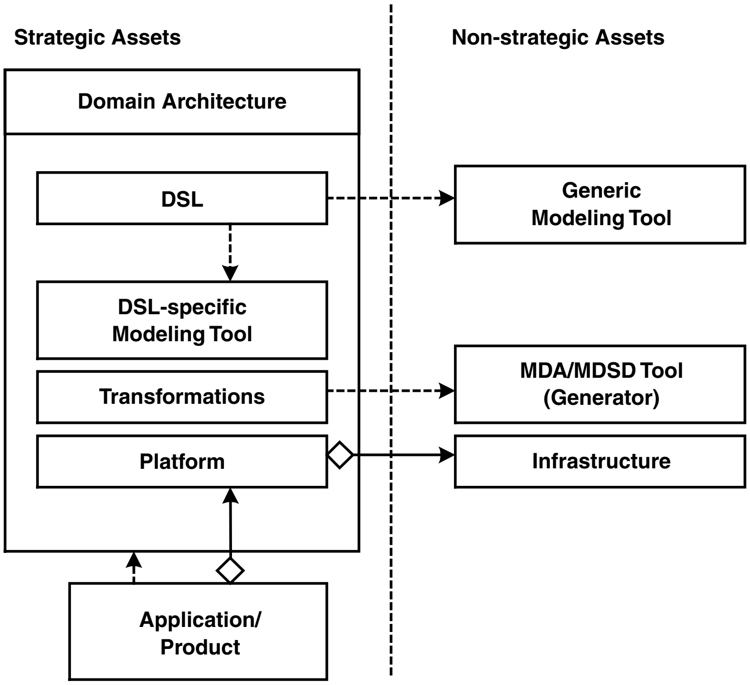


Figure 20.2 Classification of MDSD artifacts for supply chains

The application receives its structure from the domain architecture. The application encompasses the MDSD platform, and this again includes the infrastructure. The term 'infrastructure' summarizes all the non-strategic assets that have been subsumed during classification of the software inventory. Thus we get a very pragmatic definition of the boundary between platform and infrastructure.

Generic modeling tools such as UML tools are not strategic, while specific editors that were created and optimized for a concrete DSL are considered to be strategic – they hold relevant domain knowledge. For non-strategic software assets, the elaborations made in Section 20.5 are valid: for example, Open Source tools, frameworks or commercial products can be used here.

As a rule, applications and software products are strategic, of course. We can recommend in-house development, or possibly controlled offshoring, as explained in Chapter 19. In this case, it is important to design the software development process in such a way that the cost of a potential switch between offshore software service providers is not exorbitant. In this respect MDSD offers considerable advantages, because it decouples an application/product from the domain architecture. This is why development of the domain architecture, and hence of the platform, should not be outsourced (see Section 19.3.6).

It's important to emphasize that offshoring should take place only *after* successful introduction of MDSD. This guarantees that at the point of offshoring a well-defined development process and a defined software architecture exist that are supported by the according tools.

20.7 Incremental Evolution of Domain Architectures

As soon as some experience with architecture-centric MDSD has been obtained with an initial application, a functional/professional MDSD platform can be built incrementally (Section 18.6). Once in place, highly automated product families or product lines can be created based on the platform.

Domain architecture development is a continuous process that takes place during a product family's entire lifecycle. The distribution of costs between domain architecture development and application development depends on the platform's maturity and on the product family's versatility (see Section 18.6).

20.8 Risk Management

The adoption of a new method or technique should also be considered in the light of risk management of the projects involved. In this section we address typical risks that can endanger successful use of MDSD or product developments. For each risk we describe the potential damage, as well as mitigation strategies.

20.8.1 Risk: Tool-centeredness

Description

The management or the project team focus their attention on a specific tool (EAI, IDE, or MDA) in hopes that it can solve even the hardest problems. Both the software architecture and development process are neglected.

Damage

This procedure causes the actual requirements to be neglected as well. Here are a few examples: a portal is decided on, but the actual requirement is 'process integration«, or a CASE tool is purchased whose code generation does not meet the software architecture's requirements. The consequence is either that the tool is forced into the development, or that an unsustainable software architecture or development process is created. In both cases, extremely high consequential costs must often be faced. The project may even fail.

Mitigation

First define an outline of the software architecture and the development process, followed by a tool projection – that is, select the suitable tools to cover single aspects. In the context of MDSD, the reference implementation can be used for validation.

20.8.2 Risk: A Development Tool Chain Counterproductive to MDSD

Description

This point has already been discussed in Section 20.2: closed, integrated development environments can hamper the use of generators outside this environment considerably. Another stumbling block is created by modeling tools that do not support multiuser modeling, and thus impede or prevent model-driven teamwork.

Damage

Either much more effort is needed to establish a working development and build process, or the additional effort is delegated to application development and multiplied in this manner.

Mitigation

Select the tool in the MDSD tool chain according to your requirements (see Chapter 11).

20.8.3 Risk: An Overburdened Domain Architecture Team

Description

The role-related staff requirements or the necessary preparation time for the domain architecture's bootstrapping are not sufficiently considered, or the ratio of domain architects to application developers is critical.

Damage

The application development is thwarted by missing features or late deliveries. In the worst case, this can result in failure.

Mitigation

Observe the role dynamics for staff requirements described in Section 19.2.1.

20.8.4 Risk: Waterfall Process Model, Database-centered Development

Description

At the start of the project, approved and 'ready' IT concepts, GUI designs, or database designs are present that are seen as written in stone, while they are actually incomplete and maybe even inconsistent. Database development takes place outside the MDSD project.

Damage

In general the project-related risks increase compared to an iterative-incremental process. In an MDSD project, two-track, iterative development (see Section 13.3) is required. In the case of separate database development, the potential of the holistic, generative MDSD approach may not be optimally realized.

Mitigation

Establish an incremental-iterative development process.

20.8.5 Risk: The Ivory Tower

Description

The domain architectures work without sufficient project coupling and feedback loops.

Damage

Frameworks are created that either don't meet the actual requirements of the project, or that are not usable in real-world scenarios. They often contain superfluous 'technology gadgets' instead. The application developers must compensate for this, which means that they enter the architects' sphere, so that disputes and mutual accusations are inevitable. The economic damage to the project can be severe.

Mitigation

Establish frequent feedback loops and make sure that communication channels are as direct as possible. Foster understanding between team members. If applicable, change the assignment of roles between application developers and domain architects.

20.8.6 Risk: No Separation of Application and Domain Architecture

Description

The MDSD roles (Section 19.1) don't necessarily imply a staff-wise separation of application development and domain architecture development. However, the separation of activities and artifacts is essential (Chapter 13). If this is not done, application and domain architecture will merge.

Damage

The domain architecture is not at all – or only poorly – reusable in further projects. Evolution might still take place in the projects, but differently and with many derivatives. This leads to a strong demand for refactoring, or the idea of the software system family (Section 4.1.4) is discarded altogether. In the first case, more effort and time will be required: in the second, the potential of MDSD is not fully utilized.

Mitigation

Make sure that the domain architecture is decoupled from the applications, particularly at the packaging and repository level. No dependency of the domain architecture artifacts' on an application can be allowed occur: if necessary, refactor. Regardless of the assignment of roles, the team must realize that the domain architecture is an independent (sub)project (Section 13.3).

A Model Transformation Code

A.1 Complete QVT Relations alma2db Example

```
transformation alma2db(alma : AlmaMM, db : DbMM) {

  /* keys are required to avoid duplication in the target
     repository on iterative runs. */

  /* tables are solely identified by their names. */
  key Table {name};
  /* for a column, its name as well as its container, i.e. a table,
     uniquely identify a column. */
  key Column {table, name};
  /* a key is uniquely determined by its owner, which is the table.*/
  key Key {table}
  /* a foreign key is identified by its owner (the table), as well as
     the primary key it is referring to. */
  key ForeignKey{table, key}

  /*****************************/
  /* Mapping entities to tables */
  /*****************************/

  /* Each entity has a corresponding db table with the same name */
  top relation EntityToTable {
    prefix, eName : String;

    checkonly domain alma entity:Entity {
        name = eName
    };
    enforce domain db table:Table {
        name = eName
    };
    where {
       /* For each entity and associated table, we calculate
          all columns that correspond to the different fields of
          the entity. Since we flatten out all nested fields, we
          need to make sure all created columns get a unique name.
          This is achieved by the prefix. */
       prefix = '';
       RecordToColumns(entity, table, prefix);
    }
  }
```

```
/* For each field of a record, several columns may be created
   depending on the type. */
relation RecordToColumns {
  checkonly domain alma record:Record {
     fields = field:Field {}
  };
  enforce domain db table:Table {};
  primitive domain prefix:String;
  where {
    /* we have an explicit relation between each field and its
       associated columns. This is required to construct the key
       of the table as is shown below. */
    FieldToColumns(field, table);
  }
}

/* for each field, several columns may be created depending
   on the type. */
relation FieldToColumns {
  checkonly domain alma field:Field {};
  enforce domain db table:Table{};
  primitive domain prefix:String;
  where {
    /* Because we have three types of fields, we have to
       split up the expansion in columns accordingly. */
    PrimitiveFieldToColumn(field, table, prefix);
    PhysicalQuantityTypeToColumn(field, table, prefix);
    valueTypeToColumn(field, table, prefix);
  }
}

/* Primitive fields immediately lead to one column of
   primitive type. */
relation PrimitiveFieldToColumn {
  fieldName, almaPrimitiveType : String;

  checkonly domain alma field:Field {
    name = fieldName,
    type = pt:PrimitiveType {
      name = almaPrimitiveType
    }
  };
  enforce domain db table:Table {
    key = tableKey:Key{}
    columns = column:Column {
      name = prefix + fieldName,
      type = AlmaTypeToDbType(almaPrimitiveType)
    }
  };
  primitive domain prefix:String;
}

/* A PhysicalQuantityType is very much like a primitive type,
   except that it is 'annotated' with its unit. To accommodate
   this, we simply create a column for each possibly unit. */
relation PhysicalQuantityTypeToColumn {
  pqName, pqUnit, fieldName : String;
  checkonly domain alma field:Field {
    name = fieldName,
    type = pq:PhysicalQuantityType {
      name = pqName,
      /* Note that since each unit is taken separately and leads
         to one column in the db domain. */
      units = pqUnit
    }
  };
  enforce domain db table:Table {
    columns = column:Column {
      name = prefix + fieldName + '_as_' +
             pqName + '_in_' + pqUnit,
      type = AlmaPhysicalQuantityTypeToDbType(pq)
    }
  };
  primitive domain prefix:String;
}

/* value types are simple aggregations and are recursively
   mapped via its containing parts. To do this, it recursively
   calls FieldToColumns with a new extended prefix for column
   names. */
relation ValueTypeToColumn {
  fieldName, newPrefix : String;

  checkonly domain alma field:Field {
    name = fieldName,
    type = valueType:ValueType {
      fields = field:Field {}
    }
  };
  enforce domain db table:Table{};
  primitive domain prefix:String;
  where {
    newPrefix = prefix + fieldName + '_';
    FieldToColumns(field, table, newPrefix);
  }
}

/**************************************/
/* Mapping Entity Keys to Table Keys */
/**************************************/

/* For each entity key, we construct a table key */
top relation EntityKeyToTableKey {

  checkonly domain alma entity:Entity {
    key = entityKeyField:Field {}
  };
  enforce domain db table:Table {
    key = tableKey:Key {}
  };

  when {
    /* This relation only makes sense whenever we have a relation
       between an entity and its table. */
    EntityToTable(entity, table);
  }
  where {
    /* Go and look up the columns constructed for a key. */
    KeyRecordToKeyColumns(entityKeyField, table);
  }
}

/* first generalize the problem for records since we have to
   recursively go down */
relation KeyRecordToKeyColumns {

  checkonly domain alma record:Record {
     fields = field:Field {}
  };
  enforce domain db table:Table {};

  where { KeyFieldToKeyColumn(field, table); }
}

relation KeyFieldToKeyColumn {

  checkonly domain alma field:Field {};
  enforce domain db table:Table {};

  where {
    KeyPrimitiveFieldToKeyColumn(field, table);
    KeyPhysicalQuantityTypeToKeyColumn(field, table);
    KeyValueTypeToKeyColumn(field, table);
  }
}

relation KeyPrimitiveFieldToKeyColumn {

  checkonly domain alma field:Field {
    type = pt:PrimitiveType {}
  };
  enforce domain db table:Table {
    key = tableKey:Key {columns = column:Column {}}
  };

  when {
    /* notice that this functions as a lookup */
    PrimitiveFieldToColumn(field, table:Table{columns =
                                              column:Column {}});
  }
}

relation KeyPhysicalQuantityTypeToKeyColumn {

  checkonly domain alma field:Field {
    type = pq:PhysicalQuantityType {}
  };
  enforce domain db table:Table {
    key = tableKey:Key {columns = column:Column {}}
  };
  when {
    /* analogous */
    PhysicalQuantityTypeToColumn(field,
                                 table:Table{columns =
                                             column:Column {}});
  }
}

relation KeyValueTypeToKeyColumn {

  checkonly domain alma field:Field {
    type = valueType:ValueType {}
  };
  enforce domain db table:Table{};

  where { KeyRecordToKeyColumns(valueType, table); }
}

/***************************************************************/
/* We map each dependent part of an entity to its own table    */
/* with a connection to the table of the entity. The remaining */
/* nested dependent parts are expanded into the table for the  */
/* top-level dependent part.                                   */
/***************************************************************/

top relation DependentPartOfEntityToTable {
  eName, dpName, prefix, dpTableName : String;

  checkonly domain alma entity:Entity {
    name = eName,
    parts = dp : DependentPart {
      name = dpName
    }
  };
  enforce domain db entityTable:Table {
    /* we first construct the foreign key ... */
    foreignKeys = foreignKey:ForeignKey {
      /* ... which in its turn constructs a key ...*/
      key = dpKey:Key {
        /* ... with one column for the index ... */
        columns = keyColumn:Column {
          name = 'key_' + dpTableName,
          type = 'INTEGER'
        }
        /* ... and a table for the dependent part. */
        table = db dpTable:Table {
          name = dpTableName,
          /* The table for this dependent part is set up
             with a key column. The other columns are
             calculated below. */
          columns = keyColumn:Column {}
        }
      }
    }
  };
  when {
    EntityToTable(entity, entityTable);
  }
```

```
    where {
      /* we set the name for dependent part's table, which
         is composed of the entity name and the dependent part
         name. */
      dpTableName = eName + dpName;
      /* we calculate all columns for a dependent part and add
         them to the newly created table. */
      RecordToColumns(dp, dpTable, '');
      /* we construct also columns for each nested dependent part.
         To avoid ambiguity, we manage a prefix based on the
         dp's name. */
      prefix = dp.name + '_';
      DependentPartOfDependentpartToColumns(dp, dpTable, prefix);
    }
  }

  /* This relation is a recursion and only applies if a dependent
     part has more dependent parts on its own. */
  relation DependentPartOfDependentpartToColumns {
    dpName : String;

    checkonly domain alma dp:DependentPart {
      name = dpName,
      parts = depSubpart:DependentPart{}
    };
    enforce domain db dpTable:Table {};
    primitive domain prefix:String;
    where {
      RecordToColumns(depSubPart, dpTable, prefix);
      newPrefix = prefix + dpName + '_';
      DependentPartOfDependentpartToColumns(depSubPart,
                                            dpTable, newPrefix);
    }
  }

  /***********************************************************/
  /* The following two functions deal with the conversion of  */
  /* primitive types                                          */
  /***********************************************************/

  function AlmaPhysicalQuantityTypeToDbType
           (pq : PhysicalQuantityType) : String {
    if (pq.oclIsTypeOf(IntPQType)) then 'INTEGER'
    else if (pq.oclIsTypeOf(FloatPQType)) then 'REAL'
    else if (pq.oclIsTypeOf(LongPQType)) then 'BIGINT'
    else 'DOUBLE'
  }

  function AlmaTypeToDbType(almaType : String) : String {
    if (almaType = 'int') then 'INTEGER'
    else if (almaType = 'float') then 'REAL'
    else if (almaType = 'long') then 'BIGINT'
    else 'DOUBLE'
  }
}
```

A.2 Complete QVT Operational Mappings alma2db Example

```
transformation alma2db(in alma:AlmaMM,out db:DbMM);

-- auxiliary property to register whether a column
-- originates from a 'key' field
intermediate property Column::inKey : Boolean;

-- entry point of the transformation -------------------

main() {
  -- we start the transformation by transforming all entities
  -- to tables where we start with the empty prefix for table
  -- and column names
  alma.objectsOfType(Entity)->map entity2table('');
}

-- Mapping operations ---------------------

-- Abstract mapping used for records, which simply populates
-- a table with columns by flattening the fields of the record. This
-- rule is invoked implicitly through inheritance declarations
abstract mapping Record::fieldColumns(in prefix : String) : Table {
  init {
    -- Calculate the columns for the leaf fields of this record.
    -- Observe that the predicate self.key = f exploits that if
    -- self does not have the property key, it is null and
    -- thus the predicate would return false
    var mainColumns := self.fields->map(f)
                            field2Columns(prefix, self.key = f);

    var recordName := self.name;
  }
}

-- An alma entity maps onto a Table
mapping Entity::entity2table(in prefix : String) : Table
inherits fieldColumns {
  -- inheritance actually means an invocation of
  -- self.initializeTable(result, prefix) where 'result' is a Table
  -- instance created before the call (unless the table was passed
  -- to the mapping in the mapping invocation itself)

  name := recordName;

  -- We build the primary key for the table
  key := object Key {
           -- Note that Column::inKey is temporary data,
           -- which simply reports if the column takes part in a key.
           -- The part [inKey=true] is a filter on columns
           column := result.columns[inKey=true];
         };

  -- We actively construct the tables for top-level dependent parts
  var dependentTables := parts->map
                            part2table(prefix + recordName + '_');

  -- we construct the foreign key columns by fetching the column
  -- of each constructed table for the dependent part, which was
  -- designated to be a primary key
  var foreignKeyColumns := dependentTables->columns[inKey=true];

  -- the actual columns are the ones originating from fields as
  -- well as the foreign key columns
  columns := mainColumns + foreignKeyColumns;

  -- finally, we construct the foreign key objects as well, one
  -- for each dependent table
  foreignKeys := dependentTables->object(t) ForeignKey
                                            {key := t.key;};
}

-- A root dependent part is mapped to a table as well
mapping DependentPart::part2table(in prefix : String) : Table
inherits fieldColumns {
  -- the name of a dependent part table is prefixed to
  -- avoid name confusion
  var dpTableName := prefix + recordName;
  name := dpTableName;

  -- we assign the columns for the top-level dependent part
  -- This includes the definition of a primary key
  columns := mainColumns +
             object Column {
                name := 'key_' + dpTableName;
                type := 'INTEGER';
                -- observe that we set the auxiliary property on true
                inKey
             };

  -- We calculate the columns of the sub parts by passing the
  -- result table of this mapping explicitly to the mapping
  -- part2columns. This means that part2columns will not construct
  -- a new table, but only update the existing one.
  -- We also pass it the new prefix.
  end { self.parts->map part2columns(result, dpTableName + '_'); }
}

-- the following mapping is recursive over the 'parts' property
-- of DependentPart
mapping DependentPart::part2columns(in prefix : String) : Table
inherits fieldColumns {
  -- for each nested dependent part, we simply add the newly
  -- resulting columns to those of the result table
  columns += mainColumns;

  -- we now recursively call the part2columns again with the
  -- result of this mapping
  var newPrefix := prefix + recordName + '_';
  end { self.parts->map part2columns(result, newPrefix); }
}

-- This mapping flattens a field to multiple columns
mapping Field::field2Columns (in prefix:String, in iskey:Boolean)
: Sequence(Column) {
  init {
    result :=
       if self.type.isTypeOf(PrimitiveType)
       then -- creates a sequence with a unique leaf field instance
```

```
         Sequence {
           object Column {
             type := self.type.asType(PrimitiveType)->
                           convertPrimitiveType();
             name := prefix + self.name;
             inKey := iskey;
           }
         }
       else if self.type.isTypeOf(PhysicalQuantityType) then
         self.type.asType(PhysicalQuantityType}->
               pqType2Columns(prefix, iskey);
         }
       else -- the case for value types is purely recursive
         self.type.asType(ValueType).fields->map
               field2Columns(prefix + self.name + '_', iskey);
       endif
     endif;
  }
}

-- We have one mapping which maps each unit of physical quantity
-- to a column.
mapping PhysicalQuantityType::pqType2Columns (in prefix : String,
                                              in iskey : Boolean)
: Sequence(Column) {
  init {
    result := self.units->map(u)
                object Column {
                  name := prefix + '_as_' + self.name + '_in_' + u;
                  type := self->convertPQType();
                  inKey := iskey;
                };
  }
}

-- Auxiliary functions for converting primitive types------------
query PrimitiveType::convertPrimitiveType() : String =
  if self.name = "int" then 'INTEGER'
  else if self.name = "float" then 'FLOAT'
  else if self.name = "long" then 'BIGINT'
  else 'DOUBLE'
  endif endif endif;

query PhysicalQuantityType::convertPQType() : String =
  if self.isTypeOf(IntPQType) then 'INTEGER'
  else if self.isTypeOf(FloatPQType) then 'FLOAT'
  else if self.isTypeOf(LongPQType) then 'BIGINT'
  else 'Double'
  endif endif endif;
```

References

ABB+01 C. Atkinson et al., *Component-based Product Line Engineering with UML*, Addison-Wesley Professional, 2001

ALMA ESO, *ALMA – Atacama Large Millimeter Array*, *http://www.eso.org/projects/alma/*

ANDR M. Bohlen, *androMDA*, *http://www.andromda.org/*

ANT Apache Software Foundation, *http://ant.apache.org/*

AOSD Aspect-Oriented Software Association, aosd.net, *http://aosd.net*

ARIS IDS Scheer, *Aris*, *http://www.aris.com/*

ASAR The Autosar Consortium, *Automotive Open System Architecture*, *http://www.autosar.org*

ASPJ Eclipse.org, *Aspect J*, *http://www.aspectj.org*

Ale01 A. Alexandrescu, *Modern C++ Design*, Addison-Wesley, 2001

BPEL Oasis, *Business Process Execution Language*, *http://www.oasis-open.org/*

BCEL Apache Group, *The Byte Code Engineering Library*, *http://jakarta.apache.org/bcel/index.html*

BCK98 L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison-Wesley, Second Edition, 2003

Bec00 K. Beck, *Extreme Programming Explained – Embrace Change*, Addison-Wesley, 2000

Bec02 K. Beck, *Test-Driven Development: by Example*, Addison-Wesley, 2002

Bet02 J. Bettin, *Model-Driven Architecture – Implementation & Metrics*, 2002, *http://www.softmetaware.com/mda-implementationandmetrics.pdf*

Bet04c J. Bettin, *Prozessauswirkungen von MDSD – Model-Driven Development*, special edition of OBJEKTspektrum 2004, *http://www.sigs.de/publications/os/2004/MDD/bettin_MDD_2004.pdf*

Bet04b J. Bettin, *Complexity & Dependency Management: Creating an Environment for Software Asset Evolution and Software Mass Customization*, 2004, *http://www.softmetaware.com/complexity-and-dependency-management.pdf*

Bos00 J. Bosch, *Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach*, Addison-Wesley Professional, 2000

BPML Business Process Modeling Language, *http://www.bpmi.org*

BPMN Business Process Modeling Notation, *http://www.bpmi.org*

CG01 J. C. Cleaveland, *Program Generators with XML and Java*, Prentice Hall PTR, 2001

CH05 K. Czarnecki, S. Helsen, *A Taxonomy and Categorization of Model Transformation Approaches*, *http://www.swen.uwaterloo.ca/~kczarnec/*

Cla04 M. Clark, *Pragmatic Project Automation*, The Pragmatic Programmers, 2004

Cle01 J. C. Cleaveland, *Program Generators with XML and Java*, Prentice Hall, 2001

CN01 P. Clements, L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley Professional, 2001

COMP University of Karlsruhe, The *COMPOST Composition System*, *http://www.info.uni-karlsruhe.de/~compost/*

Coc01 A. Cockburn, *Agile Software Development*, Addison-Wesley Professional, 2001

CRUI Sourceforge.net, *CruiseControl – Continuous Integration Toolkit*, *http://cruisecontrol.sourceforge.net/developers.html*

DRES Sourceforge.net, *Dresden OCL Toolkit*, *http://dresden-ocl.sourceforge.net/*

DSTG Delta Software Technology GmbH, *ANGIE*, *http://www.d-s-t-g.com/neu/pages/pageseng/common/prod_et_et04_frmset.htm*

EASY easymock.org, *EasyMock Mock Object Framework*, *http://www.easymock.org*

EC00 K. Czarnecki, U. Eisenecker, *Generative Programming*, Addison-Wesley Professional, 2000

ECLI Eclipse.org, *Eclipse*, *http://www.eclipse.org*

EMF Eclipse.org, *Eclipse Modeling Framework*, *http://www.eclipse.org/emf/*

EMPO Empowertec AG, *OCL Toolkit*, *http://www.empowertec.de/*

Eva03 E. Evans, *Domain-Driven Design – Tackling Complexity in the Heart of Software*, Addison-Wesley Professional, 2003

FMP University of Waterloo, *Feature Modelling Plugin*, *http://gp.uwaterloo.ca/fmp/*

FODA Carnegie Mellon Software Engineering Institute, *Feature-Oriented Domain Analysis*, *http://www.sei.cmu.edu/domain-engineering/FODA.html*

Fow99 M. Fowler, *Refactoring – Improving the Design of Existing Code*, Addison-Wesley Professional, 1999

Fow04 M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002

Fow05 M. Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?*, *http://www.martinfowler.com/articles/languageWorkbench.html*

Fra02 D. S. Frankel, *Model-Driven Architecture*, Wiley Publishing Inc., 2003

GME Institute for Software Integrated Systems, *GME – The Generic Modeling Environment*, *http://www.isis.vanderbilt.edu/Projects/gme/*

GHJ+94 E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995

GS04 J. Greenfield et al., *Software Factories – Assembling Applications with Patterns, Models, Frameworks, and Tools*, Wiley Publishing Inc., 2004

HIBE jboss.org, *Hibernate – Relational Persistence for Java and .NET*, *http://www.hibernate.org*

Hor04 G. Hohpe, B. Woolf, *Enterprise Integration Patterns – Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley Professional, 2003

HSQL HSQL Database Engine, *http://sourceforge.net/projects/hsqldb/*

HT04 A. Hunt, D. Thomas, *Pragmatic Unit Testing in Java with JUnit*, The Pragmatic Programmers, 2003

IUML Kennedy Carter, *iUML*, *http://www.kc.com/products/iuml.php*

JASS S. Chiba, *Javassist*, *http://www.csg.is.titech.ac.jp/~chiba/*

JB00 J. Bosch, *Design and Use of Software Architectures*, Addison-Wesley Professional, 2000

JBOS jboss.org, *JBoss Application Server*, *http://www.jboss.org*

JBR99 I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley Professional, 1999

JCC java.net, *JavaCC – The Java Parser Generator*, *http://javacc.dev.java.net/*

JMI Sun, *The Java Metadata Interface*, *http://java.sun.com/products/jmi/index.jsp*

KRB91 G. Kiczales, J. des Rivieres, D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991

Kos90 T. D. Koschmann, *The Common Lisp Companion*, John Wiley & Sons, 1990

Lad03 R. Laddad, *AspectJ in Action – Practical Aspect-Oriented Programming*, Manning Publications, 2003

M2T OMG, *MOF Model to Text Transformation Language RFP*, OMG document number ad/04-04-07, April 2004, *http://www.omg.org/docs/ad/04-04-07.pdf*

MC04 Metacase Consulting, *MetaEdit+*, *http://www.metacase.com*

MDAG *MDA™ Guide*, OMG document number omg/03-06-01, June 2003, *http://www.omg.org/docs/omg/03-06-01.pdf*

Mel02 S. Mellor, *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley Professional, 2002

MPS JetBrains, *Meta Programming System*, *http://www.jetbrains.com/mps/*

OASIS Organization for the Advancement of Structured Information Standards, *http://www.oasis-open.org*

OAW openArchitectureWare Group, *openArchitectureWare Generator*, *http://www.openarchitectureware.org*

OC++ S. Chiba, *OpenC++*, *http://www.csg.is.titech.ac.jp/~chiba/openc++.html*

OCL OMG, *Object Constraint Language 2.0 Specification*, OMG document number ptc/05-06-06, June 2005, *http://www.omg.org/docs/ptc/05-06-06.pdf*

Oes01 B. Oestereich et al., *Erfolgreich mit Objektorientierung*, Second Edition, Oldenbourg Wissenschaftsverlag, 2001

OMG Object Management Group, *http://www.omg.org*

OMGM Object Management Group, *MDA*, *http://www.omg.org/mda*

OMGP Object Management Group, *MDA UML Profile*, *http://www.omg.org/mda/specs.htm#Profiles*

OMGT Object Management Group, *MDA Committed Companies and their Products*, *http://www.omg.org/mda/committed-products.htm*

OWS+03 B. Oestereich et al., *Objektorientierte Geschäftsprozessmodellierung mit der UML*, Dpunkt Verlag, 2003

Par76 D. L. Parnas, *On the Design and Development of Program Families*, IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March 1976, pp. 1–9.

PBG04 T. Posch, K. Birken, M. Gerdom, *Basiswissen Softwarearchitektur*, Dpunkt Verlag, 2004

PLP Carnegie Mellon Software Engineering Institute, *Product Line Practice*, *http://www.sei.cmu.edu/plp/*

POSA1 F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*, John Wiley & Sons, 1996.

POSA2 D. Schmidt, M. Stal, H. Rohnert, F. Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, John Wiley & Sons, 2000

POSA3 M. Kircher, P. Jain, *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*, John Wiley & Sons, 2004

POSE Gentleware AG, *Poseidon for UML*, *http://www.gentleware.com/products*

PV pure-systems GmbH, *pure::variants*, *http://www.pure-systems.com/Variant_Management.49.0.html*

QVT OMG, *MOF 2.0 Query/View/Transformation Final Adopted Specification,* OMG document number ptc/05-11-01, November 2005, *http://www.omg.org/docs/ptc/05-11-01.pdf*

QVTP QVT Partners, QVTP Home Page, *http://www.qvtp.org/*

QVTR *MOF 2.0 Query / Views / Transformations RFP*, OMG document number ad/02-04-10, April 2002, *http://www.omg.org/docs/ad/02-04-10.pdf*

RV05 M. Rudorfer, M. Völter, *Domain-specific IDEs in Embedded Automotive Software*, EclipseCon 2005, *http://www.voelter.de/data/presentations/EclipseCon.pdf*

SCA IBM, *Service Component Architecture*, *http://www.ibm.com/developerworks/library/specification/ws-sca/*

SFW05 International Workshop on Software Factories at OOPSLA 2005, *http://softwarefactories.com/workshops/OOPSLA-2005/SoftwareFactoryWorkshopAnnouncement.htm*

STRT Apache Software Foundation, *The Apache Struts Application Framework*, *http://struts.apache.org/*

TOMC Apache Software Foundation, *Apache Tomcat*, *http://tomcat.apache.org/*

UMLX E. D. Willink, *UMLX: A Graphical Transformation Language for MDA*, *http://www.softmetaware.com/oopsla2003/willink.ppt*

VELO The Apache Jakarta Project, *Velocity*, *http://jakarta.apache.org/velocity/*

VKZ04 M. Völter, M. Kircher, U. Zdun, *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*, John Wiley & Sons, 2005

Voe02 M. Völter, *A Generative Component Infrastructure for Embedded Systems*, *http://www.voelter.de/data/pub/SmallComponents.pdf*

Voe03 M. Völter, *Program Generation – A Survey of Techniques and Tools*, *http://www.voelter.de/data/presentations/ProgramGeneration.zip*

Voe04 M. Völter, *Models and Aspects – Patterns for Handling Cross-Cutting Concerns in Model-Driver Software Development*, *http://www.voelter.de/data/pub/ModelsAndAspects.pdf*

VSW02 M. Völter, A. Schmid, E. Wolff, *Server Component Patterns*, John Wiley & Sons, 2002

WfMC The Workflow Management Coalition, *http://www.wfmc.org*

WL99 D. M. Weiss, C. T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley Professional, 1999

XDOC XDoclet Team, *XDoclet – Attribute-Oriented Programming*, *http://xdoclet.sourceforge.netxdoclet/index.html*

XMF Xactium, *XMF Mosaic*, *http://albini.xactium.com/content/*

XPDL *XML Process Definition Language*, *http://www.wfmc.org/standards/XPDL.htm*