

Domain-Specific Modeling

ENABLING FULL CODE GENERATION



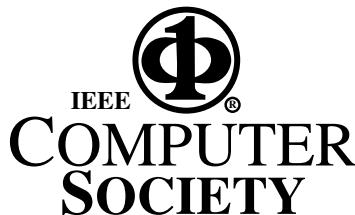
Steven Kelly
Juha-Pekka Tolvanen

Foreword by Dave Thomas

DOMAIN-SPECIFIC MODELING

Enabling Full Code Generation

STEVEN KELLY
JUHA-PEKKA TOLVANEN



A Wiley-Interscience Publication
JOHN WILEY & SONS, INC.

DOMAIN-SPECIFIC MODELING

Press Operating Committee

Chair Roger U. Fujii, <i>Vice President</i> <i>Northrop Grumman Mission Systems</i>	Editor-in-Chief Donald F. Shafer <i>Chief Technology Officer</i> <i>Athens Group, Inc.</i>
--	---

Board Members

Mark J. Christensen, *Independent Consultant*
Herb Krasner, *President, Krasner Consulting*
Ted Lewis, *Professor Computer Science, Naval Postgraduate School*
Hal Bergel, *Professor and Director School of Computer Science,
University of Nevada*
Phillip Laplante, *Associate Professor Software Engineering, Penn State University*
Richard Thayer, *Professor Emeritus, California State University, Sacramento*
Linda Shafer, *Professor Emeritus, University of Texas at Austin*
James Conrad, *Associate Professor, UNC-Charlotte*
Deborah Plummer, *Manager, Authored books*

IEEE Computer Society Executive Staff

David Hennage, *Executive Director*
Angela Burgess, *Publisher*

IEEE Computer Society Publications

The world-renowned IEEE Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available from most retail outlets.

Visit the CS Store at <http://computer.org/cspress> for a list of products.

IEEE Computer Society / Wiley Partnership

The IEEE Computer Society and Wiley partnership allows the CS Press authored book program to produce a number of exciting new titles in areas of computer science, computing and networking with a special focus on software engineering. IEEE Computer Society members continue to receive a 15% discount on these titles when purchased through Wiley or at wiley.com/ieeecs

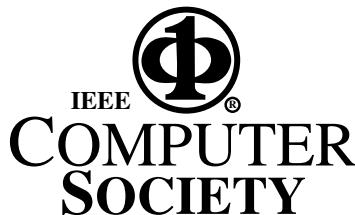
To submit questions about the program or send proposals please e-mail dplummer@computer.org or write to Books, IEEE Computer Society, 100662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314. Telephone +1-714-821-8380.

Additional information regarding the Computer Society authored book program can also be accessed from our web site at <http://computer.org/cspress>

DOMAIN-SPECIFIC MODELING

Enabling Full Code Generation

STEVEN KELLY
JUHA-PEKKA TOLVANEN



A Wiley-Interscience Publication
JOHN WILEY & SONS, INC.

Copyright © 2008 by John Wiley & Sons, Inc. All rights reserved

Published by John Wiley & Sons, Inc., Hoboken, New Jersey

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permission>.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data

Kelly, Steven, 1970-

Domain-specific modeling / Steven Kelly, Juha-Pekka Tolvanen.

p. cm.

ISBN 978-0-470-03666-2 (pbk.)

1. Programming languages (Electronic computers) 2. Computer software—Development. I. Tolvanen, Juha-Pekka. II. Title.

QA76.7.K45 2008

005.1–dc22

2007032132

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

CONTENTS

FOREWORD	XI
PREFACE	XIII
PART I: BACKGROUND AND MOTIVATION	1
1 INTRODUCTION	3
1.1 Seeking a Better Level of Abstraction / 3	
1.2 Code-Driven and Model-Driven Development / 4	
1.3 An Example: Modeling with a General-Purpose Language and a Domain-Specific Language / 7	
1.4 What is DSM? / 15	
1.5 When to Use DSM? / 18	
1.6 Summary / 19	
2 BUSINESS VALUE	21
2.1 Productivity / 21	
2.2 Quality / 27	
2.3 Leverage Expertise / 31	
2.4 The Economics of DSM / 34	
2.5 Summary / 41	

PART II: FUNDAMENTALS	43
3 DSM DEFINED	45
3.1 DSM Characteristics / 45	
3.2 Implications of DSM for Users / 52	
3.3 Difference from Other Modeling Approaches / 55	
3.4 Tooling for DSM / 59	
3.5 Summary / 61	
4 ARCHITECTURE OF DSM	63
4.1 Introduction / 63	
4.2 Language / 68	
4.3 Models / 77	
4.4 Code Generator / 79	
4.5 Domain Framework and Target Environment / 86	
4.6 DSM Organization and Process / 88	
4.7 Summary / 92	
PART III: DSM EXAMPLES	93
5 IP TELEPHONY AND CALL PROCESSING	97
5.1 Introduction and Objectives / 97	
5.2 Development Process / 100	
5.3 Language for Modeling Call Processing Services / 101	
5.4 Modeling IP Telephony Services / 111	
5.5 Generator for XML / 112	
5.6 Framework Support / 117	
5.7 Main Results / 118	
5.8 Summary / 118	
6 INSURANCE PRODUCTS	120
6.1 Introduction and Objectives / 120	
6.2 Development Process / 121	
6.3 Language for Modeling Insurances / 123	
6.4 Modeling Insurance Products / 131	

6.5	Generator for Java / 132	
6.6	Framework Support / 138	
6.7	Main Results / 138	
6.8	Summary / 139	
7	HOME AUTOMATION	140
7.1	Introduction and Objectives / 140	
7.2	Development Process / 142	
7.3	Home Automation Modeling Language / 144	
7.4	Home Automation Modeling Language in Use / 150	
7.5	Generator / 153	
7.6	Main Results / 157	
7.7	Summary / 158	
8	MOBILE PHONE APPLICATIONS USING A PYTHON FRAMEWORK	160
8.1	Introduction and Objectives / 160	
8.2	Development Process / 163	
8.3	Language for Application Modeling / 164	
8.4	Modeling Phone Applications / 174	
8.5	Generator for Python / 176	
8.6	Framework Support / 184	
8.7	Main Results / 185	
8.8	Extending the Solution to Native S60 C++ / 185	
8.9	Summary / 189	
9	DIGITAL WRISTWATCH	191
9.1	Introduction and Objectives / 191	
9.2	Development Process / 193	
9.3	Modeling Language / 193	
9.4	Models / 207	
9.5	Code Generation for Watch Models / 212	
9.6	The Domain Framework / 220	
9.7	Main Results / 222	
9.8	Summary / 224	

PART IV: CREATING DSM SOLUTIONS	225
10 DSM LANGUAGE DEFINITION	227
10.1 Introduction and Objectives / 227	
10.2 Identifying and Defining Modeling Concepts / 228	
10.3 Formalizing Languages with Metamodeling / 247	
10.4 Defining Language Rules / 250	
10.5 Integrating Multiple Languages / 253	
10.6 Notation for the Language / 257	
10.7 Testing the Languages / 261	
10.8 Maintaining the Languages / 264	
10.9 Summary / 266	
11 GENERATOR DEFINITION	267
11.1 “Here’s One I Made Earlier” / 268	
11.2 Types of Generator Facilities / 270	
11.3 Generator Output Patterns / 276	
11.4 Generator Structure / 297	
11.5 Process / 304	
11.6 Summary / 308	
12 DOMAIN FRAMEWORK	311
12.1 Removing Duplication from Generated Code / 313	
12.2 Hiding Platform Details / 315	
12.3 Providing an Interface for the Generator / 317	
12.4 Summary / 327	
13 DSM DEFINITION PROCESS	329
13.1 Choosing Among Possible Candidate Domains / 329	
13.2 Organizing for DSM / 330	
13.3 Proof of Concept / 335	
13.4 Defining the DSM Solution / 339	
13.5 Pilot Project / 345	
13.6 DSM Deployment / 347	
13.7 DSM as a Continuous Process in the Real World / 352	
13.8 Summary / 356	

14 TOOLS FOR DSM	357
14.1 Different Approaches to Building Tool Support /	357
14.2 A Brief History of Tools /	359
14.3 What is Needed in a DSM Environment /	365
14.4 Current Tools /	390
14.5 Summary /	395
15 DSM IN USE	397
15.1 Model Reuse /	397
15.2 Model Sharing and Splitting /	400
15.3 Model Versioning /	404
15.4 Summary /	407
16 CONCLUSION	408
16.1 No Sweat Shops—But no Fritz Lang’s Metropolis Either /	409
16.2 The Onward March of DSM /	410
APPENDIX A: METAMODELING LANGUAGE	411
REFERENCES	415
INDEX	423

FOREWORD

I have been an enthusiastic follower of Juha-Pekka Tolvanen and Steven Kelly's work since meeting them in the 1990s at ECOOP and OOPSLA conferences. When people mention the talented minds of Finland, my first association is not Nokia or Linux, but MetaCase.

I have spent my career searching for ways to empower application and product developers who have domain knowledge to simply and quickly express their knowledge in a form that can be readily consumed by machines. In almost every case, this has led to a little language expressed in text, diagrams or a framework in a friendly OO language such as Smalltalk or Ruby. Today we call these little languages Domain Specific Languages.

Domain Specific Language engineering is increasingly recognized as an important productivity technique which increases businesses' agility in the design, development and configuration of their products. DSLs provide a means for narrowing the communication gap between users and developers. They allow requirements and specifications to be made more tangible to users and developers. Finally, they document critical portions of the knowledge associated with an application or product thereby reducing the life cycle costs in evolving that application.

Juha-Pekka Tolvanen and Steven Kelly are pioneers in the world of DSLs. These experts have worked for over a decade in the design of DSLs and the implementation of commercial DSL tooling at MetaCase. Their popular tutorials and workshops have been featured at major conferences all over the world. Juha-Pekka and Steven have worked closely with customers to implement DSLs specific to their needs. Few in our industry have their breadth of knowledge and experience.

When I first encountered their work I assumed from the name MetaCase and the demonstrations that it was a very clever constraint drawing framework which could be

used to build visual modeling tools. Their frameworks and tools made it easy to express a custom visual notation allowing one to have a full-blown visual modeling tool in weeks rather than years. Anyone with experience using modern graphical modeling frameworks such as Eclipse GMF will be very impressed with how quickly one can define a new visual language.

I soon learned that their real interests were not confined to visual frameworks, but rather they too had a passion for domain specific languages. They partnered with their clients to help them model, design and implement DSLs for their business needs. In doing so they developed the process, practices, tooling and most importantly pragmatics for the industrial use of DSLs.

This book presents practical design and development lessons covering topics including domain modeling, language definition, code generation and DSL tooling. They thoroughly discuss the business and technical benefits of using a DSL. While being proponents of the approach they provide sound arguments for when it is appropriate to consider using a DSL. Importantly, they explore issues associated with the evolution of software using domain specific languages.

The case studies in telephony, insurance, home automation and mobile applications clearly illustrate the use of DSLs for different domains and are based on actual client experiences. They provide the reader with the benefit of a real world perspective on DSL design and implementation. Students and Educators will appreciate the Digital Watch which is a complete pedagogical example used in their popular tutorials.

I have had the pleasure of observing the authors during their journey from research to practice. Far too often the principals in small technology companies are too busy doing to take time to share their unique experiences. We are very fortunate that Juha-Pekka and Steven have made the effort to produce this practical book based on their experiences. I have attended their tutorials and read drafts of the book, each time learning something new. This book is a must read for anyone who wants to understand the appropriate use, benefits and practices of DSL engineering.

DAVE THOMAS

PREFACE

This book is for experienced developers who want to improve the productivity of their software development teams. Productivity is not just about speed, but also about the quality of the systems produced, both intrinsically and as perceived by the people who have to use them. Our focus is on the power and tools we can put in the hands of our software developers, and how to make the best use of the experience and skills already present in our teams.

We are not talking here about squeezing another 20% out of our existing developers, programming languages, or development environments. Our industry is long overdue for a major increase in productivity: the last such advance was over 30 years ago, when the move from assemblers to compilers raised productivity by around 500%. Our experiences, and those of our customers, colleagues, and competitors, have shown that a similar or even larger increase is now possible, through what we call Domain-Specific Modeling. Indeed, the early adopters of DSM have been enjoying productivity increases of 500–1000% in production for over 10 years now.

WHAT IS DOMAIN-SPECIFIC MODELING?

Domain-Specific Modeling requires an experienced developer to create three things, which together form the development environment for other developers in that domain. A domain here is generally a highly focused problem domain, typically worked on by 5–500 developers in the same organization. The three things are as follows:

- A domain-specific modeling language

- A domain-specific code generator
- A domain framework

With these three parts of a DSM solution in place, the developers need only create models in the DSM language, and the applications are automatically generated as code running on top of the domain framework. The generated code need not be edited or even looked at, thus completing the analogy with the move from assemblers to compilers: with each major leap of our industry, developers need no longer look at the previous generation's source format.

The changes wrought by Domain-Specific Modeling may seem radical, but at its heart are three simple practices that any experienced software engineer will recognize:

- Don't repeat yourself
- Automate after three occurrences
- Customized solutions fit better than generic ones

Other books have discussed these principles, the basic ideas of modeling, and how to move modeling to be more central to the development process. In this book, we will explain what Domain-Specific Modeling is, why it works, and how to create and use a DSM solution to improve the productivity of your software development.

BACKGROUND OF THE AUTHORS

Both authors have been working in DSM for over 15 years at the time of publication. In that time we have seen DSM successfully applied in a vast array of different problem domains, to create applications in a similarly broad collection of programming languages and platforms. Across the board, DSM has consistently increased productivity by a factor of 5–10. We take no personal credit for those results: the approach itself simply works.

Similar results have been achieved by our customers creating DSM solutions on their own, and by people using different tools. However, there have also been plenty of failures in those two situations: the approach and its tooling are not so self-evident that anybody can create them anew *in vacuo*. We too have made plenty of mistakes along the way and have learned from and with our customers. In particular, by teaching our customers and others, we have been forced to put our experience into words and try various ways of modularizing and presenting it. By writing this book, we hope to be able to pass on our experience in an easily digested form, to help you to have a smoother path to success.

An important part of smoothing the way to success in creating your own DSM solution is good tooling. It is possible to create your own modeling tool from scratch using graphics frameworks and so on, but for all but the largest groups of developers such an approach will be prohibitively expensive and time consuming. There are currently several DSM tools available that will allow you to simply

specify your modeling language, and which offer in return a modeling tool for that language.

The authors have played a central role in the development of one such DSM tool, MetaEdit+. The earlier MetaEdit was created as a research prototype in 1991, and released commercially in 1993, being the first tool to allow people to define their modeling languages graphically. As is common with such first versions, the original architecture was found to be too limiting for large-scale commercial use, lacking support for multiple simultaneous modelers, multiple modeling languages, and multiple integrated models. These and other requirements were met in MetaEdit+, released commercially in 1995. MetaEdit+ was created from a clean slate, but building on the experience gained with MetaEdit: a prime case of “build one to throw away.”

This book, however, like DSM itself, is not limited to or focused on any particular tool. As far as possible, we have steered clear of tool-specific details. The principles presented here can be applied in any mature tool, and the benefits of DSM can be obtained—albeit at a higher cost—even with immature tools. That at least was our experience and that of our customers with the first version of MetaEdit+, which was definitely in that category in terms of its user interface!

HOW TO READ THIS BOOK

The book is divided into four main parts.

- Part I explains what DSM is (Chapter 1) and what value it has to offer (Chapter 2).
- Part II defines DSM, both with respect to current practices (Chapter 3) and in terms of its constituent parts (Chapter 4).
- Part III presents five in-depth examples of DSM in increasingly complex domains (Chapters 5–9).
- Part IV teaches how to create the various parts of a DSM solution (Chapters 10–12), discusses the processes and tools for creating and using the DSM solution (Chapters 13–15), and wraps up with a summary and conclusions in Chapter 16.

In Parts I and II after Chapter 1, readers will find material directed toward those of a more technical, business minded, or academic bent, and should feel free to skip sections, returning to them later if necessary. The examples in Part III build on each other and are also often used in explaining the principles in Part IV, so readers would be advised to at least skim all the examples. In Part IV, the various parts of a DSM solution may all be the responsibility of one person, or then they may be split between two or more people. Chapters 11 and 12, and to a slightly lesser extent Chapters 14 and 15, will make most sense to experienced programmers. Chapters 10 and 13 may interest those in more of an architect or project management role.

The book web site at <http://dsmbook.com> contains updates, the modeling languages from Part III, and a free MetaEdit+ demo.

ACKNOWLEDGMENTS

With the solid base we had to build on, subsequent years of work with colleagues and customers, and great interactions with our peers in this community, it is clear that we are indebted to far too many people to mention even a representative sample of them here. Even just the OOPSLA workshops on DSM account for several hundreds of authors over the past seven years: seeing so many others coming to similar conclusions and achieving similar successes played a major role in encouraging us on this path.

Some people however simply must be mentioned, so we shall first return to the beginnings of DSM to thank Richard Welke, Kalle Lyytinen, and Kari Smolander for the research that we built on, and for an environment where what was right mattered more than who was right. From that MetaPHOR project to the present day Pentti Marttiin and Matti Rossi have played a central role in keeping us sane, amused, and at least tolerably fit, not to mention an innumerable amount of discussions ranging over all possible metalevels.

All our colleagues at MetaCase have contributed to pushing DSM forward, but we would particularly like to thank Janne Luoma and Risto Pohjonen for their work, discussions and support, as well as their major roles in some of the example cases in Part III. Many, many customers and prospects have shaped and stretched our understanding of DSM as it has been applied to ever new areas. Most are happier keeping their competitive advantage to themselves, but we simply must mention Jyrki Okkonen for his pioneering work with DSM at Nokia. We would also like to thank our esteemed competitors, in particular, Microsoft's Alan Cameron Wills and Aali Alikoski, for their genuine and unprecedented levels of cooperation.

For the book itself, we would first like to thank Stan Wakefield for suggesting we write it, and for playing matchmaker between the publishers and us. Rachel Witmer at Wiley has earned our gratitude for her patience in the face of implausibly long deadline overruns, as DSM took off as a hot topic at just the right and the wrong time. Dave Thomas has fought our corner in North America for a long time, and we are deeply honored to have him write our foreword. In addition to many of those already mentioned, we would like to thank the following for their contribution to or feedback on the book: Jeff Gray of the University of Alabama at Birmingham, Angelo Hulshout of ICT NoviQ, Juha Pärssinen of VTT, Laurent Safa of Matsushita Electric Works, and Jos Warmer of Ordina.

PART I

BACKGROUND AND MOTIVATION

We start by introducing Domain-Specific Modeling (DSM). First we highlight the difference to manual coding and to modeling languages originating from the code world. This difference is demonstrated with a practical example. In Chapter 2, we describe the main benefits of DSM: increase in productivity and quality as well as use of expertise to share the knowledge within the development team.

CHAPTER 1

INTRODUCTION

1.1 SEEKING A BETTER LEVEL OF ABSTRACTION

Throughout the history of software development, developers have always sought to improve productivity by improving abstraction. The new level of abstraction has then been automatically transformed to the earlier ones. Today, however, advances in traditional programming languages and modeling languages are contributing relatively little to productivity—at least if we compare them to the productivity increases gained when we moved from assembler to third generation languages (3GLs) decades ago. A developer could then effectively get the same functionality by writing just one line instead of several earlier. Today, hardly anybody considers using UML or Java because of similar productivity gains.

Here Domain-Specific Modeling (DSM) makes a difference: DSM raises the level of abstraction beyond current programming languages by specifying the solution directly using problem domain concepts. The final products are then generated from these high level specifications. This automation is possible because both the language and generators need fit the requirements of only one company and domain. We define a domain as an area of interest to a particular development effort. Domains can be a horizontal, technical domain, such as persistency, user interface, communication, or transactions, or a vertical, functional, business domain, such as telecommunication, banking, robot control, insurance, or retail. In practice, each DSM solution focuses on even smaller domains because the narrower focus enables better possibilities for

automation and they are also easier to define. Usually, DSM solutions are used in relation to a particular product, product line, target environment, or platform.

The challenge that companies—or rather their expert developers—face is how to come up with a suitable DSM solution. The main parts of this book aim to answer that question. We describe how to define modeling languages, code generators and framework code—the key elements of a DSM solution. We don’t stop after creating a DSM solution though. It needs to be tested and delivered to modelers and to be maintained once there are users for it. The applicability of DSM is demonstrated with five different examples, each targeting a different kind of domain and generating code for a different programming language. These cases are then used to exemplify the creation and use of DSM.

New technologies often require changes from an organization: What if most code is generated and developers work with domain-specific models? For managers, we describe the economics of DSM and its introduction process: how to estimate the suitability of the DSM approach and what kinds of expertise and resources are needed. Finally, we need to recognize the importance of automation for DSM creation: tools for creating DSM solutions. This book is not about any particular tool, and there is a range of new tools available helping to make creation of a DSM solution easier, allowing expert developers to encapsulate their expertise and make work easier, faster, and more fun for the rest.

1.2 CODE-DRIVEN AND MODEL-DRIVEN DEVELOPMENT

Developers generally differentiate between modeling and coding. Models are used for designing systems, understanding them better, specifying required functionality, and creating documentation. Code is then written to implement the designs. Debugging, testing, and maintenance are done on the code level too. Quite often these two different “media” are unnecessarily seen as being rather disconnected, although there are also various ways to align code and models. Figure 1.1 illustrates these different approaches.

At one extreme, we don’t create any models but specify the functionality directly in code. If the developed feature is small and the functionality can be expressed directly in code, this is an approach that works well. It works because programming environments can translate the specification made with a programming language into an executable program or other kind of finished product. Code can then be tested and debugged, and if something needs to be changed, we change the code—not the executable.

Most software developers, however, also create models. Pure coding concepts are, in most cases, too far from the requirements and from the actual problem domain. Models are used to raise the level of abstraction and hide the implementation details. In a traditional development process, models are, however, kept totally separate from the code as there is no automated transformation available from those models to code. Instead developers read the models and interpret them while coding the application and producing executable software. During

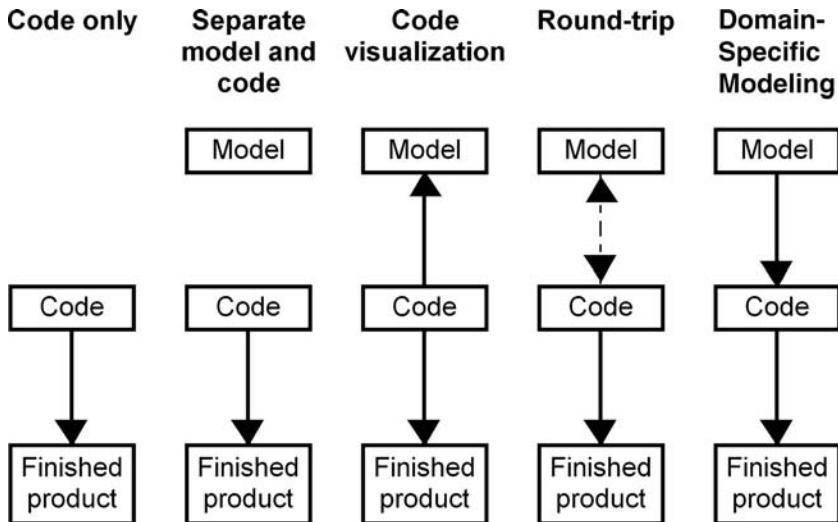


FIGURE 1.1 Aligning code and models

implementation, models are no longer updated and are often discarded once the coding is done. This is simply because the cost of keeping the models up-to-date is greater than the benefits we get from the models. The cost of maintaining the same information in two places, code and models, is high because it is a manual process, tedious, and error prone.

Models can also be used in reverse engineering: trying to understand the software after it is designed and built. While creating model-based documentation afterwards is understandable, code visualization can also be useful when trying to understand what a program does or importing libraries or other constructs from code to be used as elements in models. Such models, however, are typically not used for implementing, debugging, or testing the software as we have the code.

Round-tripping aims to automate the work of keeping the same information up-to-date in two places, models and code. Round-tripping works only when the formats are very similar and there is no loss of information between the translations. In software development, this is true in relatively few areas and typically only for the structural specifications. For instance, a model of a schema can be created from a database and a database schema can be generated from a model. Round-tripping with program code is more problematic since modeling languages don't cover the details of programming languages and vice versa. Usually the class skeletons can be shown in models but the rest—behavior, interaction, and dynamics—are not covered in the round-trip process and they stay in the code. This partial link is represented with a dotted line in Fig. 1.1.

If we inspect the round-trip process in more detail, we can also see that mappings between structural code and models are not without problems. For example, there are no well-defined mappings on how different relationship types used in class diagrams, such as association, aggregation, and composition, are related to program

code. Code does not explicitly specify these relationship types. One approach to solve this problem is to use just a single source, usually the code, and show part of it in the models. A classical example is to use only part of the expressive power of class diagrams. That parts is, where the class diagram maps exactly to the class code. This kind of alignment between code and models is often pure overhead. Having a rectangle symbol to illustrate a class in a diagram and then an equivalent textual presentation in a programming language hardly adds any value. There is no raise in the level of abstraction and no information hiding, just an extra representation duplicating the same information.

In model-driven development, we use models as the primary artifacts in the development process: we have source models instead of source code. Throughout this book, we argue that whenever possible this approach should be applied because it raises the level of abstraction and hides complexity. Truly model-driven development uses automated transformations in a manner similar to the way a pure coding approach uses compilers. Once models are created, target code can be generated and then compiled or interpreted for execution. From a modeler's perspective, generated code is complete and it does not need to be modified after generation. This means, however, that the “intelligence” is not just in the models but in the code generator and underlying framework. Otherwise, there would be no raise in the level of abstraction and we would be round-tripping again. The completeness of the translation to code should not be anything new to code-only developers as compilers and libraries work similarly. Actually, if we inspect compiler development, the code expressed, for instance in C, is a high-level specification: the “real” code is the running binary.

Model-Driven Development is Domain-Specific To raise the level of abstraction in model-driven development, both the modeling language and the generator need to be domain-specific, that is, restricted to developing only certain kinds of applications. While it is obvious that we can't have only one code generator for all software, it seems surprising to many that this applies for modeling languages too.

This book is based on the finding that while seeking to raise the level of abstraction further, languages need to be better aware of the domain. Focusing on a narrow area of interest makes it possible to map a language closer to the actual problem and makes full code generation realistic—something that is difficult, if not impossible, to achieve with general-purpose modeling languages. For instance, UML was developed to be able to model all kinds of application domains (Rumbaugh et al., 1999), but it has not proven to be successful in truly model-driven development. If it would, the past decade would have demonstrated hundreds of successful cases. Instead, if we look at industrial cases and different application areas where models are used effectively as the primary development artifact, we recognize that the modeling languages applied were not general-purpose but domain-specific. Some well-known examples are languages for database design and user interface development. Most of the domain-specific

languages are made in-house and typically less widely publicized. They are, however, generally more productive, having a tighter fit to a narrower domain, and easier to create as they need only satisfy in-house needs. Reported cases include various domains such as automotive manufacturing (Long et al., 1998), telecom (Kieburtz et al., 1996; Weiss and Lai, 1999), digital signal processing (Sztipanovits et al., 1998), consumer devices (Kelly and Tolvanen, 2000), and electrical utilities (Moore et al., 2000). The use of general-purpose languages for model-driven development will be discussed further in Chapter 3.

1.3 AN EXAMPLE: MODELING WITH A GENERAL-PURPOSE LANGUAGE AND A DOMAIN-SPECIFIC LANGUAGE

Let's illustrate the use of general-purpose modeling and domain-specific modeling through a small example. For this illustration, we use a domain that is well known since it is already in our pockets: a mobile phone and its applications. Our task as a software developer for this example is to implement a conference registration application for a mobile phone. This small application needs to do just a few things: A user can register for a conference using text messages, choose among alternative payment methods, view program and speaker information, or browse the conference program via the web. These are the basic requirements for the application.

In addition to these requirements, software developers need also to master the underlying target environment and available platform services. When executing the application in a mobile phone or other similar handheld device, we normally need to know about the programming model to be followed, libraries and application program interfaces (APIs) available, as well as architectural rules that guide application development for a particular target environment.

We start the comparison by illustrating possible development processes, first based on a general-purpose modeling language and then based on a domain-specific modeling language. UML is a well known modeling language created to specify almost any software system. As it is intended to be universal and general-purpose according to its authors (Rumbaugh et al., 1999), it can be expected to fit our task of developing the conference registration application. The domain-specific language is obviously made for developing mobile applications.

1.3.1 UML Usage Scenario

Use of UML and other code-oriented approaches normally involves an initial modeling stage followed by manual coding to implement the application functionality. Design models either stay totally separate from the implementation or are used to produce parts of the code, such as the class skeletons. The generated code is then modified and extended by filling in the missing parts that could not be generated from UML models. At the end of the development phase, most of the models made will be thrown away as they no longer specify what was actually

developed while programming the application. The cost of updating these models is too high as there is no automation available.

Modeling Application Structure Let's look at this scenario in more detail using our conference registration example. The common way to use UML for modeling starts with defining the use cases, as illustrated in Fig. 1.2. For each identified use case, we would specify in more detail its functionality, such as actions, pre- and postconditions, expected result, and possible exceptions. We would most likely describe these use cases in text rather than in the diagram. After having specified the use cases, we could review them with a customer through a generated requirements document or, if the customer can understand the language, at least partly with the diagram.

Although use cases raise the level of abstraction away from the programming code, they do not enable automation via code generation. Use case models and their implementation stay separate. Having identified what the customer is looking for, we would move to the next phase of modeling: Define a static structure of the application with class diagrams, or alternatively start to specify the behavior using sequence diagrams or state diagrams. A class diagram in Fig. 1.3 shows the structure of the application: some of the classes and their relationships.

While creating the class diagram, we would start to consider the application domain: what widgets are available and how they should be used, where to place

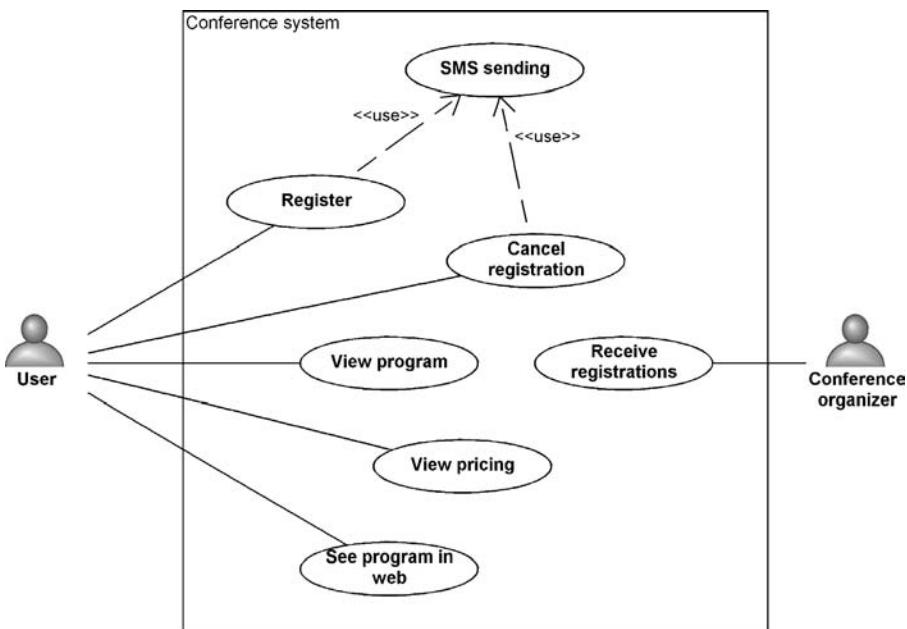


FIGURE 1.2 Use cases of the conference application

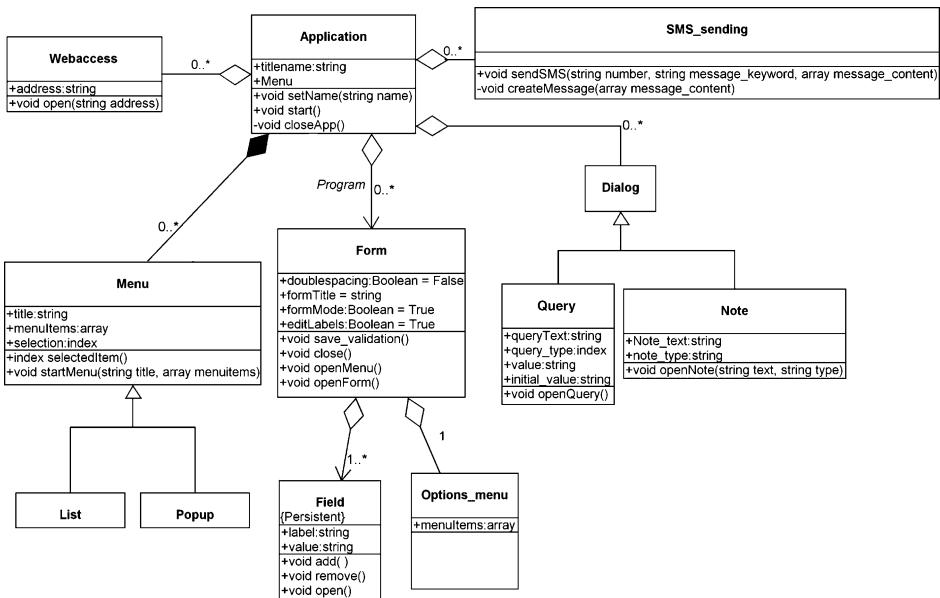


FIGURE 1.3 Class structure of the application

menus, what is required to send text messages, and so forth. Unfortunately, UML will not help us at all since it does not know anything about mobile applications. Our job as developers is to first find a solution using the domain concepts and then map the solution into the UML concepts. So instead of describing the phone application, we would describe classes, their attributes and operations, and various connections they may have. Even if our class diagram only targets analysis or is platform independent, it would not change the situation as we would still be using implementation concepts to describe our solution.

Adding Implementation Details to the Models In a later design phase, we need to extend the analysis model with implementation details. Here we would expect the coding concepts of the class diagram to be helpful as they map to implementation details, but in reality UML helps us very little during the application design. We could draw whatever we like into the class diagram! It would be relatively easy, and likely too, to end up with a design that will never work, for example, because it breaks the architecture rules and programming model of the phone. To make the model properly, we need to study the phone framework, find out its services, learn its API, and study the architectural rules. These would then be kept in mind while creating the models and entering their details. For instance, sending text messages requires that we use a SendSMS operation (see Fig. 1.3) and give it the right parameters: a mandatory phone number followed by a message header and message contents.

After finishing the class diagram, we could generate skeleton code and continue writing the code details, the largest part, manually. Alternatively, we could continue modeling and create other kinds of designs that cover those parts the class diagram did not specify. Perhaps even in parallel to static structures, we would also specify what the application does. Here we would start to think about the application behavior, such as menu actions, accessing the web, user navigation, and canceling during the actions.

Modeling Application Behavior In the UML world, we could apply state machines, collaboration diagrams, or perhaps sequence diagrams to address application behavior. With a sequence diagram, our designs could look something like Fig. 1.4.

When specifying the interaction between the objects, we would need to know in more detail the functionality offered by the device, the APIs to use, what they return, the expected programming model to be followed, and architectural rules to be obeyed. We simply can't draw the sequence diagram without knowing what the phone can do! So for many of the details in a design model, we must first consult libraries and APIs.

Unfortunately, sequence diagrams would not be enough for specifying the application behavior as they don't adequately capture details like alternative choices or decisions. We can apply other behavioral modeling languages, like activity diagrams or state diagrams, to specify these. Figure 1.5 illustrates an activity diagram that shows how the application handles conference unregistration. This model is partly related to the code, for example, through services it calls, but not adequately so that it could be used for code generation. We could naturally fill more implementation details into the activity diagram and start using the activity modeling language as a programming language, but most developers switch to a programming language to make it more concise.

If we were to continue our example, the activity diagrams could be made to specify other functions as well, but to save space we have omitted them. We should also note that there is no explicit phase or time to stop the modeling effort. How can we know when the application is fully designed without any guidance as to what constitutes a full design? If we had UML fever (see Bell, 2004), we could continue the modeling effort and create further models. There are still nine other kinds of modeling languages. Should we use them and stop modeling here or should we have stopped earlier? Since development is usually iterative in this model creation process, we most likely would also update the previously made class diagrams, sequence diagrams, activity diagrams, etc. If we wouldn't do that our models would not be consistent. This inconsistency might be fine for sketching but not for model-driven development.

Implementing the Application The example models described earlier are clearly helpful for understanding and documenting the application. After all the modeling work, we could expect to get more out of the created models too. Generate code perhaps? Unfortunately a skeleton is the best we can generate here and then

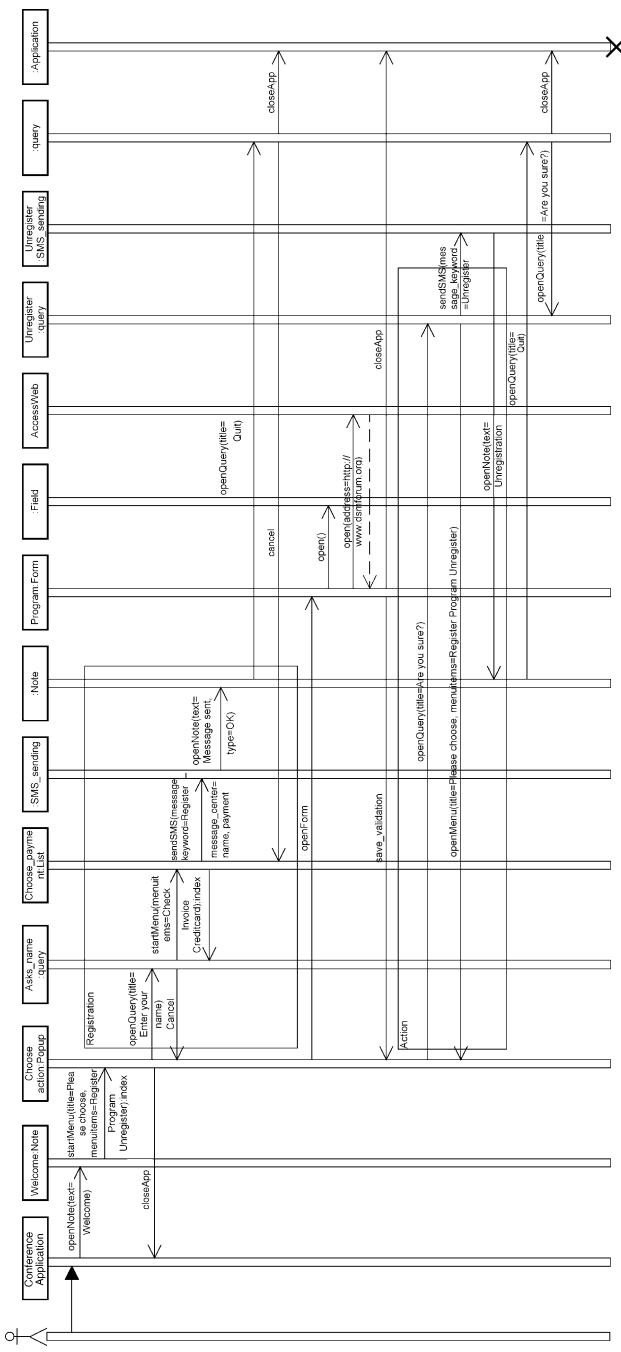


FIGURE 1.4 A view of the application behavior

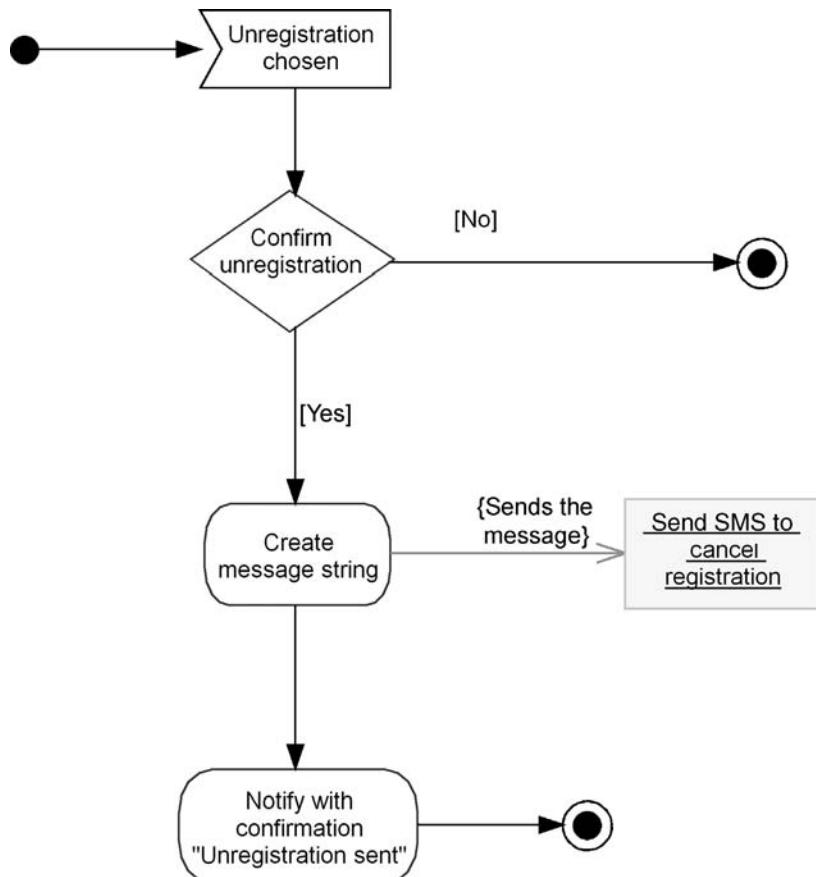


FIGURE 1.5 Activity diagram specifying steps while unregistering from a conference

continue by modifying the generated code to implement the functionality and logic—the largest part of the application. To get the finished application, we would implement it by writing the application code.

At this point, our models and code start to be separate. During the programming, we will face aspects that were inadequately specified, false, or totally ignored while modeling. It is also likely that our design models did not recognize the architectural rules that the application must follow to execute. After all, UML models did not know about the libraries, framework rules, and programming model for our mobile applications. As the changes made to the code during implementation are no longer in sync with the designs, we need to decide what to do with the models. Should we take the time to update the models or ignore them and throw them away? Updating the models requires manual work as the semantics of the code is different than most of the concepts used in UML models. Even keeping part of the class diagram

up-to-date in parallel with the code does not help much since it just describes the implementation code.

1.3.2 DSM Usage Scenario

Next let's contrast the above UML approach to DSM. Here the modeling language is naturally made for developing applications for mobile phones. Figure 1.6 shows a model that describes the conference registration application. If you are familiar with some phone applications, like a phone book or calendar, you most likely already understand what the application does. This model also describes the application sufficiently unambiguously, that it can be generated from this high-level specification. Take a look of the modeling in Fig. 1.6 and then compare it to UML models that did not get even close to having something running.

In Domain-Specific Modeling we would start modeling by directly using the domain concepts, such as Note, Pop-up, SMS, Form, and Query. These are specific to the mobile phone's services and its user-interface widgets. The same concepts are also language constructs. They are not new or extra concepts as we must always apply these regardless of how the mobile application is implemented. With this modeling language, we would develop the conference registration application by

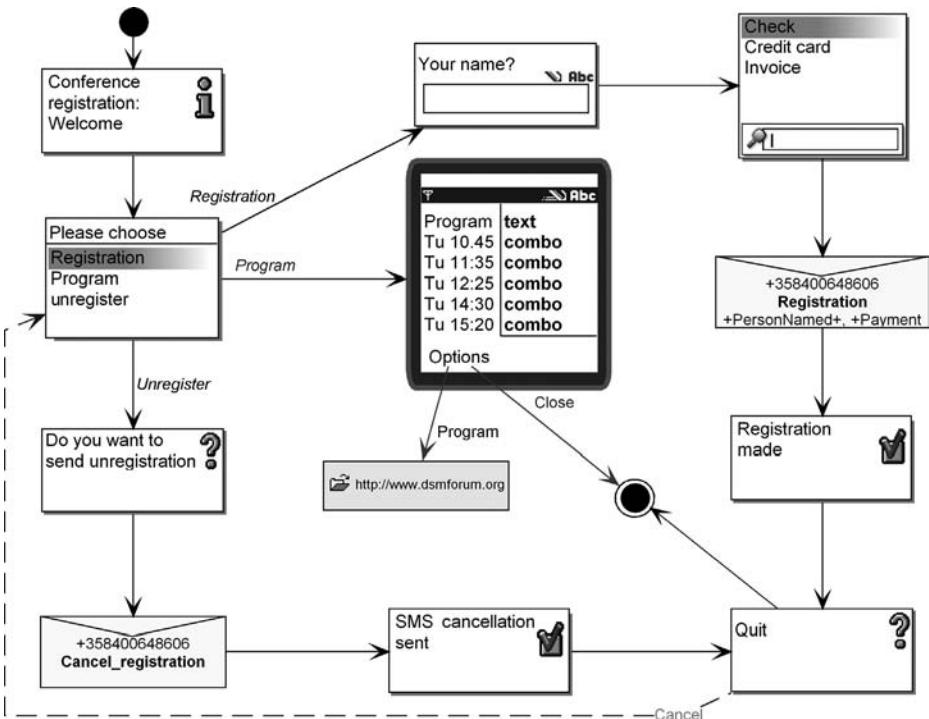


FIGURE 1.6 Conference registration application

adding elements to the model and linking them together to follow the flow of navigation and application control.

The Domain-Specific Modeling language also includes domain rules that prevent us from making illegal designs. For example, after sending an SMS message, only one UI element or phone service can be triggered. The modeling language “knows” this rule and won’t let us draw another flow from an SMS element. If a relevant part of the design is still missing, for example, the application never reaches a certain widget, model checking reports warn us about incomplete parts. This means that we do not need to master the details of the software architecture and required programming model, and we can focus instead on finding the solution using the phone concepts.

As can be seen from this DSM example, all the implementation concepts are hidden from the models and are not even necessary to know. This shows that the level of abstraction in models is clearly higher. Once the design, or a relevant part of it, is done, we can generate and execute the conference registration application in a phone. There is no need to map the solution manually to implementation concepts in code or in UML models visualizing the code. Unlike with UML, in DSM we stopped modeling when the application was ready. If the application needs to be modified, we do it in the model.

For the DSM case, we left the code out of the scenario since it is not so relevant anymore. This is in sharp contrast to the UML approach, where the modeling constructs originate from the code constructs. Naturally in DSM code is also generated, but since the generator is defined by the developers of the DSM language and not by the developers of this particular mobile application, we won’t inspect the implementation in code here. Later in Part III of this book we look at examples of DSM in more detail, including this mobile application case.

1.3.3 Comparing UML and DSM

The above example illustrates several key differences between general-purpose and Domain-Specific Modeling languages. Do the comparison yourself and think about the following questions:

- Which of these two is a faster way to develop the application?
- Which leads to better quality?
- Which language guided the modeler in developing a working application?
- Which specifications are easier to read and understand?
- Which requires less modeling work?
- Which approach detects errors earlier or even prevents them from happening?
- Which language is easier to introduce and learn?

No doubt, DSM performed better compared to UML or any other general-purpose language. For example, the time to develop the application using DSM is a fraction of the time to create the UML diagrams and write the code manually. DSM also prevents errors early on since the language does not allow illegal designs or

designs that don't follow the underlying architectural rules. The code produced is also free of most kinds of careless mistakes, syntax, and logic errors since a more experienced developer has specified the code generator. We inspect the effect of DSM development on productivity, product quality, and required developer expertise in more detail in the next chapter.

One example obviously cannot cover the wide spectrum of possible development situations and application domains. The mobile example is one of thousands, if not hundreds of thousands, of different application domains. In principle, different DSM solutions can be used in all of them since every development project needs to find a solution in the problem domain and map it into an implementation in the solution domain. All development projects have faced the same challenge. In Part III of this book, we inspect different cases of DSM and aim to cover a wider range of problem domains generating different kinds of code, from Assembler to 3GL and object-oriented to scripting languages and XML.

The benefits of DSM are readily available when the DSM solution—the language and generator—is available. Usually that is not the case as we need to develop the DSM solution first. In Part IV we show how to define modeling languages and code generators for application domains. These guidelines are based on our experience of building model-driven development with DSM in multiple different domains generating different target code for different target environments.

1.4 WHAT IS DSM?

Domain-Specific Modeling mainly aims to do two things. First, raise the level of abstraction beyond programming by specifying the solution in a language that directly uses concepts and rules from a specific problem domain. Second, generate final products in a chosen programming language or other form from these high-level specifications. Usually the code generation is further supported by framework code that provides the common atomic implementations for the applications within the domain. The more extensive automation of application development is possible because the modeling language, code generator, and framework code need fit the requirements of a narrow application domain. In other words, they are domain-specific and are fully under the control of their users.

1.4.1 Higher Levels of Abstraction

Abstractions are extremely relevant for software development. Throughout the history of software development, raising the level of abstraction has been the cause of the largest leaps in developer productivity. The most recent example was the move from Assembler to Third Generation Languages (3GLs), which happened decades ago. As we all know, 3GLs such as FORTRAN and C gave developers much more expressive power than Assembler and in a much easier-to-understand format, yet compilers could automatically translate them into Assembler. According

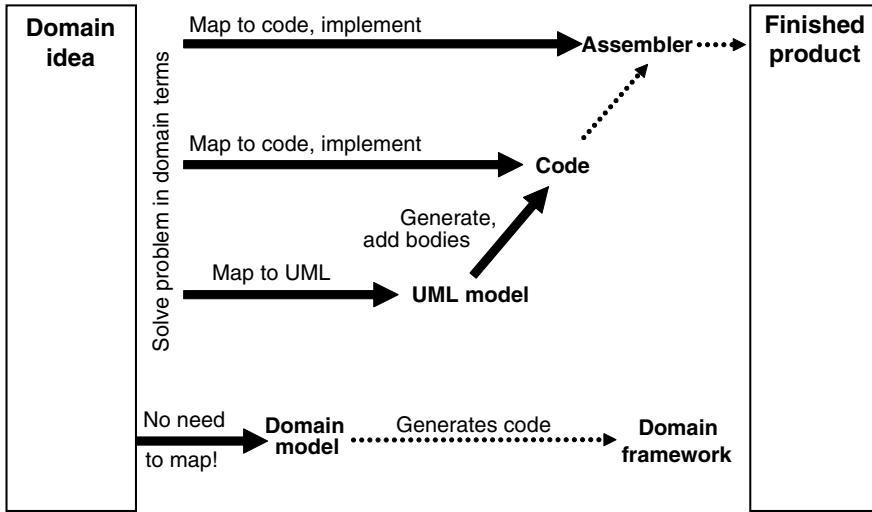


FIGURE 1.7 Bridging the abstraction gap of an idea in domain terms and its implementation

to Capers Jones' Software Productivity Research (SPR, 2006), 3GLs increased developer productivity by an astonishing 450%. In contrast, the later introduction of object-oriented languages did not raise the abstraction level much further. For example, the same research suggests that Java allows developers to be only 20% more productive than BASIC. Since the figures for C++ and C# do not differ much from Java, the use of newer programming languages can hardly be justified by claims of improved productivity.

If raising the level of abstraction reduces complexity, then we need to ask ourselves how we can raise it further. Figure 1.7 shows how developers at different times have bridged the abstraction gap between an idea in domain terms and its implementation.

The first step in developing any software is always to think of a solution in terms that relate to the problem domain—a solution on a high abstraction level (step one). An example here would be deciding whether we should first ask for a person name or for a payment method while registering to a conference. Having found a solution, we would then map that to a specification in some language (step two). With traditional programming, here the developers map domain concepts to coding concepts: “wait for choice” maps to a while loop in code. With UML or other general-purpose modeling languages, developers map the problem domain solution to the specification with the modeling language: like “wait for choice” triggers an action in activity diagram. Step three then implements the full solution: giving the right condition and code content for the loop code. However, if general-purpose modeling languages are used, there is an extra mapping from a model to code. It is most remarkable that developers still have to perform step one without any tool

support, especially when we know that mistakes in this phase of development are the most costly ones to solve. Most of us will also argue that finding the right solution on this level is exactly what has been the most complex.

1.4.2 Automation with Generators

While making a design before starting implementation makes a lot of sense, most companies want more from the models than just throwaway specification or documentation that often does not reflect what is actually built. UML and other code-level modeling languages often just add an extra stepping stone on the way to the finished product. Automatically generating code from the UML designs (automating step three) would remove the duplicate work, but this is where UML generally falls short. In practice, it is possible to generate only very little usable code from UML models.

Rather than having extra stepping stones and requiring developers to master the problem domain, UML, and coding, a better situation would allow developers to specify applications in terms they already know and use and then have generators take those specifications and produce the same kind of code that developers used to write by hand. This would raise the abstraction level significantly, moving away from programming with bits and bytes, attributes, and return values and toward the concepts and rules of the problem domain in which developers are working. This new “programming” language then essentially merges steps one and two and completely automates step three. That raised abstraction level coupled with automatically-generated code is the goal of Domain-Specific Modeling.

DSM does not expect that all code can be generated from models, but anything that is modeled from the modelers’ perspective, generates complete finished code. This completeness of the transformation has been the cornerstone of automation and raising abstraction in the past. In DSM, the generated code is functional, readable, and efficient—ideally looking like code handwritten by the experienced developer who defined the generator. Here DSM differs from earlier CASE and UML tools: the generator is written by a company’s own expert developer who has written several applications in that domain. The code is thus just like the best in-house code at that particular company rather than the one-size-fits-all code produced by a generator supplied by a modeling tool vendor.

The generated code is usually supported by purpose-built framework code as well as by existing platforms, libraries, components, and other legacy code. Their use is dependent on the generation needs, and later in this book (Part III), we illustrate DSM cases that apply and integrate to existing code differently. Some cases don’t use any additional support other than having a generator.

At this point, we need to emphasize that code generation is not restricted to any particular programming language or paradigm: the generation target can be, for instance, an object-oriented as well as a structural or functional programming language. It can be a traditional programming language, a scripting language, data definitions, or a configuration file.

1.4.3 DSM Solution Evolves

Changes to the DSM language and generators are more the norm than an exception. A DSM solution should never be considered ready unless all the applications for that domain are already known. The DSM solution needs to be changed because the domain itself and related requirements change over time. Usually this leads to changes in the modeling language and related generators. If a change occurs only on the implementation side, like a new version of the programming language to be generated or using a new library, changes to just the code generators can be adequate. This keeps the design models untouched and hides implementation details from developers using DSM.

A DSM solution also needs to be updated because your understanding of a domain, even if you are an expert in it, will improve while defining languages and generators for it. Even after your language is used, your understanding of your domain will improve through modeling or from getting feedback from others that model with the language you defined. Partly you will understand the domain better and partly you will see possible improvements for your language.

1.5 WHEN TO USE DSM?

Languages and tools that are made to solve the particular task that we are working with always perform better than general-purpose ones. Therefore DSM solutions should be applied whenever it is possible. DSM is not a solution for every development situation though. We need to know what we are doing before we can automate it. A DSM solution is therefore implausible when building an application or a feature unlike anything developed earlier. It is something unique that we don't know about. In such a situation we usually can only make prototypes and mock-up applications and follow the trial-and-error method, hopefully in small, agile, and iterative steps.

In reality, we don't often face such unique development situations. It is much more likely that after coding some features we start to find similarities in the code and patterns that seem to repeat. In such situations, developers usually agree that it does not make sense to write all code character by character. For most developers, it would then make sense to focus on just the unique functionality, the differences between the various features and products, rather than wasting time and effort reimplementing similar functionality again and again. Avoiding reinventing the wheel is good advice for a single developer, but even more so if colleagues are implementing almost identical code too.

In code-driven development, patterns can evolve into libraries, reusable components, and services to be used. Building a DSM solution requires a similar mindset as it offers a way to find a balance between writing the code manually and generating it. How the actual decision is made differs between application domains. In Part III, we describe five DSM cases where the partitioning is done in various ways, and in Part IV, we give guidelines on how you can do it.

Using resources to build a DSM solution implies that development work is conducted over a longer period within the same domain. DSM is therefore a less likely option for companies that are working in short term projects without knowing which kind of application domain the next customer has. Similarly, it is less suitable for generalist consultancy companies and for those having their core competence in a particular programming language rather than a problem domain.

Although the time to implement a DSM solution can be short, from a few weeks to months, the expected time to benefit from it can decrease the investment interest. The longer a company can predict to be working in the same domain, the more likely it will be interested in developing a DSM solution. Some typical cases for DSM are companies having a product line, making similar kinds of products, or building applications on top of a common library or platform. For product lines, a typical case of using domain-specific languages (e.g., Weiss and Lai, 1999) is to focus on specifying just variation: how products are different. The commonalities are then provided by the underlying framework. For companies making applications on top of a platform, DSM works well as it allows having languages that hide the details of the libraries and APIs by raising the level of abstraction on which the applications are built. Application developers can model the applications using these high level concepts and generate working code that takes the best advantage of the platform and its services. DSM is also suitable for situations where domain experts, who often can be nonprogrammers, can make complete specifications using their own terminology and run generators to produce the application code. This capability to support domain experts' concepts makes DSM applicable for end-user programming too.

1.6 SUMMARY

Domain-Specific Modeling fundamentally raises the level of abstraction while at the same time narrowing down the design space, often to a single range of products for a single company. With a DSM language, the problem is solved only once by visually modeling the solution using only familiar domain concepts. The final products are then automatically generated from these high-level specifications with domain-specific code generators. With DSM, there is no longer any need to make error-prone mappings from domain concepts to design concepts and on to programming language concepts. In this sense, DSM follows the same recipe that made programming languages successful in the past: offer a higher level of abstraction and make an automated mapping from the higher level concepts to the lower-level concepts known and used earlier. Today, DSM provides a way for continuing to raise the description of software to more abstract levels. These higher abstractions are based not on current coding concepts or on general-purpose concepts but on concepts that are specific to each application domain.

In the vast majority of development cases general-purpose modeling languages like UML cannot enable model-driven development, since the core models are at substantially the same level of abstraction as the programming languages

supported. The benefits of visual modeling are offset by the resources used in keeping all models and code synchronized with only semiautomatic support. In practice, part of the code structure is duplicated in the static models, and the rest of the design—user view, dynamics, behavior, interaction, and so on—and the code are maintained manually.

Domain-specific languages always work better than general-purpose languages. The real question is: does your domain already have such languages available or do you need to define them? This book aims to answer the latter: it guides you in defining DSM for your problem domain and introducing it into your organization.

CHAPTER 2

BUSINESS VALUE

A new approach should not be used just for the sake of technology but because of the value it gives. In this chapter, we describe the key benefits Domain-Specific Modeling (DSM) offers for companies. These benefits are common to other ways of moving toward higher levels of abstraction: improved productivity, better product quality, hiding complexity, and leveraging expertise. We also address the economics of DSM: return on investment and the ownership of a DSM solution.

2.1 PRODUCTIVITY

A higher level of abstraction generally leads to better productivity. This covers not only the time and resources needed to make the product in the first place but also its maintenance. The productivity increase is usually so significant that companies don't want to make their DSM solution public as it has become key to their competitive advantage. This is especially true for vertical DSM solutions. They are also inherently less known because their domains are narrower and the applicable practitioner community is smaller.

The exact productivity gain, however, is often difficult to measure. Only in research settings, something that a software development company usually can't afford to do, is there the possibility of having controlled tests over time. As with any case study where significant benefits have been achieved, reporting them in an educational way is faced with two difficulties: the danger of sounding triumphalist when providing

evidence from case study participants and the problem of providing enough detail without revealing trade secrets. Changes in productivity, however, are very visible since they can be detected in the daily work of developers, especially the change from using code concepts to using domain concepts, and automatically producing the code can be drastic: For example, a development task that earlier took days can be done with DSM in minutes. Consider the difference in using UML and DSM in the mobile phone application described in Chapter 1. Because the difference is so clear, development organizations normally have little interest in conducting further comparisons to calculate return on investment for the creation of a DSM solution. Later, once the DSM is in place, the organization may “forget” the older practices and there are no longer clear yardsticks available to compare productivity. Also, if the DSM solution is used by nontechnical people, often called domain experts, the automatic production of code can be taken as a self-evident fact. Chapter 6 reports such a case in more detail. Next we discuss productivity gains as reported in some public industry cases.

2.1.1 Productivity Improvement in Development Time

Domain-specific approaches are reported to be on average 300–1000% more productive than general-purpose modeling languages or manual coding practices (Kieburtz et al., 1996; Ripper, 1999; Weiss and Lai, 1999; Kelly and Tolvanen, 2000). Productivity improvements are usually also emphasized by industry analysts. For example, the Cutter Consortium report emphasizes the productivity gains over general-purpose modeling languages (Rosen, 2006) and the Burton Group sees the use of domain-specific languages and custom metamodels to be the greatest aid to productivity (Haddad, 2004). These productivity improvements are considerable when compared to the use of general-purpose modeling languages. For example, if we inspect UML experiences, the best productivity figures we found were from a vendor-sponsored MDA study showing a 35% productivity increase (Compuware, 2003). There are not many studies available on UML use that measure the productivity gains: surprising since UML is widely known. There are, however, empirical studies that show no gains or even productivity decreases. A study by the Modelware project (Dubinsky et al., 2006) in five different companies did not find any significant difference between UML-based modeling and traditional manual approaches. Some other studies, like Bettin, 2002, show that productivity can also decrease when moving from manual coding to standard UML. Next let’s describe three cases based on the domain-specific approach in more detail.

Mobile Phone Applications at Nokia Nokia has reported a 1000% increase in productivity when compared to earlier manual practices (Narraway, 1998; Kelly and Tolvanen, 2000). This case deals with having a DSM solution to develop applications for mobile phones. At this point, we should note that the DSM solution applied at Nokia is not the same as reported in this book. The DSM example demonstrated earlier and to be described in more detail in Chapter 8 is targeting relatively simple administrative applications that are developed into already shipped phones.

The case of Nokia is interesting for DSM as mobile computing is not a static area but is constantly changing with different product requirements and product features. Mobile phones and mobile computing are some of the fastest growing markets today. In this kind of development environment, where releasing a product a month before a competitor translates to millions of dollars, the productivity of the development process is vital. General-purpose methods were not seen as useful in this kind of development environment: “UML and other methods say nothing about mobile phones. We were looking for more,” said a Project Manager. Nokia also detected that existing languages and generators assumed “clean-sheet” implementations, whereas the majority of Nokia’s projects built on existing previous work. In this case, the DSM solution was built on top of existing architectures. Tooling was based on the available metamodeling tools and code generators (MetaCase, 2000).

The DSM solution was developed especially to improve developer productivity. The productivity improvements were to be achieved by applying the following strategies:

- Working at higher abstraction levels so that designers don’t have to know everything. They can focus on designs rather than on how to implement them in code.
- Linking designs to code generators so that designers are effectively “writing code” as they design. This worked very well in practice as modelers did not usually realize the mappings from designs to code.
- Underpinning the development process with a tool effective enough that no one will want to develop outside the tool.

Comparing the productivity was somewhat straightforward as Nokia had already been developing mobile phones and their applications and had internal metrics available. A point of comparison for the study was developing modules from scratch to a finished product. It therefore also included phases after generating code, such as testing and quality assurance. The comparison was based on developing applications that were similar to applications and functionalities made into earlier products. As mobile applications were increasing considerably in size and complexity, we may well expect that the applications developed with DSM were no simpler than earlier ones, and most likely had more functionality.

Military Command and Control Systems at USAF In the U.S. Air Force (USAF) domain-specific languages with code generators have been compared against best-practice code components (Kieburtz et al., 1996). This comparison, covering both initial development and maintenance tasks, detected productivity improvements in the ratio 3:1. The study is reported in an academic conference and has a carefully planned and followed research method. Other researchers can therefore repeat the study in their application domain. Experiments to compare two alternative development technologies have been conducted in relation to message specification in the Air Force C3I systems. For the message translation and validation domain, a

dedicated domain-specific language and generator were developed and compared against currently used manual practices using ADA.

In the study, the researchers analyzed the experiments statistically by comparing the variances. More than 130 development tasks, including both initial development and maintenance, were studied. The study was well planned having a control project making the same functionality with traditional manual practices. It highlighted the productivity gains of the domain-specific approach over manual practices, showing a 300% increase in productivity. Regardless of the tasks, developers consistently expended less effort to accomplish tasks with domain-specific languages and generators than with manual practices. The differences in the average performance of the subjects are statistically significant at confidence levels exceeding 99%.

Although the system developed was not a product line like the mobile phones discussed earlier, the researchers found that the domain-specific approach gives “superior flexibility in handling a greater range of specifications” than with code components. In other words, specifications are easier to handle as they operated at a higher level of abstraction than the implementation code. Such capability is especially useful when handling larger systems and in maintenance tasks.

Product Families at Lucent Lucent has developed several domain-specific languages to automate software development. The main objective for creating domain-specific tooling has been the fundamental improvements in productivity: between 300 and 1000% (Weiss and Lai, 1999). The main focus was on supporting product family development, where a large amount of functionality is shared between several products. The languages let developers specify only those aspects that differ from other family members; common aspects shared among all the family members did not need to be specified. The FAST process, discussed in detail by Weiss and Lai, guided in this domain engineering process to identify expected variation needs.

The experiences reported from Lucent are based on systems ranging in size from small laboratory-type systems to large industrial applications, like the 5ESS phone switch. Domain-specific modeling languages and related transformations were seen as particularly useful in 5ESS as the life span of phone switches is several decades. To support domain-specific modeling languages and transformations from them Lucent developed several in-house tools, some based on textual languages and others on graphical languages. In Lucent’s experience the graphical ones tended to be more successful.

The benefits gained from domain-specific methods at Lucent are as follows:

- Productivity improvement of about 3–10 times. In the beginning, the productivity gains were smaller, factors of 3 to 5, but in areas with experienced developers, the productivity gains were typically a factor of 10 or more.
- Decreased product development costs: In the product family domain, the cost-saving aspect was seen as significant and clearly measurable. Developing several similar kinds of products, that is, family members, data collection, and comparison became easier to do than in one-shot development.

- Shorter intervals between product releases: The improvement ratios were highly dependent on the domain, but most projects experienced an interval reduction of four to one.

These cases report fundamental productivity improvements via domain-specific languages and use of code generators. In internal discussions with companies building their own in-house DSM solutions, even bigger productivity increases are claimed. It is not exceptional that the expert developers who created the DSM solution report even 100 times faster development but this performance usually decreases when introduced to the larger organization.

2.1.2 Productivity Changes After Initial Introduction

Although when a DSM solution is introduced the productivity difference is most visible compared to earlier approaches, possible productivity gains are not limited to its introduction.

The Important Learning Aspect Usually productivity studies comparing traditional and new approaches don't recognize the learning aspect: once modelers get acquainted with the new approach as well as with the earlier practices, the difference can be expected to become even bigger. This is especially important for analyzing DSM cases, as the use of modeling for complete code generation is new for many developers. For example, in the DSM case discussed in Chapter 6 dealing with the insurance domain, the company estimated its productivity increase as 300% compared to earlier manual programming practices. This estimation was done after specifying the first 30 insurance products out of more than 200. Since the software modeling was new to almost all of the developers, it was expected that productivity would continue to increase later too. Unfortunately the company did not collect data from these later phases.

Productivity in Maintenance Phases Often most development effort and time go into the maintenance phase. There are roughly three main kinds of maintenance work: bug corrections, responses to new or changed requirements, and responses to platform or environment changes. Since code is automatically generated, many common types of bugs are avoided in DSM. Bugs in the models and responses to new or changed requirements are handled by changing the models, with the same productivity gains compared to hand coding as when creating the initial models.

If there are changes in the underlying platform or environment, then this normally has an effect on the code that needs to be generated and possibly on the modeling language. In this case, only the experienced developers who defined the DSM solution need to make changes, and other developers' models and code will be updated automatically. For instance, a Java generator can be changed to create MIDP Java or a new generator can be made to create C code from the exact same models. The key is that just one or a few experts define the DSM solution. This is unlike traditional programming, where every developer must master the domain and then manually map

their design decisions to code. As we well know, some do this well and some not so well. Thus the person who does it well should define the DSM automation, and others just use it. This is nothing new, as we treat other development tools, for example, compilers, in a similar way.

2.1.3 Gains from Improved Productivity

Improved productivity offers several benefits for the companies. Depending on the company, market situation, and competition, the advantages can be leveraged in different ways:

- Shorter time to market. Companies can develop products faster and bring them to the market before their competitors. The role of increased use of automation is key here as just adding human resources (Brooks, 1975) does not cut delivery times but can even prolong it.
- Faster customer feedback loops. Especially companies that develop products in close collaboration with their customers benefit from DSM as it enables a faster feedback loop. The automation can be used first to produce a prototype application for requirements approval and later to generate the application in the real target. This is typical in embedded software where it is relatively easy to generate prototypes, for example, running in a PC, which customers and other stakeholders can check and review before final implementation in the real target.
- Lower development costs. Cost reduction is a natural consequence of improved productivity. Where to allocate the saved resources is a management decision: making new products and features, improving the quality and process, or decreasing the number of developers from subcontractors or in-house.
- Allow later changes. DSM may provide agility since the models are closer to the requirements than the source code is. Changes in customer requirements can be expressed in domain terminology directly in models and the new implementation can be generated. The difference to manual programming can be large since changing a few elements in a model can map to huge sections of the application. The advantage of making quick changes can also be realized through generators, as some changes can be done just by changing the generator. A classic example is generating the code into a different target environment or even programming language.
- New customer segments can be addressed. The capability to develop cost-effectively new variants that address requirements of different customer segments is one of the key reasons product family or product line engineering companies apply automation. Without automation, making and maintaining products manually would simply be too costly.
- Reduced need for outsourcing. With DSM, the key development work can be kept in-house instead of outsourcing. Usually this key knowledge is needed to build the DSM solution. Outsourcing in general is still possible but it means

DSM use rather than its creation. As DSM hides the details, the number of potential developers is larger. This gives better possibilities to find development resources.

2.2 QUALITY

DSM leads to better quality applications, mainly because of two reasons. First, modeling languages can include correctness rules of the domain, making it difficult, and often impossible, to create illegal or unwanted specifications. Elimination of bugs in the beginning is far better than finding and correcting them later. Finding corrections is usually easier and cheaper when done earlier. Models can also be checked and analyzed to detect errors or find specifications that lead to poor performance in the product build. Second, generators along with framework code provide a mapping to a lower abstraction level, normally code, and the generated result does not need to be edited afterwards. By automating the mapping from a problem domain to a solution domain we also reduce the risk of the implementation not corresponding to the solution specified in the problem domain.

As with the productivity gains, the improvements in quality do not come automatically. Someone must define the modeling languages, generators, and supporting frameworks. In DSM, a key reason for the better quality is that a large part of all applications is effectively built by the more experienced developers—those who created the DSM solution. We next describe in more detail how DSM contributes to improving quality.

2.2.1 Measuring Quality Improvements

Quality can be measured in many ways, such as the number of errors found, how reliable the product is, how much maintenance effort is needed, or how fast the application performs the required operations. Perhaps the best way to analyze quality is to compare the number of errors in the finished product. If errors can be decreased with DSM, we can expect that the quality should be better too. However, often companies don't have the time and resources to conduct carefully planned comparisons as these need to be objective, have representative applications or features, and aim for complete data collection. Instead the changes in quality are based on observations made while conducting individual design tasks and on gut feelings. For example, in a company specifying insurance products (described in Chapter 6) with DSM, the CTO recognized fewer errors but could not measure exactly the reduction ratio. Similarly at EADS (MetaCase, 2006), the Chief Engineer stated that with DSM “the quality of the generated code is clearly better, simply because the modeling language was designed to fit our terminal architecture. If applications were coded by hand there would be a lot of errors and the range of error types would be wider.”

While the studies performed in-house are not usually available, academic studies could help. Unfortunately, there is a lack of research examining quality influences

when moving to DSM. The USAF study reported by Kieburtz et al. (1996) is a welcome exception: It also compared the reliability of the software built. This was done by comparing the number of failed tests with manual approaches and with domain-specific languages and generators. The study found significantly fewer errors in the domain-specific approach. Although the generator was newly created for the task, the acceptance tests found 50% fewer errors from systems built using the domain-specific approach than from systems built manually. The differences in the average performance of the subjects were statistically measured and found significant at a confidence level of 97%.

2.2.2 Mapping to Requirements and Problem Domain

By providing support for problem domain concepts, DSM extends the role of languages from traditional design and implementation use towards requirements capture. In many cases, the domain-specific models express things that in traditional approaches would be close to the requirements a customer specified. Consider here again the mobile application example from Chapter 1. The requirements on application functionality, type of widgets to be used, and user navigation are directly expressed in the domain-specific model (Fig. 1.6). This close mapping from models to requirements enables customers and other domain experts to participate in development work.

Participation with Customers and Domain Experts Specifications made with domain terms are usually easier to read, understand, remember, validate, and communicate with. Although you might fully master the UML models shown in Chapter 1, the specification made with the domain-specific language may better explain what the application does. This closer alignment with the problem domain makes the models more usable right from the requirement specification phase. Customers and domain experts can now more easily participate in the development and validate the models at the specification phase. They can understand the application functionality much better from the domain-specific models than, for example, from UML.

The mapping to the problem domain is so close in Fig. 1.6 that the modeling language could almost be given to phone users to specify their own applications. In practice, in this domain only a limited percentage of phone owners could use the language for product development. However, the case of insurance products described in Chapter 6 illustrates a DSM language that is used directly by domain experts, with no programming background, to specify applications.

DSM Support for Early Validation In addition to simply reading the models to validate them, DSM may improve quality if concept demonstrations and early prototypes can be created from models. Customers and domain experts can then create partial models or use modeling languages specifically made for specifying requirements. Ideally, those parts of the requirements can be executed immediately after having specified them into a model.

Later, the same models can be extended by application developers or they can use different domain-specific languages but now have requirements that can be partially or completely executed. Naturally, if the execution is complete, further application development is not needed, but at least in developing embedded applications it is usual for prototypes specifying and illustrating requirements to be implemented in a different target environment than the actual product. This is simply done to speed up the feedback loop and have functionality fixed earlier. For example, the generated prototype for a car infotainment system could be some Java code running in a browser to highlight functionality, whereas the production code needs to be integrated in the particular car infotainment platform.

2.2.3 Testing with DSM

Rather than handling application quality by conducting testing activities on the code, DSM addresses errors earlier—at the modeling stage when specifications are created. This is relevant, as the earlier we can detect errors, the cheaper their correction becomes. Obviously, DSM does not make testing obsolete: functional testing is still needed but developers now have more time for this since many typical tests are no longer needed. This is very relevant for testing activities since it is not uncommon for an application developer to spend more than half of his day in testing: specifying tests, executing them, and making corrections.

Testing is Done by Experienced Developers The experienced developers who created the DSM solution have already done some testing. The errors prevented by the rules and constraints of the modeling language don't need to be tested again. In a similar vein, the automated generation of code reduces the testing needs of developers. We discuss the testing of DSM later in Part IV.

Naturally, all errors don't disappear with DSM, but their prevention and correction is left to the more experienced developers in the company. The nature of the errors is also different from that of manual practices. In DSM, they tend to be more consistent rather than random as in manually written code. In DSM when an application developer finds an error, he can most likely correct it by changing the model. If an error is found in the generated code, the correction should be made in the generator instead of correcting just the particular application under development. This allows other application developers and other design models to benefit from the corrections.

What is Tested? The kinds of testing targets that are usually handled manually can be partly covered automatically by DSM.

Application Architecture. Architectural rules will be enforced automatically at the design stage. This is unlike in manual practices where individual developers can make different choices and implement the software without following the planned architecture. For example, in mobile phones, an architectural rule related to the user interface is that there can only be one menu per form element. In DSM, this can be directly enforced by simply not allowing the modeler to add more than

one menu. Later, this architectural rule doesn't need to be tested anymore: all applications generated from the DSM models will follow it. This greatly eases the development process and makes it more predictable. Further benefits of this are available in the maintenance phase as we now know how the applications are implemented.

Coding. With generators many typical errors, like missing references, typos, or forgotten initialization, simply don't exist anymore for application developers. A generator and supporting domain framework may also prevent the creation of incorrect or otherwise unwanted applications, like those leading to poor performance or not following proven practices. Since the generator produces the code similarly for each case, the maintenance becomes easier: All developers implement the code similarly and there is no need to inspect and modify code implemented in different styles.

Reuse. The quality of the created applications can be expected to improve since reuse can be better enforced across the company (Griss and Wentzel, 1994). The DSM solution can prevent developers from making designs that reinvent the wheel or create an alternative implementation of the same functionality. These are very common in manually written code and their testing and maintenance can be time consuming and error prone. DSM can enforce reuse by automatically using design information from available models or referring to an existing library or platform code.

Validation at the Domain Level. A DSM language may also be designed to support the validation of desirable properties at the domain level. Validation is usually a lot simpler at the domain level than verifying code in a general-purpose modeling language or via a general-purpose programming language. Let's consider here a desirable property of mobile applications: that all selection dialogs must offer the possibility to cancel. With DSM, this property is built in to the modeling language: if no explicit cancel navigation is defined, the generator provides the default behavior.

Influence of Changes and Refactoring. Studies comparing DSM to manual practices (for example, Kieburtz et al., 1996) suggest that changes to the specifications are easier to do at the domain-specific level than at the code level. DSM expresses specifications with domain concepts without going into non-domain-related issues.

Documentation. Accurate and up-to-date documentation is often seen as a characteristic of quality in the development process. In DSM, the availability of good documentation can be guaranteed in the same manner as good code: documents, review reports, packing, and deployment guides are generated from the same source as the code, improving accuracy, consistency and standard compliance.

2.3 LEVERAGE EXPERTISE

DSM brings change to the organizational structure by identifying two clearly separate roles: those creating the DSM solution and those using it. Traditionally, all developers work with the problem domain concepts and map them to the implementation concepts manually. And among developers, there are big differences. Some do it better, but many not so well. In DSM, the mapping is instead made only once and by experienced developers. Often the capability to collect and formalize the domain knowledge is seen as valuable per se—even if there would be no modeling languages or generators. The automated support for development work makes DSM a powerful way to leverage the abilities of expert developers in a team. Once an experienced developer defines domain rules into the language, he enforces the development guidelines and best practices for all other developers. Similarly a code generator, made again by the expert developer, produces applications with better quality than could be achieved by normal developers writing the code manually.

2.3.1 The Group of Potential Developers Becomes Larger

DSM expands the group of potential developers by raising the level of abstraction closer to the problem domain and by hiding the implementation details. All developers don't need to be experienced in the application area in the same way as expected earlier. For example, they don't need to master the code libraries, know the programming models, or even be particularly good at programming. Instead they need to master the problem domain and know what solutions the customers are seeking. This needs to be mastered regardless of whether DSM is used or not.

At one extreme, DSM may allow even domain experts to develop applications or parts of applications themselves. This boosts productivity directly by removing the need for programming assistance and manual mappings to implementation. For example, the DSM case discussed in Chapter 6 of specifying financial products demonstrates this kind of DSM solution. Development was given to the domain experts as they knew the insurance and financial product problem domain better than the programmers. This can also improve quality since the DSM solution reduces the risk of having code that does not correspond to the solution specified in the problem domain.

Domain experts may create models not to generate production code but to create other kinds of artifacts. For example, DSM can support the creation of concept designs, prototypes, or product simulations. A DSM solution can also be used by test engineers to design test specifications and generate the test applications based on the designs. Domain-specific modeling languages can also focus on architecture design (like AUTOSAR in the automotive industry) specifying components, their interfaces, communication, and distribution of functionalities in a specific domain. Here the generation possibilities mainly produce interface definitions but can also address architecture analysis and simulation.

Complexity hiding naturally also supports technical developers, programmers, helping less experienced developers develop applications effectively. For example, in the controlled study of the USAF (Kieburtz et al., 1996), all developers perceived the

domain-specific approach as easier and felt completion of development tasks was more reliable. Similarly at EADS (Raunio, 2007), even new developers with less coding experience found themselves able to effectively develop application features, since the DSM solution supported the rules of the domain, guided model-based design work and prevented developers from creating erroneous designs.

2.3.2 Less Training and Guidance Needed

With DSM, introduction of new developers becomes easier. Obviously developers who need to master just the problem domain become productive faster than those who also need to learn the implementation domain and the mappings between the domains. At Nokia, for instance, the domain knowledge contained in the modeling language produced a faster learning curve for new employees or transferred personnel. Earlier a new developer was able to make applications and become productive only after being exposed to the material for 6 months. With DSM, the time was reduced to 2 weeks (Narraway, 1998; Kelly and Tolvanen, 2000).

When introducing DSM to existing developers in a domain, we can expect that less training is needed as the concepts of the modeling language are usually already familiar. Instead of learning both the particular problem domain and a solution domain along with effective mappings between them, in DSM developers ideally need to learn just the problem domain. This minimizes the need for learning in the first place and later reduces the quality assurance team's work to check that the development guidelines are correctly followed.

Learning How to Model Today when most developers are more experienced with coding than using formal modeling languages, the most fundamental change can be learning how to model. First, they need to learn that they can trust the models just as they trusted the code. Later, efficient model creation, modification, and reuse practices must follow. Diagrammatic languages, matrix representation, or tables offer representations other than linear text. We discuss the difference of modeling with DSM versus traditional manual coding in Chapter 3.

To introduce DSM, it is especially relevant to know how developers, those doing the modeling, react when the nature of development work changes from manual coding to modeling and generating the code. This topic is a less studied area, but we can expect that, like all new technology that changes working practices, attitudes toward DSM can vary from highly accepted to rejected. In Ruuska (2001), 17 developers in a telecom company, all working with the same DSM solution, were interviewed to examine their attitudes toward changed working practices. Every developer interviewed had experience with traditional manual development practices. Out of the 17, only two developers preferred the earlier manual coding practices to DSM. Interestingly, those two developers preferring manual coding were still using DSM although the manual coding approach was also available. Nine of the 17 developers found DSM better than earlier manual practices and the remaining six developers could not say their preference.

There can be multiple reasons for negative attitudes, but generally, those losing their position as “top coders” may see DSM as a threat. With DSM, less experienced developers can produce code that is as good as, if not better than, code written manually by more experienced developers. This change must be taken into account when introducing DSM. Most likely the same kind of situation exists as when the need for assembly programming was reduced while introducing languages like Fortran and Basic. In DSM, there is still a need for developing components and supporting framework code along with generators, although most developers can use the domain-specific languages.

Learning the Domain and Modeling Languages Introduction of DSM requires that the modeling languages be learned. Learning them, however, can be easier than say learning UML since the domain concepts are known, or at least need to be known regardless of how the software is developed. For example, an empirical study in a telecom company (Ruuska, 2001) showed that a maximum of 2.5 days was needed to learn the DSM solution. The study showed that most developers (11 of 17) received less than a day’s training but were soon capable of creating nontrivial applications.

Preserving Domain Knowledge Although a DSM solution may seem to offer the possibility of developing applications without preserving further knowledge about the internal structure of the DSM solution, such a scenario is usually not viable. Sooner or later the domain changes and software must be modified or updated. This requires similar domain knowledge and expertise to the initial creation of the DSM solution. The changes are easier and faster to implement if knowledge of the DSM solution is already available and the tooling allows incremental changes. The persons maintaining the existing DSM solution, however, may be different from those who originally created it. In many cases, this is natural since a DSM solution can be applied over several decades. We describe the various roles in DSM creation in more detail in Section 4.6.1.

Although the creation of a DSM solution is often kept in-house, the situation with DSM use can be different. Since the complexity can be hidden, the development tasks supported by a DSM solution can be outsourced more easily and the potential group of developers is bigger. DSM can also make the outsourced development efforts more measurable and comparable since they are based on the same abstractions—those created by the company outsourcing. We should note here that a shared DSM solution cannot be changed separately by each of the companies using it. Actually, very seldom do companies want to share their languages and generators. The reason is simple. Your DSM solution shows more than the source code: It shows how you make the applications and what kind of automation you apply. Most importantly it shows which kind of applications you could develop in the future. This information is not visible from the models alone but is seen from the metamodel and partly from the generators. These show the possible combinations of model elements based on the metamodel. If we rely again on the mobile phone example from Chapter 1, the metamodel would reveal how many physical displays

the mobile phones can have, how many views individual applications can have, how many softkey buttons are available, and so on.

Introducing DSM concepts and tools to development teams is a process that involves a paradigm shift. Team members who have been working at a low level of abstraction, writing code, are now able to work at a higher level of abstraction and develop the models from which code, tests, documentation, and so on, are generated automatically. This change is nothing new: our industry made a similar shift when moving from assembly to third generation languages (3GLs). Such a change is not always easy to introduce. Application developers may also reject DSM as it changes their work too much and may prefer to use a traditional approach. For example, in the study of a telecom company 2 developers out of the 17 interviewed preferred using the old manual coding approach instead of DSM (Ruuska, 2001). These figures can vary highly, for example, depending on the organizational cultures and individual backgrounds. There is, however, a place for traditional manual coding work—areas which cannot be automated with DSM. The capabilities of the best programmers can be used there. In the DSM architecture, discussed later in Chapter 4, the more experienced programmers can focus on making the generators, framework code, or other components of the target environment.

2.4 THE ECONOMICS OF DSM

Domain-specific approaches should generally be used whenever possible. They normally produce better results than general-purpose approaches. Ready-to-use DSM solutions, however, are normally not available. Because of the differences in the problem domain, underlying architectures, target environments, and generators targeting different programming languages, etc., it is often the case that a company must consider building its own DSM solution, or at least modifying an available one. The decision to create a DSM solution is both technical and economical. While the remaining parts of this book mostly address the technical side, we focus next on the economics of DSM. We will begin by discussing the investment and its payback and then move on to ownership.

2.4.1 Use an Existing DSM Solution?

Often the use of existing DSM solutions is simply impossible because companies that created them want to keep them in-house. This is natural, given that often the potential users of a DSM solution are competitors working in the same domain and producing similar kinds of applications. Why would any organization want to help a competitor create better quality products faster? Making a DSM solution available for others also requires extra resources and knowledge, which is not initially available in a company focused on improving its internal software development efforts.

There are some situations, though, where companies want to make their internally developed DSM solutions publicly available. First, a company outsourcing its

application development efforts usually wants its subcontractors to work with its existing DSM solution and tools. While the DSM solution is shared, it is not usually made available for changes. A second well-known situation for publishing a DSM solution is when it makes using an underlying platform easier or more palatable for a larger number of developers. For example, in the embedded area, it is very common for hardware manufacturers to provide tooling to develop features on top of their target platform.

A third case covers initiatives, usually inside a vertical domain, that create standards in terms of metamodels. These are aimed at improving communication and information exchange and lowering the cost of entry to create competition. For example, in the healthcare area, initiatives like Health Level Seven (www.hl7.org) have created information models that represent clinical data and its various connections. In the automotive industry, AUTOSAR (www.autosar.org) defines several modeling languages that enable the specification of interfaces for a specific software platform. Zachariadis *et al.* (2006) suggest a metamodel for adaptable mobile systems and Sztipanovits (1998) describes a metamodel for signal processing. Further examples are data warehousing and its CDW metamodel (www.omg.org) and software-defined radio (www.sdrforum.org) in which several metamodels have been implemented (e.g., Bhanot *et al.*, 2005). Usually such vertical languages target data structures and interfaces and not necessarily behavioral and dynamic aspects of software, at least not in such detail that complete code generation can be achieved. By narrowing them further, for a (sub)domain in a company, higher automation via generators may be achieved. Finally, companies that sell their existing DSM solution, usually for a specific task in a given vertical domain, naturally want to make their DSM solutions publicly available. Some familiar tools here are Simulink, for digital signal processing and simulations, and Labview, with its G language, particularly suitable, for example, for developing instrument control systems.

If you are able to find a DSM solution that fits your domain, it is most likely a better choice than developing the same functionality manually with general-purpose modeling languages. However, although a DSM solution can be found for a given domain, it is not guaranteed that it can be readily applied. It can be based on somewhat different concepts, use different rules, and generate different programming languages or, at least, in a different way than is preferred. We have been involved in a number of mobile phone applications and found that, although the domain is clearly the same, the modeling language, generators, and underlying target environment are different. For example, in a phone the main user interface elements are called and specified differently by the companies. In some companies, a domain concept can be called a view whereas others call it a book or panel and give somewhat different characteristics for them. The code to be generated is seen to have even more significant differences than the modeling languages. A working DSM solution, if publicly available, would not therefore necessarily be suitable for any of the companies. For example, development of TETRA phones at EADS, once part of Nokia, could start by using the same target environment and DSM solution as Nokia's consumer phones used. Differences in the domain, and the divergent evolution of the markets for the created

products, could later require that EADS modify the DSM solution to satisfy its own requirements.

2.4.2 When to Invest in Building a DSM Solution?

The basic economic assumption is that investment in building a DSM solution is paid back later during its use. This payback can be calculated from many factors such as a faster development cycle, improved product quality, or being able to allocate work differently. Some of these factors are not always easy to calculate beforehand. What is the value of having up-to-date documentation? Or being able to reach new customer segments? Or having a product ready a few months earlier than your competitors? While there is certainly significant value in using DSM for these reasons, we will focus here solely on development costs that are typically already known or can be reliably estimated.

The basic decision for most companies is between two major alternatives: continue using a general-purpose approach or build a domain-specific one. Figure 2.1 illustrates these alternatives in relation to cumulative costs of application development. For the sake of simplicity, we will assume here that the development costs are linear.

The increased use of automation by DSM gives a potential payback. The *y* axis represents the cumulative costs of application development. The *x* axis represents the repetition—the frequency at which the DSM solution is used. The repetition can be measured in different ways, and therefore, the *x*-axis can mean many things:

- Number of product variants: How many applications or products can we build with the DSM solution? Obviously, the more products we are building, the more likely is the return on the investment in creating a DSM solution. Work in product lines suggests that if there are more than three variants then the effort to build automation pays off (Weiss and Lai, 1999).

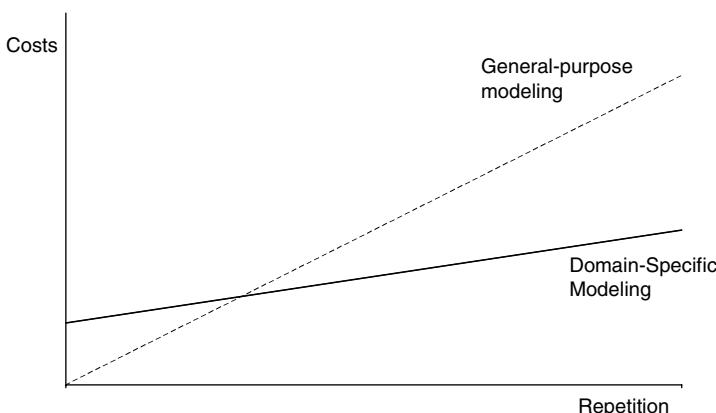


FIGURE 2.1 Comparative application development costs (cumulative)

- Number of product versions: DSM does not necessitate a product line. Repetition may occur also when there is a single product with multiple consecutive versions or configurations.
- Number of features within a product: Even inside a single application we may have several features that all share the same target environment and domain framework. Consider here a typical ERP system. Although a manufacturer usually builds only one system, its development requires a lot of repetitive development actions. There are usually thousands of similar kinds of data access operations, fields, forms, workflow actions, and common data elements. A DSM solution can then be made to automate their development. Repetition is thus not between the products, but inside a single product.
- Number of developers: Often companies have multiple developers for creating similar kinds of functionality. A typical example is creating customer-specific services based on a common target environment, such as a workflow engine or telecom services. The larger the number of application developers, the greater the benefits of having a DSM solution.

The payback point is reached when the cumulative costs of application development become smaller with DSM than with the current approach (as seen in the lines crossing in Fig. 2.1). The more we create variant products, similar features, or product versions, the faster the return on investment is achieved. For example, the experiences at Lucent discussed in Section 2.1.1 indicate that after the third product payback is reached. Furthermore, the empirical study conducted at USAF (Kieburtz et al., 1996) clearly proved that DSM makes sense even with a single product with repetitive development tasks for individual features. Similarly payback can be reached if there are multiple developers each benefiting from DSM. For example, at Nokia where the number of developers for mobile applications is relatively large, counted in three figures (Narraway, 1998), the DSM solution can benefit many developers. In contrast, if we have just one developer and we are creating a product only once, automation is unlikely to prove beneficial.

Repetition, however, is not the only factor when estimating value. With a DSM solution, less experienced developers can create applications or the work can be allocated differently than with traditional manual practices. This is possible because the technical details can be hidden from application developers. In some cases, the development work can be radically reallocated by giving it to domain experts, who do not necessarily have programming experience.

The Cost of Building a DSM Solution The benefits of DSM are not free; someone must create the DSM solution. This is the initial investment. In Fig. 2.1, the cost of using DSM is therefore larger at the beginning than when continuing with current practice, which does not require any additional investment.

The initial building cost for DSM is usually comprised almost entirely of human resource costs, namely, domain experts and experienced developers. Earlier, the cost of tool development was also relatively high since companies needed to

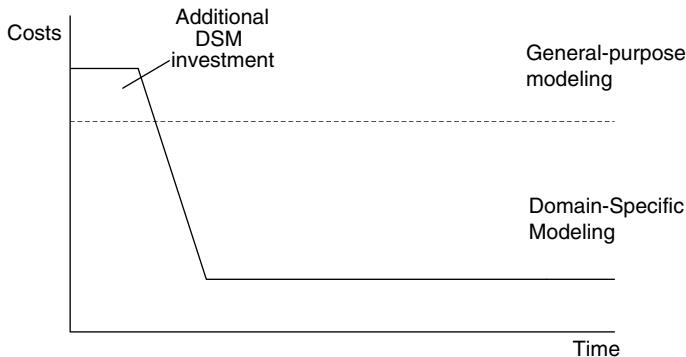


FIGURE 2.2 Comparative costs including DSM investment costs

build tools from scratch or by using supporting libraries and frameworks. For example, implementation of a UML-based modeling tool with Eclipse and related frameworks is calculated to have taken over 20 man-years to complete (Ströbele, 2005). Most companies simply don't have such resources. Today, such resource-intensive approaches have been replaced by higher-level and more automated solutions for building tool support for DSM. We will inspect the tooling side further in Chapter 14.

In practice, the initial costs are not large since most of the developers do not participate at all in building a DSM solution. They may continue application development with current practices until the DSM solution is available. Figure 2.2 shows an alternative way to look at the development costs: comparative costs including DSM initial investment costs.

After the initial introduction phase, the development costs decrease significantly thanks to improved productivity, fewer errors, being able to get feedback earlier, and so on. Although the development costs can decrease immediately, in practice, companies rarely follow the big-bang approach. A more common scenario is that the company first pilots the DSM solution and then gradually introduces it into the organization—usually first in some domains or subdomains. We describe the guidelines for the DSM definition process in Chapter 13.

As only a few people participate in the DSM creation, the costs are usually very small compared to overall costs. The largest investment is usually in allocating domain experts and experienced developers for the project—people who are already key resources. Typically a single individual or a very small group is responsible for creating the modeling language and generators. Depending on the domain, there can then be more developers working with the components of the target environment. It is not exceptional to have just one or two experienced developers in the DSM creation project and then tens or hundreds in the team applying DSM.

Maintenance and Upgrading Costs Later, once the DSM solution is in use, additional costs occur for its maintenance. These costs are relative to the changes made in the DSM solution. If the domain is static and the company continues using the

original DSM solution unchanged—which is usually rare—the maintenance costs are minimal. It is more likely that the domain changes or the company finds better ways to automate application development with DSM. These changes are again potentially paid back once introduced to the rest of the application developers. As experience in creating DSM solutions grows, we can expect that the maintenance cost will decrease over time. The knowledge gained is utilized when creating a new DSM solution for another domain.

The maintenance costs are not normally fixed (linear, as in Fig. 2.2) but change over time so that a few developers focus for a certain period of time on updating the DSM solution. At other times they may be part of the team using the DSM solution. It is worth noting that the amount of expert resources needed to build and maintain a DSM solution does not grow with the size of products developed in the domain or with the number of modelers.

Waiting Cost Figure 2.2 shows a simplified version of the costs, since DSM development is only one part of the overall costs. We need to consider also the cost of waiting: the benefits lost because the DSM solution was not yet available. If you have decided to improve development productivity, what is the value of being able to start next month versus next year? Let's assume that you have 10 developers and that each application or feature in an application takes 6 man-months. Then you allocate one developer from the application development team to build a DSM solution for 4 months. Of course, the other developers would not sit idle, waiting for the DSM solution; they continue building applications in the old way. After 4 months your nine developers would have completed six applications compared to nearly seven—hardly noticeable. Then you introduce the new DSM solution and assume that each application can now be made in 1 month. After month 5 you then have 15 applications ready. We assume here that one developer would still be improving the DSM solution. After month 6 you have 25 applications ready compared to the 10 you would have had without DSM. Getting those same 25 applications without DSM would have taken until the end of month 25! To decide whether to invest in DSM, you can calculate your own costs of application development to see the difference.

Usually, the cost of waiting is much larger than the original cost of building a DSM solution and introducing it. If the target environment is ready, the cost of defining modeling languages and generators is usually a matter of man-weeks and in worst cases not longer than a few man-months. The development times are described in relation to practical examples in Part III, where we describe different DSM solutions and their development.

2.4.3 Ownership of Your DSM Solution

A DSM approach, like all automation, requires two different teams: one building the automation and one using it. While users get the benefits, it is also relevant to know who develops the DSM solution. Generally, it could be your own developer, an external consultant, or a tool vendor. The pros and cons of each are as follows.

Made In-House In-house development is often seen as the preferred choice simply because it guarantees control of the automation. How could anybody else know better than you how your software should be developed? This is especially the case with companies creating their products with a DSM solution. It would be hard to see how a company like Nokia or Lucent could outsource their key competence in software development. Having full control means not only competitive advantage but also the capability to focus on a narrow domain—to raise the level of abstraction in a particular domain. The cost of building a DSM solution in-house is the need for expertise and resources.

Usually, the best opportunity to define a DSM solution is within the organization that owns the target environment with which the DSM environment operates. The changes in the target environment can then be better integrated with the DSM solution. It is relevant to mention that this ownership does not necessarily mean complete ownership of the whole target environment but of the layer with which the generated code integrates. It is not an accident that the significant productivity increases reported in Section 2.1.1 are often found in companies having defined a DSM solution for their own platform.

Made by Consultants If there is limited time or expertise in the domain, external consultants may help. This is more typical in the initial phase of DSM creation and introduction than in its maintenance. It is even predicted that new value chains will be established on building the automation (Greenfield and Short, 1994). Since the DSM solution becomes one of the core competencies, today it is usually preferred to keep the maintenance in-house. Often the use of consultants only in the beginning is natural since they don't always have the detailed domain knowledge of your business.

Use of consultants is then limited to a few tasks, like training or organizational introduction. Perhaps the most typical participation is in developing supporting tools since tool development is a key competency in very few companies. In the past, the use of external consultants for tool development was more relevant since the cost of tool development was a major project, counted in man-years. Today metamodel-based tools have radically changed this. For example, the implementation of a basic modeling editor can be done in less than a man-day. In creating a DSM solution, most of the time is spent on finding a good abstraction for development work, not on creating tool support for it.

Provided by a Tool Vendor If you find a working DSM solution, you should start using it. This is the best option for minimizing the cost of waiting as you can use a DSM solution that is already operational right away. Such DSM solutions are most often made available via tools that are available as standalone products or are part of the hardware or target environment you are already using. They do not, however, give you any competitive advantage: All your competitors can use the same automation.

Use of existing DSM tools also means outsourcing your core competency: tool lock-in in the worst possible way. If you have specific needs for the modeling language or generator, they may not be available when you need them, or maybe even never. Ready DSM solutions usually lead to limited possibilities for automation since they

don't recognize all your specific needs. Since the tool vendor wants to have a bigger market for the tooling, the support becomes more generic, hindering possibilities for raising the level of abstraction in your application domain. In other words, you have limited control on how you develop your applications.

2.5 SUMMARY

DSM offers several fundamental benefits over general-purpose and manual approaches, such as increased productivity, improved quality, and shared expert knowledge. Perhaps most visible is the increased productivity: companies are witnessing order of magnitude improvements. The benefits to the industry of such an increase in productivity are clear. Even clearer are the commercial benefits to be gained in areas of fast technological development and short product lifespan in terms of reduced time to market.

Automation with DSM also offers possibilities to achieve better quality applications: instead of adding more testing work to improve quality, we prevent errors from happening during the specification. This is when they are cheapest to correct too. A fundamental element in DSM is that expertise can be leveraged for the whole team: a few experienced developers define a modeling language containing the domain concepts and rules, and specify the mapping from that to code in a domain-specific code generator. Then the application developers create models with the modeling language and code is automatically generated. In an era of rapid change and fast employee turnover, some other important improvements are related to reduced training costs and easier introduction of new team members.

DSM allows companies to divide the development work differently. Often a company that automates its development activities can better keep its core competency in-house; seeking cost-effectiveness by outsourcing is not as relevant anymore. On the contrary, DSM can also enable outsourcing as it becomes easier to find external companies to do development work—now the external ISVs and consultants need to master just the domain and language, and the rest can be hidden within the DSM solution. Implementation of the DSM solution is not an extra investment when we consider the whole application development cycle from initial requirements and design to working code. Rather, it saves development resources quite quickly. Traditionally, all the developers work with the domain concepts and map them to the implementation concepts manually. And among the developers, there are big differences. Some are experts, but most are not. In DSM, the experienced developers define the concepts and mapping once, and others need not do it again. A code generator that is defined by an expert will, no doubt, produce applications faster and with better quality than those created manually by average developers.

PART II

FUNDAMENTALS

In this part, we describe the fundamentals of Domain-Specific Modeling (DSM). In Chapter 3, we define DSM and how it differs from other languages and generators. In Chapter 4, the architecture of DSM is presented and its main elements—language, models, generators, and domain framework—are described.

CHAPTER 3

DSM DEFINED

In this chapter, we start by defining key characteristics of DSM—those distinguishing it from other modeling and code generation approaches. Then in Section 3.2, we discuss how it affects the daily life of developers and declare what DSM is not. In Section 3.3, we compare DSM to other model-based approaches, namely, general-purpose languages like UML, executable UML, MDA, and UML customization via profiles. Finally, we conclude by discussing the role of tooling and how tools for DSM differ from traditional CASE tools that provide a fixed way of modeling, working, and generating code.

3.1 DSM CHARACTERISTICS

Raising the level of abstraction and using automation can be done in multiple ways: for example, using software platforms, frameworks, or component libraries. These offer abstractions that help in managing complexity but usually still require developers to program and specify mappings to the components manually, in code. Traditional modeling languages, such as UML, IDEF, and SSADM, usually do not help developers much here since the languages are normally based on coding concepts and other concepts with loosely defined semantics. In UML, an example of the former is class diagrams and of the latter, state machines, activity diagrams, or use case diagrams. In both cases, the modeling concepts do not relate to any particular problem domain or, on the implementation side, to any particular software platform,

framework, or component library. Modelers can therefore create models and connect their elements together regardless of the rules of the domain or particular implementation.

DSM changes this. It allows one to continue raising the level of abstraction and provides the necessary automation. A key element in raising the level of abstraction is having modeling languages that map more closely to the problem domain. DSM hides complexity while still guiding developers in making designs within the particular domain. To provide automation, models need to be mapped to implementation code. In many cases this can be achieved by purpose-built generators and supporting framework code. Next let's discuss key characteristics of DSM.

3.1.1 Narrow Focus

DSM focuses on automating software development in a narrow area of interest. As its name indicates, it is domain-specific rather than general-purpose. The narrower and more restricted the focus can be made, the easier it becomes to provide support for specification work and for automating otherwise manual programming work. Unlike a general-purpose approach, DSM can support the development tasks since the modeling language knows about the problem domain and generators can master the solution domain, that is, the implementation side.

Although often a narrow domain is best found inside a single company, it is possible to define domain-specific solutions that can be reused in multiple competing or cooperating companies. Also, domain-specific standards may form the basis for domain-specific languages and may be accompanied by company-specific code generators. Think about the opportunities offered by a standard like AUTOSAR in the automotive world or the set of standards the IETF has defined for IP telephony.

Size of the Domain for DSM A narrow focus means that a particular DSM solution rules out all other application areas: It can't be used for developing any other kind of features or applications than those the developers of the DSM solution intended. Usually a narrow focus can best be defined inside a single company. This means that sharing exactly the same DSM solution with another company in the same domain is not generally possible. Naturally, many of the basic concepts can be directly applied, but not necessarily all the details. And it is the details that matter when providing automation. If we just want to provide an overview or make design sketches, we could use any language—UML diagrams, presentation slides, or even plain English.

Inside a company, a DSM solution usually addresses just part of the whole company domain. A whole bank, a car, or a television is too large a domain as such and DSM addresses smaller domains. For instance, in a bank, a DSM solution could be narrowed to leasing operations or to investment products. Similarly, in a car manufacturer, narrow domains could include light management, an infotainment system, or just voice control. In a television manufacturer, a narrow domain could be firmware or setup and settings applications available to the consumer via a remote

TABLE 3.1 Example Domains for DSM

Problem domain	Solution domain/generation target
Applications in microcontroller	8-bit assembler
Business processes	Rule engine language
Call services	XML
Car infotainment system	Third generation language (3GL)
Control unit of a medical device	3GL
Deployment of telecom network elements	Proprietary directory language
Diving instruments	C
Environment control and management	C
eCommerce marketplaces	J2EE, XML
ERP configuration	3GL
ERP development	C#
Geographic information system	3GL, rule language, data structures
Handheld device applications	3GL
Household appliance features	3GL
Industrial automation	3GL
Industry robots	C
Insurance products	J2EE
IP telephony	XML
Machine control	3GL
Medical device configuration	XML
Phone switch services	CPL, voice XML, 3GL
Phone UI applications	C
Phone UI applications	C++
Platform installation	XML
Portal configuration	Java, HTML, SQL
Retailing system	SQL
SIM card applications	3GL
SIM card profiles	Configuration scripts and parameters
Smartphone UI applications	Python
Telecom services	Configuration scripts

control. The narrower the area of interest is, the easier it usually becomes to provide effective languages and automation with generators.

Table 3.1 outlines 30 examples of areas in which the authors have been involved in defining DSM solutions. The problem domain is usually most visible in the modeling language and the generator produces the code according to the solution domain.

Narrow Problem Domain For most developers, a narrow focus is set by providing a language that operates on already known concepts relevant to the specific domain and has rules that guide developers in making the specifications. For example, a language can prevent incorrect or poor designs by simply making them impossible to specify. This prevents errors early on—when they are cheapest to correct. Such guidance can be done during every modeling step by checking that models follow the

metamodel (language specification) or during a separate model checking process. In the former case, the language could prevent making illegal connections between certain model elements or force a modeler to specify certain data. In the latter case, model checking could report illegal structures or incomplete designs. The narrow focus of the language can also help modelers in following approved design guidelines and in reusing available specifications. By focusing on precise concepts that are already known, the models in DSM become easier to read, remember, check, and validate.

Narrow Solution Domain for Generation The generators are obviously domain-specific too as it would be impossible to have general-purpose generators. In DSM, the generator reads the models based on the metamodel of the language to produce the required code. DSM is not restricted to a specific target language. As the cases in Table 3.1 illustrate, the target languages can cover the whole spectrum from assembly to 3GL and object-oriented languages as well as scripting languages and various proprietary languages. The target language alone is not the factor that narrows the focus; each case specified structures for the generated code and many used libraries and platforms that further narrowed the implementation space. For example, in a mobile phone that can run code written in C++ it is not possible to run just any C++ whatsoever; it needs to follow the programming model, services of the operating system (e.g., Symbian), related user interface framework (e.g., S60), and application framework (e.g., one for developing enterprise applications).

A narrow focus enables generators that provide efficient code, ideally following the same structure that the best programmers would write manually. The code produced therefore follows similar structures and patterns. If the code needs to be changed to work based on different structures or even programming language, the change is done mostly in the generator. This change can be made by a few experts in the company rather than by all the developers, as in traditional manual approaches. It is worth noting that in DSM a generator does not do everything on its own as there is often legacy code and platform code that already handles part of the task on the implementation side. In Part IV, we discuss in more detail how to integrate generators with existing code.

3.1.2 High Level of Abstraction

DSM raises the level of abstraction beyond current programming languages and their abstractions by specifying the solution directly using domain concepts. As discussed in Chapter 2, such an upward shift in abstraction generally leads to a corresponding increase in productivity. Improved productivity refers not just to the time and resources needed to make the specification in the first place but also to the maintenance work. For example, requirement changes usually come via the problem domain, not the implementation domain, so such changes are most naturally specified using the same domain terms.

Language Concepts Map to the Problem Domain For developers, the modeling languages provide the mechanism for raising the level of abstraction. In DSM, the model elements represent things in the domain world, not the code world. The modeling languages follow the domain abstractions and semantics, allowing modelers to perceive themselves as working directly with domain concepts. Ideally every language construct originates from the domain and the rules of the domain are included in the language as constraints.

This close alignment of language and problem domain offers several benefits. Many of these are common to other ways of moving toward higher levels of abstraction: improved productivity, better hiding of complexity, and better system quality. For similar reasons, in the specification language, it is usually a good idea to use not the concepts of implementation but the concepts of the actual problem. If models had been used in the past to visualize part of the assembler code, the move to higher abstraction in C could not have been achieved. Similarly, today, using class diagrams to visualize related definitions in code prevents modelers from raising the level of abstraction.

Generators Map Models to a Solution Domain Generators close the gap between the model and code worlds. The generator specifies how information is extracted from the models and transformed into code. In the simplest cases, each modeling symbol produces certain fixed code, including the values entered into the symbol as arguments. The generator can also generate different code depending on the values in the symbol, the relationships it has with other symbols, or other information in the model.

The generator itself is usually invisible to modelers and the construction and modification of the generator is done by just a few experienced programmers. At this point, we should note that the generator is not usually solely responsible for providing the mapping to the code since the supporting domain framework and target platform bring the implementation world closer to the problem domain. For instance, libraries, components, frameworks (Fayad and Johnson, 1999), and domain-specific architectures (Duffy, 2004) already make the work of generators easier by raising the level of abstraction on the code side.

3.1.3 Full Code Generation

In DSM, full code is generated from the application developer's point of view and manual rewriting of the generated code is not needed. This completeness has been the cornerstone of other successful shifts made with programming languages. The generated code can be linked with the existing code and compiled to a finished executable without additional manual effort (e.g., Batory et al., 2000). The generated code is thus simply an intermediate by-product on the way to the finished product, like .o files in C compilation. Such automated full code generation is possible because both the modeling language and generators need fit only narrow requirements.

We should keep in mind that full code generation is inspected here from the modelers' perspective: legacy code, components, and other supporting framework code may be written manually. In Chapter 4, we discuss the division between languages, generators, manually written and legacy code in more detail.

Both Static and Behavioral Code is Covered Full code generation requires that the code produced cover both static and behavioral structures. DSM can provide support for both. Usually producing static code is relatively simple, and most template-based generators provide for this. Since generating static code like class skeletons or database schemas is well known, in this book we focus mostly on the behavioral side. For the same reason, the example cases in Part III deal (with one exception) with generating behavioral code. Generating behavioral code is more challenging and support from the modeling language and usually from the domain framework also becomes necessary.

Single Source, Multiple Targets Models expressed in domain terms can also be used for purposes other than producing code. In DSM, generators can produce simulation, prototypes, metrics, test material, configuration and deployment, and build scripts as well as documentation. Having a single source, models, is a powerful concept as changes in one place can automatically update other related artifacts.

3.1.4 Representations Other Than Text

Specifications in a problem domain cannot necessarily be best represented using pure linear text, as typically used in programming languages. What works for a compiler does not work for specifying a solution in a problem domain. Although text is quick to enter and concise, it is prone to errors on entry, hard to manipulate during generation, and any constituent parts are difficult to reuse elsewhere. Partly for these reasons during the past few decades newer programming languages have not realized a closer alignment between the problem domain and solution domain.

In DSM, other representations such as graphical diagrams, matrices, and tables are used along with text to provide the desired closer mapping to the problem domain. For example, in a graphical flow diagram, the execution order is based on connections between model elements, not on the sequential order of lines of text. The connections can describe the system in a richer manner, for example, with typed, parallel, and directed connections that are not well supported in pure text-based specifications. Graphical models allow almost any structures, and are easy to understand, work with, generate from, and reuse. It is worth remembering the well-known saying that a picture can say more than 1000 words. This is becoming increasingly important since the amount of information in current systems is beyond what we can handle. Pictures are also especially good for humans since we are good at spotting patterns in images, whereas a textual representation works better for computers.

Different Views Different views may require different languages and representations. In DSM, this is achieved by creating several languages that share some of the same model data or link to each other. The number of different views or languages depends on the domain. At one extreme, there can be one language for each individual view or it is possible to embed multiple different views or aspects into a single modeling language. For example, the watch case in Chapter 9 uses two languages, one for specifying the products and their static structures, the other for specifying the application behavior. The latter language follows as MVC architecture using different concepts and coloring for different architectural aspects.

Different users may also require different views and languages, especially if their roles in making specifications are different. For example, there can be one language that focuses on hardware structures, another that specifies communication networks, a third the device architecture, and a fourth the application functionality. In DSM, such different views can be integrated by integrating the modeling languages or by using separate languages and integrating the specifications during code generation.

Scaling and Information Hiding Graphical diagrams, matrices, and tables also scale better than pure text by offering different levels of detail. This can be offered with submodels, hiding unnecessary information, providing different views, or linking related specifications. Rather than copying the same data to multiple specifications or places, a DSM language (and related tools) can minimize the need for specifying explicit links between specifications and keeping them up-to-date. Concepts in the modeling languages can be integrated so that reuse is guided or enforced: rather than giving new design data, modelers are forced to choose from data already given elsewhere. For example, in the mobile phone case, the content of the SMS message is selected from the variable data already given elsewhere in the application. As a result, the message data can be kept automatically up-to-date even if the original variable name is changed. In Chapter 10, we give more detailed guidelines for modeling language construction.

3.1.5 Larger Number of Potential Users

We are used to thinking that programming languages are used by programmers, whereas models are used by designers, along with most other stakeholders, such as analysts, requirements engineers, customers, and managers. Some groups, like those doing the testing, configuration, and deployment, can apply both. Models expressed in DSM have a larger group of potential users than code.

On the specification creation side, the models can also be made by people other than traditional developers. In some cases, even nonprogrammers can make complete specifications. For example, the case of insurance products in Part III shows how models are created (and code generated) by insurance experts. Similarly, the telecom example demonstrates how IP telephony services can be created by service engineers instead of programmers.

Specifications are not used by just their creators. With DSM, models expressed in higher level domain terms are easily understandable by other stakeholders in a team.

For example, customers and managers can be expected to read, check, and accept them as models are based on known concepts. This can be seen to improve communication and participation in the development work. Models expressed in domain terms can also be used by test engineers, deployment, and product configuration personnel and we can expect that cooperation with quality assurance is improved.

DSM also introduces a special group of people: those creating the DSM solution that others use. They create the languages and generators along with possible code for a domain framework making DSM work. This group is usually found from within the same company that uses the DSM solution but can also be an external consulting force or external company providing the DSM solution along with its tooling.

3.2 IMPLICATIONS OF DSM FOR USERS

DSM has major implications for the role of languages and generators. We describe here some of the most notable. Implications of DSM in companies, especially when compared to using manual practices, were discussed in Chapter 2.

3.2.1 What Does DSM Offer for Developers?

DSM brings changes to the daily life of application developers. The following changes are evident from real-world experience, although some may seem incredible to those used to general-purpose languages and partial code generation.

You Can Trust the Models, They Are Formal. In DSM, models can be used to generate code, so they can equally be used for executing, testing, and debugging the application or feature developed. Models are the primary source to edit. All coding, however, does not disappear for everybody since we need developers who implement the generators, provide framework code, and make reusable libraries and components. Part IV of this book gives guidelines for defining the DSM solution.

No Need to Learn New Languages and Semantics. Problem domain concepts are typically already known and used. They have well-defined semantics and are considered “natural” as they are formed internally. Because these domain-specific semantics must be mastered anyway, in DSM they are given first class status. Developers do not need to learn additional semantics (e.g., UML) and map back and forth between domain and UML semantics. This unnecessary mapping takes time and resources, is error prone, and is carried out by all designers—some doing it better, but often all doing it differently.

Routine Tasks Are Minimized. Generators can’t provide intelligence but rather automate repetitive tasks. This is not new, for decades we have successfully used, for example, compilers to automate similar kinds of repetitive tasks. Now generators provide the same automation but in the context of a specific application domain. This

automation allows developers to focus on more interesting topics and address application functionality rather than its implementation details.

Less Specification Work Needed. Keeping specifications at a significantly higher level of abstraction than traditional source code or class diagrams means less specification work. As the language need fit only a specific domain, usually in only one company, the DSM can be very lightweight. It does not include techniques or constructs that add unnecessary work for developers. Consider here, for example, the difference between the amounts of modeling work needed in the UML and DSM approaches presented in the mobile application case in Chapter 1.

Less Testing Needed and Many Typical Errors Disappear. In DSM, a large portion of the testing is effectively done before the modeling stage as the language contains the rules of the domain. The rules need to be in the modeling language to avoid generating code from models that are full of errors. Similarly, most typical errors in manually written programs, such as typos, missing references, using variables not yet initialized, and errors in memory allocations, no longer occur because the code is generated. Application developers thus don't need to test for these anymore as they are effectively tested by the language.

No Need to Change the Generated Code. The generator is made by your own expert developer, not a vendor, so it already produces the code you need. And if it doesn't, your expert developer can change it. At this point, we need to mention that, instead of changing just the generator, the expert developer may also change the modeling language or the framework code that supports the generated code. Such changes to the modeling language generally focus on having all the relevant design data needed for the generator to produce good code.

DSM also brings several smaller changes to the daily life of a developer. For many, their value might not be small at all, like having no extra step to document what has been developed or maintain configuration files and build scripts in parallel with the application code. These can be generated from the same source so that they are always up-to-date. DSM can also make the process agile as changes are faster and easier to make at a higher level of abstraction: changes are made with domain concepts and a changed application can be generated. Since the raise in the level of abstraction hides the implementation details, developers don't need to learn the details of the underlying framework and libraries. Using the mobile application case as an example (Chapter 1), with domain-specific language we did not need to remember in which library the text message function is, how to include it in the application, how to call it, what parameters it requires, and how it behaves in the application.

3.2.2 What DSM is Not

Implications for development work can also be inspected by describing situations that are not typical for DSM.

Modeling with Pure Code Concepts. In DSM models do not try to visualize code or apply coding concepts as the constructs of the modeling language. Coding-related constructs are usually easy to add to the language later if needed, but not necessarily the other way around. If the domain is close to the code, the domain-specific language can also apply code concepts. Most likely, however, the level of abstraction will not then be raised much and the benefits of code generation will remain modest.

Modeling for Sketching or for Documentation Only. While models serve as a mechanism to get a better understanding in DSM, they are also used as input for code generators. DSM does not require an additional modeling phase for documentation as it can be generated from the same single source.

Heavy Up-Front Modeling. For some people, modeling is seen as an inefficient step creating a lot of unnecessary models at the beginning of the project. While this might be true when models are separate from the application built, in DSM the models are the source. In DSM, we model only those aspects that are also needed in later development stages, such as for generating production code, simulation, configuration, or test cases.

Generating Partial Code that Needs to be Modified. We would not be happy if after writing C and compiling it we needed to modify and rework the assembly language or machine code produced. Similarly, we are not happy if after modeling we need to modify and rework the generated code.

Generating Inefficient Code. Many developers have bad experiences with third party generators because the generator vendor has fixed the method of code production. Despite the existence of multiple ways to write code for a certain behavior, the vendor has chosen just one of them. The vendor's chosen way, one-size-fits-all code, is often not likely to be ideal for your situation, taking into account the target language generated, the programming model used, memory use, etc. Third party generators often don't have enough information about an organization's specific requirements or possible legacy code and libraries to generate ideal code, so it is not surprising that many have found the generated code unsatisfactory. Because modifying the generated code is usually not a realistic option, organizations end up throwing away the generated code. The value of the generated code is then limited to prototyping and simulation.

Using Round-tripping. Round-tripping aims to minimize the effort needed to keep information up-to-date in two or more places, for example, one model and multiple files. In DSM, round-tripping is not usually relevant at all as the level of abstraction is raised from code to models: people don't expect to round-trip between changes in C and assembly language either. Reverse engineering still has a place in DSM, for instance, when importing libraries, usually their signatures, to be referred to in models.

Tool Vendor Dictatorship. One reason why CASE tools failed was that they were built based on the idea that a third party tool vendor knows best how your particular application should be developed. Even worse, the languages and generators were fixed in the tool so that users could not change them. In DSM, the core competence of software development, mapping from a problem domain to a solution domain, is not outsourced to tool vendors but is kept in house. The experienced developers who build the language and generators can freely change them at will. They don't need to wait for the next version from the tool vendor and hope that it includes the required functionalities.

3.3 DIFFERENCE FROM OTHER MODELING APPROACHES

There is a wide variety of modeling languages available. Most of them are not made to enable truly model-based code generation, though. This is especially true for many general-purpose modeling languages that have become best known because of their standardization. Examples of such are Merise, SSADM, IDEF, UML, and SysML. It is worth noting that in the past what has made the languages viable is not pure standardization but their practicality in automating development. For example, in the telecom area SDL works well for protocol design and similarly many standards for data modeling and schema definition (e.g., Express and Express-G). Standards cannot offer much security either. In the short term they evolve and newer versions of the language may not be compatible with the old one, or at least the tools implementing it may not exist anymore. Some languages, notably UML, are so poorly defined in places that it becomes impossible for tool vendors to implement the language in the same way. Standard languages also have a life cycle and in the next major change may become obsolete. This happened to many past modeling language standards. In the following, we describe how DSM differs from other modeling approaches.

3.3.1 How Does DSM Differ from UML?

UML has done a great favor to the software industry by emphasizing the need to consider design first. Unfortunately, the UML standard offers very little help in automating development work or increasing productivity. As demonstrated with the example in Chapter 1, UML does not raise the level of abstraction above code concepts nor adequately support code generation. You can test our claim by trying to apply UML to automate the development of any of the five examples presented in Part III. We should keep in mind, however, that UML was originally set up not for automating development but for agreement on modeling concepts, their naming, and symbols. The emphasis of the language was on “specifying, visualizing, and documenting the artifacts” (Booch and Rumbaugh, 1995, page 1) rather than on supporting developers in making the design decisions or automating development with generators. Within a narrow domain, DSM aims to do all these.

The central concepts of UML originate from the code world: classes, methods, attributes. Each company that uses UML also has its own domain: its own set of

concepts that make up the products it produces. Furthermore, even two companies making similar products will each have their own kind of code. UML tries to offer a “one size fits all” set of concepts and generators, making it a “jack of all trades but master of none.” From most UML models, virtually no code can be produced, and even if the full set of UML models is made, only a small fraction of the total code can be generated. Test and see by trying to implement generators from the models described in Chapter 1.

If we turn from code generation to inspect the wider development process, the lack of support available from a general-purpose language is evident. Unlike DSM, with UML it is not possible to know how and when to reuse data from models or from external code, choose between patterns for a given task, ensure that application developers follow your architectural rules, check design correctness based on your domain, separate the model data into different aspects relevant in your domain, and so on. The reason for this is simple: these are impossible to standardize as they differ from one domain and company to another. Even in the same team, use of UML for model-driven development would require that all developers remember all these rules and twist standard UML semantics similarly to convey their design to other members of the team. In some cases, often very close to the implementation, UML has been used to automate development with more extensive code generation. Deeper inspection of such cases shows that UML is not followed as in the standard: the notation may look the same but the meaning of the concepts and structure of the language (metamodel) have been changed. In practice, the first step toward a domain-specific approach has been taken.

3.3.2 How Does DSM Differ from Executable UML?

Similarly, the initiatives that aim to use UML as a programming language (Mellor and Balcer, 2002; Raistrick et al., 2004) cover parts of the whole UML. Deeper inspection of these approaches and their implementation in tools (like BridgePoint, iUML, OlivaNova) shows that the UML standard is not followed and the UML modeling concepts are modified and extended. This is not particularly surprising since the foundation for executable modeling ideas already existed before UML (e.g., Mellor and Shlaer, 1991; Pastor and Ramos, 1995). In executable approaches, additional textual languages are applied along with models, by using constraint languages like OCL (OMG, 2006) and various action languages or even traditional programming languages to describe state changes and other actions in the models. Perhaps most typical here is to provide a class diagram to specify the structure while keeping the rest in coding terms. This is not truly model-driven nor does it make creating models attractive: Developers first need to learn UML, or rather a subset of it with some modifications to the standard version, then learn some constraint languages, and finally learn some action languages if writing the functions and actions with the preferred programming language is not possible in that tool.

While executable UML targets code generation, the level of abstraction in models is low and the support for creating specifications modest. Similarly to UML, executable UML does not know anything about a particular problem domain. Modelers can therefore create and connect model elements, but must write related

OCL and action languages without having support for finding a solution. Again the reason is that languages for executable UML are general purpose. The low level of abstraction leads to models that illustrate code. This is evident also in models that describe behavior: a typical example is showing direct method names, parameters, variables, and other code constructs in state machines. This approach has not received much interest since the models only map to a few domains that are already close to technical coding terms. Also most developers working at the code level often find it better to write the same structures directly in a programming language rather than using UML and additional textual languages.

3.3.3 How Does DSM Differ from MDA?

OMG has tacitly admitted that full code generation from UML is not going to happen and aims toward a model-driven architecture (MDA, OMG, 2003). MDA uses three different kinds of models: models that are independent of computing details (CIM), models that are independent of the computing platform (PIM), and models that are specific to a particular computing platform (PSM). This model structure is the A (architecture) in MDA. At its most basic, MDA involves transforming one UML model into another UML model, possibly several times and possibly automatically, then automatically generating substantial code from the final model.

In MDA, model transformations normally mean that during each step developers extend the automatically produced models with further details. DSM aims to generate code directly from the models without having to modify generated models or code. This is the same recipe that made compilers successful. The difference between MDA and DSM is very visible in agile processes and especially in maintenance, where changes need to be made to models created earlier. In MDA, the obvious, and still unresolved, challenge will be in correctly making changes to models that were created partially by generation. Therefore, the MDA approach leads to the same results as wizards: lots of code (and models) that you didn't write yourself but that you are expected to maintain. Such wizards can sometimes be helpful, and they do offer increased productivity at the start, but over time creating a mass of unfamiliar models and code that needs maintaining tarnishes the picture considerably. The MDA idea gets even worse when you consider round-tripping—would you like to update the manually made changes to the code and lower level models back to all the higher-level models?

The MDA way to handle such model updates, forgetting here reverse engineering, is to use the same language at all the levels and use only a very few concepts, like a class. This naturally lowers the abstraction level we can use in the models. Ironically, each move toward better synchronization between the levels is thus a move away from having a higher-level language and a lower-level language.

When it comes to standardization, it is obvious that the best DSM solutions will never be standardized. Modeling languages and generators that fundamentally increase productivity and improve quality give competitive advantage and are naturally kept for internal use only. Why would any company that outperforms its competitors publish its DSM solution? Standards that are widely adopted are still

good but we should remember that then the leading edge of competition has moved elsewhere. For example, standard libraries are good for automation since they narrow the focus for a DSM solution and often raise the abstraction from the implementation side. For practical DSM, the existence of standard platforms is not that relevant since DSM solutions can be built based on standards or equally well on proprietary and in-house platforms and target environments.

3.3.4 What If We Customize UML?

Some MDA proponents envisage higher forms of MDA incorporating elements of DSM. In these, the base UML can be extended with domain-specific enhancements using profiles that let us add new attribute types for model elements, classify them with stereotypes, and have domain-specific constraints right in the language by using OCL, a constraint language. Profiles allow taking a first step toward DSM. However, profiles offer a limited extension mechanism since totally new types can't be added to the language. Also, profiles can't allow taking anything away from the UML since profiles are based on existing UML concepts. Therefore the use of models for code generation, analysis, checking, or documentation needs to access the extended language concepts via mandatory and possibly unnecessary UML concepts. Currently, the OMG standards give an indication that such removals could be done, but no reference implementation or tool support yet exists. An obvious test for such a standard would be to remove the whole UML with profiles and build a totally different language.

Profiles can still be used in cases where the difference from basic UML concepts is small. This leads again to applying the abstractions of UML and making any larger deviation—to map modeling concepts more closely to the problem domain—would lead to unnecessarily large and complex language definitions. Usually, it is far better to just say “dog” than to say “cat” then explain how “dogs” differ from cats. You may test this yourself by defining profiles to implement the sample languages from Part III.

Because of these limitations, the OMG has proposed another form of customizing modeling support, the Meta-Object Facility, MOF. This approach is similar to DSM since MOF describes the concepts of a language and how models of those concepts are to be stored and interchanged. Although the MOF specification is large, use of it to model languages describes little about those aspects that are of direct interest to its user: what models in the language actually look like or how the user interacts with them. MOF also lacks an explicit language concept and does not support language integration or *n*-ary relationships between model elements.

3.3.5 Other Domain-Specific Approaches

Developers may also use some readily available DSM solutions. For example, entity-relationship modeling is a widely known and used technique to design schemas, especially for relational databases. Although this is a relatively small domain, there are multiple different languages available. Standardization has not been prioritized here; the capability of each language to support the specific characteristics of a

particular database has been seen as more important. The same applies for GUI design and the design of protocols which can also be characterized as domain-specific solutions. Similarly, tools that come with model-based code generation, like Labview or Matlab/Simulink, can also be considered as domain-specific solutions. The main difference from DSM is that these are often fixed so that no users, not even the experienced developers in a company, can change them.

3.4 TOOLING FOR DSM

Modeling and code generation do not work without tools: they are a fundamental part of the automation. Modeling tools allow creating, checking, verifying, reusing, integrating, and sharing design specifications, among others. The importance of tools for modeling was already recognized in the 1960s: the first software product sold independently of a hardware package was a flowchart modeling tool called Autoflow (Johnson, 1998). If the specification models are just throw-away sketches, then naturally tools don't matter. In truly model-based development (see Fig. 1.1), however, tools are mandatory. Obviously, generators translating models into code also need to be based on tools.

3.4.1 Rethinking the Tooling

DSM gives freedom to experienced developers in companies: they define how the applications should be designed, specified, and produced. This requires different kinds of tools than the current development tools in which languages and transformations are fixed. These tools, earlier called CASE tools, dictate the development by providing a few fixed modeling languages, enforcing a certain way of modeling, and generating code in a certain way. In reality the situation should be the opposite. In many cases, experienced developers in a company have better knowledge of how their software should be developed than a modeling tool vendor!

Unfortunately, traditional modeling tools, such as UML tools today or CASE tools in the past, do not give adequate freedom and power to developers to adapt the modeling languages and generators. These tools are based on a two-level architecture: system designs are stored in files or a repository, whose schema is programmed and compiled into the modeling tool. This "hard-coded" part defines what kind of models can be made and how they can be processed. Most importantly, only the tool vendor can modify the modeling language, because it is fixed in the code. Metamodel-based technology removes this limitation and allows flexible modeling languages. This is achieved by adding one level above the level of modeling languages, as illustrated by the top right box of Fig. 3.1.

Metamodel-based tools follow a three-level architecture. The lowest, the model level, is similar to that of CASE tools. It includes system designs as models. The middle level contains a model of the language, that is, a metamodel. A metamodel includes the concepts, rules, and diagramming notations of a given language. For example, a metamodel may specify concepts like a "use case" and an "actor," how

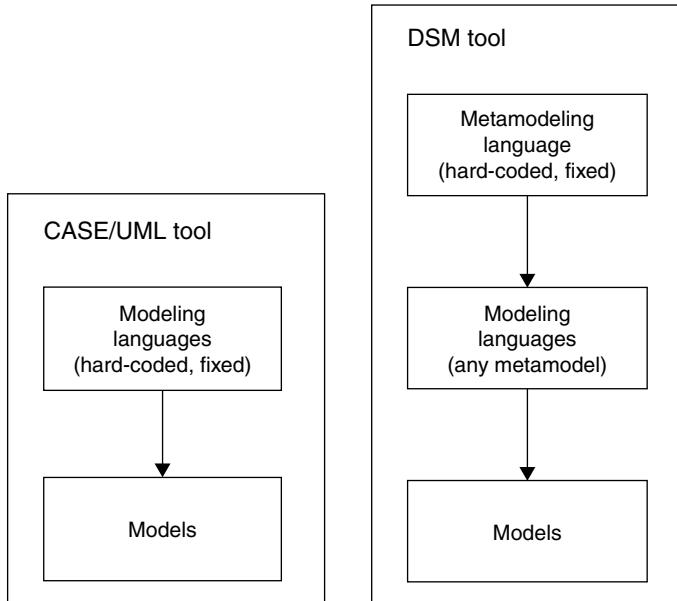


FIGURE 3.1 CASE or UML tool versus DSM tool

they are related, and how they are represented. However, instead of being embedded in code in the tool, as in a fixed CASE tool, the modeling language is stored as data in the tool.

Unlike a UML or CASE tool, a metamodel-based tool allows the user to access and modify the language specifications. This is achieved by having a third, higher level that includes the metamodeling language for specifying modeling languages. This level is usually the “hard-coded” part of the metamodeling tool. All three levels are tightly related: a model is based on a metamodel, which in turn is based on a metamodeling language. Clearly, no modeling is possible without some sort of metamodel. This dependency structure is similar to that between objects, classes, and metaclasses in some object-oriented programming languages.

3.4.2 DSM Tool Capabilities

Having the possibility to change modeling languages and generators is the primary requirement for increasing automation, but by itself it is not enough. Most companies don’t have the expertise and resources to implement their own modeling and code generation tools from scratch. Therefore, there must be a way to quickly, easily, and safely get tool support for DSM.

Time to Implement Tool Support. It should be possible to implement a DSM solution quickly. If we consider the language definition in a modeling tool it should not take longer than a day or two per modeling language. If it requires a lot of

resources, the increasing cost of having a DSM solution limits the number of cases in which automation can be applied. In addition to creation time, we also need to consider the maintenance of the DSM solution: costly and time-consuming modifications to languages and generators can hinder the development process. Usually developers can't wait long for a newer version of the DSM solution. The most important reason for a efficient tool implementation is obviously the value of having the automation in use as early as possible.

Difficulty of DSM Tool Development. Building a DSM solution should be possible without having to program the DSM tool. Metamodeling tools of the early 1990s, which could generally only be used by the people who built them, made the creation of DSM solutions too difficult and resource intensive. This limited the availability of domain-specific tools to only larger companies. At best the developers of a DSM solution would only need to define their language and generators, along with a domain framework to support the generated code and the tool should provide the rest. In this book, we follow this ideal and focus on defining and using modeling languages, generators, and domain frameworks. Tool implementation details are intentionally outside the scope of this book.

Safety of DSM Development. The safety of tool customization becomes crucial in the longer run. If all the design specifications are based on the language and generators defined by a few experts, then changes in the language must propagate to all specifications without deleting or corrupting them. In the worst tools, a change in the modeling language can make it impossible to load design models made earlier. Therefore, the tools should guide expert developers in both creating and maintaining the DSM solution.

Current IDE environments are unlikely to be the best place for creating or using a DSM solution as they originate from and focus on lower-level programming constructs. Too close a link between the model and code worlds can drag the level of abstraction of models back down to the level of the code, as seen with UML. If we really want to raise the level of abstraction, then tools need to change too. Domain concepts and rules are unlikely to be best expressed in code but rather in representations that are closer to the actual domain representations and other forms already in use. For instance, they can be pictorial with spatial information, diagrammatic where elements are connected to each other, matrices, tables, spreadsheets and so on. Naturally, integration with compilers, debuggers, and testing tools is still needed. We will inspect tooling support in more detail in Chapter 14.

3.5 SUMMARY

DSM copies the principle that made the compiler so successful—raising the level of abstraction by removing the need for developers to write Assembly and letting them work in 3GLs and OOP languages instead. But to make it work for a given company, the principle needs to be applied to that particular setting, tailor-made for the

company's own problem domain and code style. The most feasible way of doing this is to give companies full control over creating and maintaining their own domain-specific modeling languages and generators: Experienced developers in a domain are often better able to define how their products should be developed than external tool vendors.

Part of an experienced developer's skill is being able to express in terms of the domain concepts precisely what the requirements mean. By creating a modeling language and rules for that domain, the expert can enable and guide other developers in creating precise definitions of the product at a high level of abstraction. The need for higher abstraction and automation normally prevents the use of general-purpose modeling languages. The root problem for these languages is that changing the representation of a construct without increasing the abstraction level doesn't improve productivity. For example, UML (as a standard defined by OMG) does not contain knowledge of particular problem domain, or component library. Modelers can therefore connect UML elements together regardless of the domain rules. The same applies to many other general-purpose modeling languages such as IDEF, Merise, SSADM, and SysML. A second part of the expert developer's skill is in turning those precise definitions into good code that works on top of their target environment. By specifying that skill into a code generator, the expert developer makes it possible for full code to be generated directly from models.

Because the investment of building a DSM solution is often made by only one or two expert developers at any one company, it pays off quickly as all the other developers can then model in the new language and use the generators to create full code. For building automation, tools play a crucial role; otherwise specifications that can be used as a source for generating code would not be possible. Tools for DSM allow experienced developers to specify the automation in a cost-effective manner: ideally the experienced developer can focus on just defining a language that maps closely to the problem domain and a generator or that produces the implementation; the DSM tool should provide the rest.

CHAPTER 4

ARCHITECTURE OF DSM

In this chapter, we introduce the key elements of a Domain-Specific Modeling (DSM) solution: languages, models, generators, and a domain framework. We start by outlining the architecture of DSM in Section 4.1 and then the following sections describe each architectural element in further detail. Finally in Section 4.6, we inspect the organizational structure and roles in creating a DSM solution, and outline its creation process.

4.1 INTRODUCTION

To get the DSM benefits of improved productivity, quality, and complexity hiding, we need to specify how the automation from high level models to running systems should work. For this task DSM proposes a three-level architecture on top of the target environment, as illustrated in Fig. 4.1:

- A domain-specific language provides an abstraction mechanism to deal with complexity in a given domain. This is done by providing concepts and rules within a language that represent things in the application domain, rather than concepts of a given programming language. Generally, the major domain concepts map to modeling language objects, while others will be captured as object properties, connections, submodels, or links to models in other languages. Thus, the language allows developers to perceive themselves as working directly with domain concepts. The language is defined as a

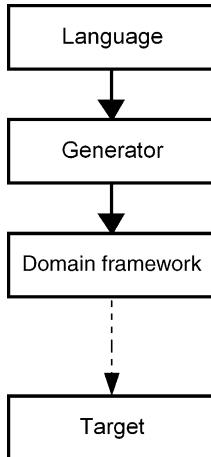


FIGURE 4.1 Basic architecture of DSM

metamodel with related notation and tool support. We inspect the role of languages in DSM further in Section 4.2.

- A generator specifies how information is extracted from the models and transformed into code. In the simplest cases, each modeling symbol produces certain fixed code, including the values entered into the symbol as arguments. The generator can also generate different code depending on the values in the symbol, the relationships it has with other symbols, or other information in the model. This code will be linked with the framework and compiled to a finished executable. While creating a working DSM solution the objective is that after generation, additional manual effort to modify or extend the generated code is not needed. The generated code is thus simply an intermediate by-product on the way to the finished product, like .o files in C compilation. We describe generator characteristics in more detail in Section 4.4.
- A domain framework provides the interface between the generated code and the underlying platform. In some cases, no extra framework code is needed: the generated code can directly call the platform components, whose existing services are enough. Often, though, it is good to define some extra utility code or components to make the generated code simpler. This framework code can range in size from components down to individual groups of programming language statements that occur commonly in code in the selected domain. Such components may already exist from earlier development efforts and products. The role of domain frameworks is discussed in Section 4.5.

The generated code is not executed alone but rather together with additional code in some target environment. This target comes with platform code—the code that is already available with the target. This is used regardless of how the implementation is done, manually or using generators. The developed product can use a selected part of a larger platform (e.g., J2EE), a whole platform (e.g., a call processing server or

microcontroller library), or a number of platforms. For example, in Chapter 8, the DSM solution operates on top of a Python library that is written to work using an S60 mobile phone UI platform, which is again made based on the services of Symbian operating systems.

4.1.1 Dividing the Automation Work

At this point, we should emphasize that there is no single way to divide the automation work between a language, a generator, and a domain framework. It depends on the case. Figure 4.2 illustrates some alternative allocations. In Part III, where we describe the DSM examples, we show examples using different kinds of work allocation within the architecture.

The domain framework can be very thin, or even nonexistent, and then most abstraction work is done with the modeling language (case a). The example case of Call Processing Services and generating XML in Chapter 5 clearly belongs to this class. A generator simply takes each concept of the modeling language and maps model elements to elements in the XML schema. No domain framework is needed and the execution engine, a call processing server, runs the service. The case of developing microcontroller applications in Chapter 7 places more emphasis on the generator as it needs to understand the flow logic and memory allocation (case b).

We can also move toward creating some framework code to enable code generation from models. That framework code can be manually written on top of the platform, to be, for instance, called by the generated code, or the generator can contain boilerplate sections of framework code that are output when needed. The case of mobile phone applications in Chapter 8 shows how framework code is provided and produced during the code generator (case c). Finally, the watch example in Chapter 9 shows how existing general platform code (Java and MIDP) is extended with framework code

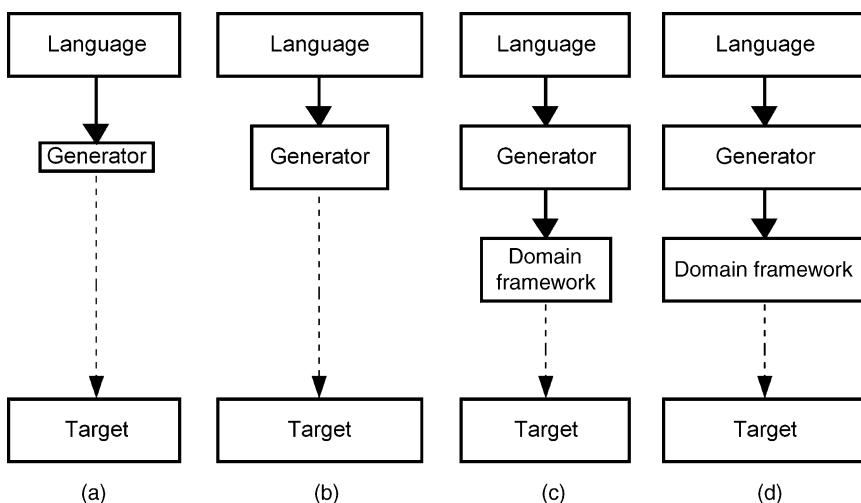


FIGURE 4.2 Dividing the automation work in DSM

(case d) for a chosen application domain. This framework code is implemented into components that are then reused by the generators.

Among the different ways to allocate work inside the DSM architecture, we can also identify some allocations to be avoided. Usually we should avoid automation where a generator carries most of the burden. First and foremost, if modeling languages are not used to increase the abstraction, the benefits of having a generator tends to be modest. Second, the maintenance of generators that aim to do most of the work easily becomes the bottleneck. Generators tend to grow larger in the longer run, and will even during generator construction if the modeling language is not suitable for the design task. A large part of the generator is then for checking the validity of the designs before starting the actual generation or for clumsy navigation in models using data structures (i.e., a metamodel) that are not suitable for generation. This kind of situation is usually detected when developers find themselves learning certain ways to make models just to make the generator work. Then they are actually modeling to feed the generator, not to design the application or the feature.

Similarly approaches where modeling languages can only capture partial design information should be avoided. Usually this becomes evident quickly as it is difficult to use code generators if the input data are not adequate. The classic case here is using plain class diagrams for code generation: generating a class definition into a file(s) from a class diagram is simply transforming the representation from a diagram to text rather than a helpful code generation step. The level of abstraction in model and code is the same, and since the generated code is partial, you need to fill in the rest manually. Even protected areas for your manually edited code don't work: models can change so they invalidate manually written parts, and references in manually written code don't update if you regenerate from a changed model.

4.1.2 Evolution Within the Architecture

The DSM architecture also allows evolution. Any of the elements can be changed if needed. This flexibility makes the DSM approach different from CASE and 4GLs, which fix at least one of the architectural elements. One notable way is changing the generator to a different target while keeping the modeling language the same. The cases discussed in Part III demonstrate such situations: insurance products in Chapter 6 were planned to be generated in different target languages while keeping the target environment and the functionalities it provides for application execution the same. In Chapter 9, the execution target for wrist watches is extended without modifying the models. Changes to the generator are kept minimal by making most changes directly to the domain framework. In the mobile phone case of Chapter 8, an alternative target platform was available to enable more functionality and wider access to phone services. Although here the domain framework and code generators changed, most parts of the modeling language could be used in both cases.

Usually, however, there is only one main generation target (and framework) per DSM solution. Having multiple generators for different purposes is more usual: In addition to having a generator to produce production code, there can be another one for making an early prototype or one that produces code enabling model debugging

while executing the generated code. Further, generators can also produce test cases, simulations, or metrics.

4.1.3 Models in DSM Architecture

For the application developers, the language remains the most visible part. The language is used to make designs, and a code generator is used to produce production code. The domain framework is normally not invisible to the modelers, in a similar manner as BIOS code or primitives called by the running application are not visible to programmers in a 3GL. In Fig. 4.3, the left side describes the DSM definition and the right side describes DSM use—mainly modeling.

The language is formalized into a metamodel and all models describing applications or features are instantiated from this metamodel. Thus models can't express anything else other than what the language allows. This language instantiation ensures that application developers follow the concepts and rules of the domain in models. In DSM, the languages built for internal use are normally defined by just one or a few people. The role of models in development is therefore a little different from what you may be used to: In DSM, models are the source and primary artifacts with which to work. We describe the role of models in more detail in Section 4.3.

The generator structure is usually not visible to modelers and can be considered similar to a compiler. Following this analogy, the transformation from models to

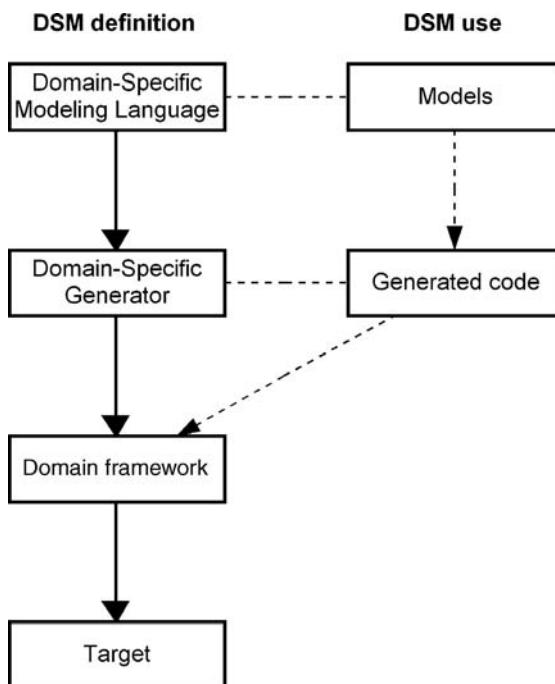


FIGURE 4.3 DSM definition and DSM use

running product code is unidirectional and modification of the generated code is not needed. This completeness has been the cornerstone of other successful shifts made with programming languages. The DSM architecture also shows that all code is not necessarily generated. The domain framework and target environment may be available as code or as interfaces the generator can integrate with. Generated code can also be integrated with manually written code if needed. The code generator and domain framework are often made by the same people, and it is a task that only a few developers perform: most developers are not involved in generator definition, they just use it.

In the following, we describe each elementary architectural unit of DSM in more detail, namely, the language, models, generators, and domain framework.

4.2 LANGUAGE

Language provides the abstraction for development and as such is the most visible part for developers. In DSM, it is used to make the specifications that manual programmers would treat as source code. If the language is formed correctly, it should apply terms and concepts of a particular problem domain. This means that a domain-specific language is most likely useless in other problem domains.

Generally the major domain concepts map to the main modeling concepts, while others will be captured as object properties, connections, submodels or links to models in other languages. This allows users of DSM to perceive themselves as working directly with domain concepts. The focus for the narrow domain is provided through the language properties: its modeling concepts, underlying model of computation, and notational symbols. In the following, we discuss some of the key elements of languages in general and domain-specific modeling languages in particular.

4.2.1 Fundamentals

For domain-specific languages, the same definitions apply as apply to languages in general. Modeling languages are typically seen to consist of syntax and semantics. On the syntax side, we can further distinguish between abstract and concrete syntax. The former denotes the structure and grammatical rules of a language. The latter deals with notational symbols and the representational form the language uses. To increase design abstraction and generate more complete code, you usually need to extend both syntax and semantics.

Syntax Syntax specifies the conceptual structure of a language: the constructs of a modeling language, their properties and connections to each other. In DSM, the modeling constructs ideally come directly from the problem domain. The abstract syntax of a modeling language is normally specified in a metamodel (see Section 4.122.4 for details).

The syntax of a modeling language means more than just reserved words. It is commonly seen as also covering grammatical rules that need to be followed while

specifying models. In DSM, these rules are from the domain and they are defined in the language in relation to the modeling concepts. Rules are needed to avoid generating code from models that have errors. Having rules in place during modeling also makes implementation of the generators easier as generators don't need to start by first checking if models are correct. In DSM, the rules are checked, if possible, as early as possible because this allows detecting and preventing errors when they are cheapest to correct. Consider here the alternative: finding the errors during code generation or from the generated code. Placing rules in the language, rather than in the generator, makes even more sense if there are several generators: it is always better to check the model once than do it for each generator.

The rules of the language typically constrain how models can be created: they define the legal values, relationships between concepts, and how certain concepts should be used. The rules can vary from strict model correctness rules and consistency checking to rules that guide rather than enforce a particular way of modeling. Once the rules are defined, the modeling language—enacted by the supporting tool—guarantees that all developers follow the same domain rules. The rules again significantly reduce the possible design space—the kinds of applications that can be written with this language—and help ensure designers only make appropriate applications. Should the range of applications need to be extended, the modeling language can of course be extended later. We discuss language evolution and the definition of rules in more detail in Chapter 10.

Semantics Every modeling concept has some meaning, semantics. When we add an element into a model or connect elements together we create meaning. In DSM, the semantics of the modeling language come to some extent directly from the problem domain. An example helps here: if we are developing an infotainment system for a car, the modeling concepts, such as a “knob,” a “menu,” and an “event”, already have well-defined meanings within the application domain. This is unlike general-purpose modeling languages where the semantics do not map to a particular problem domain, but it is left to developers to map the semantics and concepts of a language to a given problem domain. Research on modeling language use (e.g., Wijers, 1991) shows that each developer makes this mapping differently. It is no surprise that modelers using general-purpose languages create different kinds of models for the same problem. DSM is different as it aims to rely on concepts and semantics that come directly from the problem domain.

Use of domain semantics in the language is not limited just to the concepts but also covers the connections between the modeling constructs as well as related rules. Following the car infotainment example, a menu in an infotainment system can usually trigger an action or open a submenu. Accordingly, in the domain-specific language a menu can be connected only to an action or to another menu. The former could be defined with a “transition” relationship from a menu to an action and the latter with a “submenu” relationship from a menu to another instance of a menu. To follow the semantics closely, the modeling language also includes a constraint that allows only one relationship, either the “transition” or the “submenu,” to be specified from each menu.

The semantics of the problem domain is not the only source for DSM semantics. Like all modeling languages targeting code generation, we must recognize the semantics of the implementation side: how modeling constructs are mapped to a given solution domain. This mapping is made not to the problem domain but to another language, here to a programming language. This is usually called operational semantics. It is important to realize that the operational semantics cannot be the only source for semantics. If it were, then the modeling language would actually map one-to-one to the generated programming language. The abstraction would be the same and the benefits of code generation minimal. A classic example here is mapping a class in a diagram to a class in a code. The developer who makes the class model is thus already thinking with the concepts and semantics of code. If we want to increase abstraction and improve productivity, the semantics of the problem domain matter more than the semantics of the solution domain.

Concrete Syntax: Representation Pure abstract syntax and semantics are not enough for a language definition: models must be accessed through some visual formalism. We need a concrete syntax in addition to the abstract one. Every modeling language follows some representational form along with a notation. The representational form of most modeling languages is graphical combined with text. Modeling languages can also be based on other representations, like matrices, tables, and forms, or be purely textual.

The choice of notation for a DSM language closely follows the actual presentation of the domain concepts: a valve in a paper mill should look like a valve in the modeling language too, and a control knob for a car infotainment system should have a similar illustration in the modeling language. Ideally, each concept of the modeling language has exactly one notational representation, such as a symbol. This principle minimizes the overload of notational constructs and guarantees that all concepts can be represented in the language. Accordingly, the completeness of representations (Batani et al., 1992; Venable, 1993) or representational fidelity (Weber and Zhang, 1996), that is, availability of only one notational construct for each concept, is a well-known criterion for dealing with interpretations between modeling concepts and notations.

4.2.2 Model of Computation

A modeling language is usually based on some kind of computational model, such as a state machine, data flow, or data structure. The choice of this model, or a combination of many, depends on the modeling target. Most of us make this choice implicitly without further thinking: some systems call for capturing dynamics and thus we apply for example state machines, whereas other systems may be better specified by focusing on their static structures using feature diagrams or component diagrams. For these reasons a variety of modeling languages are available.

Among languages we can find big differences in how they see the software system to be modeled. Do they see it, for example, as a concurrent one, distributed, perhaps using asynchronous communication, having parallel characteristics, using

bidirectional connections, or acting nondeterministically? These kinds of characteristics define the computational model used by the modeling language. Some of them are specified in the abstract syntax (e.g., tree, directed graph, parallel flows), some are also visualized in the concrete syntax (e.g., arrow heads on both ends of a line representing a bidirectional connection), and all in the semantics. Because of these different “flavors” there are various versions of modeling languages. For example, there are different versions of state machines, several classes of Petri-net diagram, and various kinds of data flow diagrams.

Modeling languages can be roughly divided into those modeling static structures and those specifying dynamic behavior. In reality, the division is not so clear and we have languages that specify both sides. For a more comprehensive review of different models of computations used in modeling languages see Buedo (1999).

Modeling Static Structures One class of modeling languages addresses solely, or mostly, the static structures of an application. Perhaps the best known domain-specific modeling languages are those used in database design for specifying data structures, normalizing them, and generating database schemas. Another well-known area that uses specifications made with a domain-specific language and targets code generation is GUI design.

If we inspect modeling languages that originate from coding, we can see that most class diagrams fall into this category too. Although there is interaction between the class instances, they are not usually described in the model. There are, however, some class diagram versions that also capture method calls among classes, taking a step into the behavioral side too. Other example languages here are those producing cluster diagrams, feature diagrams, system diagrams, network diagrams, component diagrams, process matrices, deployment diagrams, Venn diagrams, inheritance models, and naturally entity-relationship diagrams with their numerous dialects.

For DSM, static languages are often easier to specify than behavioral ones. Their use of code generation is also simpler as they produce static properties of a system but not how to compute them. Among the cases illustrated in Part III, we included only one case out of five that specifies pure static structures to generate declarative code. An example of static modeling can be found from Chapter 6.

Modeling Dynamic Structures and Behavior The second major class of modeling languages is the one specifying behavior and dynamic aspects of a system or its part. State machines, interaction diagrams, and Petri-nets are perhaps the most common modeling languages in this category, along with their many dialects. These modeling languages are very typical in cases where the system can be considered as event or state based.

We can also consider here various process and flow diagrams used for process modeling, workflow modeling, data flow, and signal processing. If we look at tools for model-based development, Labview and its modeling language G (National Instruments, 2005) and Matlab/Simulink toolboxes (Mathworks, 2007) focus on

describing behavior and functionality. Their languages are actually domain-specific and within their scope these languages work well. Provided by the tool, their extensions and modifications are done typically by the tool vendor.

In DSM, behavioral aspects need to be specified to generate code that deals with functionality, business rules, and other application logic. In domain-specific languages, such behavior is often captured by extending some of the well-known models of computation. Such extensions can deal with adding specific business process concepts and rules to the language, logic operations, conditions, decision points, and other functional aspects.

Most of the literature describing model-based development uses modeling to generate structural aspects of software systems. As systems almost always have a behavioral side too, and structural aspects tend to be easier, in this book we want to focus on dynamic and behavioral aspects. In Part III, we mostly address cases that capture dynamic behavior and generate code for nonstatic structures. On the modeling language side, state machines and various flow, process, and interaction diagrams are used. Using the models produced by these languages, generators can produce functional code, not just configuration or static code.

Combining MOCs and Extending the Languages Some languages fall into more than one category as they can be seen to capture both static structures and dynamic behavior. Usually they are stronger on one side but also address the other. For instance, in business process modeling, a workflow language describing the behavior can also have modeling constructs that specify structures for the data that is passed between the processes. Conversely, a data model can also include modeling constructs that specify how the data is used by the system. A single language can usually be extended this way only to a certain limit, motivating the use of multiple languages. Several languages are needed not only because of the different models of computation needed but also because of the size of the domain and its rules, having different people creating and using the model or sharing designs with various stakeholders, such as subcontractors.

The choice of a language involves not only choosing a particular modeling language or computational model per se but also the selection of a subset of modeling constructs and extending them for a given domain. Regarding the subset, in every domain where a state diagram is used it is not necessary to apply a history state to remember the state of the system before it was interrupted. Similarly, using combined fragments and specifying inner connections between objects in a sequence diagram based on UML2 are not needed in most cases. A language providing these constructs just adds extra burden for most modelers. Regarding the language extensions, adding new syntactic and semantic constructs is usually necessary to specify software systems adequately. These constructs are domain-specific as their naming, properties, constraints, connections to other language constructs, and overall semantics are formalized keeping only one domain in mind. Remember that seeking automation makes the difference here! While it is fine to apply different approaches in models for sketching and documentation, code generation sets more detailed requirements for choosing the computational model.

In other words, models need to capture enough data to provide input for code generators.

4.2.3 Integrating Languages for Modeling Multiple Aspects

Larger software systems usually require specifying different views, aspects, and levels of detail. Often this calls for using multiple models and languages. Although we can most likely create a language that covers multiple aspects, like user navigation, persistency, real-time aspects, and data structures, it will most likely be difficult to use and maintain. The size of a language in terms of the number of syntactic and semantic language constructs is simply too large. Also, models created with a single large language do not support variability and modification well. For example, while developing automotive infotainment systems, it is usually best to have at least two languages: one for static display definitions and another for behavioral functionality. This allows displays to be changed without necessarily changing any specifications of functionality. Separate languages also allow better hiding of complexity from the design task and guide the modeling work for the given task.

Other reasons that favor the use of multiple languages are that there are different developer roles and that models are made at different times or different phases of the development life cycle. For example, within the same domain one language may target prototyping, whereas another covers the details to produce production code. Sometimes we may need different languages for third parties so that some details can be kept in-house. This often means having languages that are a subset of those used internally.

For integrating modeling languages, two different methods can be applied. In one, transformation integrates models when generating code or at modeling time. In the other, languages are integrated at the language level. That is having an integrated metamodel.

Integration using transformations keeps language specifications separate: once a specification model is made with one language, it can be transformed into another model based on a different language. Usually the transformations are applied only between two languages, although it is possible to have transformations between multiple languages too (Fig. 4.4).

The transformation knows the mapping from one language to another. It usually expects that source models made with one language are complete so that model

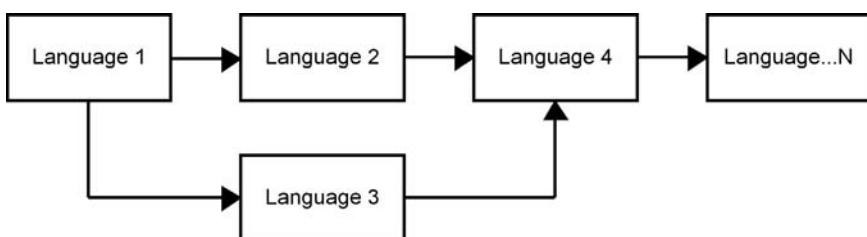


FIGURE 4.4 Integration of languages via transformations

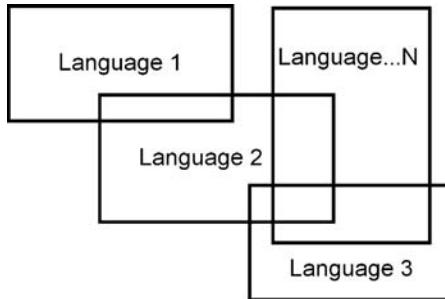


FIGURE 4.5 Integration of languages via common elements

correctness can be checked before the transformation. Otherwise, transformation may lead to incomplete or wrong results. In most cases, the transformations are best performed only once as maintaining the changes in different models based on different languages is difficult, requiring a lot of manual effort. Because changes are difficult to propagate, in the transformation process a waterfall model is expected. Model transformations scale poorly to concurrent development in larger teams and they don't support reuse among designs based on multiple languages.

Modeling languages can also be integrated with shared or linked modeling constructs. Here the language specification is made keeping integration in mind. For example, a data concept can be used on one hand to specify elements passed in the workflow model and on the other hand to specify the data structures for the database schema. If we then change the specification at the schema level, the change can be made available without additional transformations to the workflow model too (Fig. 4.5).

Generally speaking, integrating languages at the specification level is better than using separate transformations. Transformations require copying the same kind of information, either the same design element or a proxy element, to multiple places, whereas integrated languages ideally have only one copy of the concept. This greatly supports the reuse of designs and model refactoring. Also, domain rules can be checked during the modeling stage if the languages are based on the same metamodel. Possible errors are then easier and cheaper to correct when they are identified during model creation. Integrated languages also support concurrent development better and allow the changes made with one language to be more easily shared with other languages.

4.2.4 Language Specification: A Metamodel

In DSM, the modeling language must be defined formally and be supported by some tool. Otherwise, it would not be possible to create models and generate code from them. The language specification is usually called a metamodel. The word “meta” is used because the language specification is one level higher than the usual models. In its simplest form, we can say that a metamodel is a conceptual model of a modeling

language. It describes the concepts of a language, their properties, the legal connections between language elements, model hierarchy structures, and model correctness rules (see Appendix 1 for more details). In all but the smallest cases, support for reuse and different model integration approaches is also essential.

Language design and definition therefore also includes a metamodeling task: mapping domain concepts to various language elements, such as objects, their properties, and their connections, specified as relationships and the roles that objects play in them. Again the word “meta” means that (meta)modeling takes place one level of abstraction and logic higher than the usual modeling in software development. A more comprehensive description of metamodeling can be found in Jarke et al. (1998).

Four Levels Based on Instantiation The idea of a metamodel is well known and has long existed in computer science. More than 20 years ago Kotteman and Konsynski (1984) showed that at least four levels of instantiation are necessary to integrate the modeling of the usage and evolution of systems. A similar observation underlies the architecture of the International Standards Organization’s IRDS framework (Information Resources Dictionary Standard, (ISO, 1990)) and later in OMG’s four-level modeling framework (OMG, 2002; Bezivin and Ploquin, 2001). The levels and their hierarchy are illustrated in Fig. 4.6. One clear way to understand the hierarchy is via instantiation: if you can instantiate your concept once, it is a model, and if the result can be instantiated again, it is a metamodel.

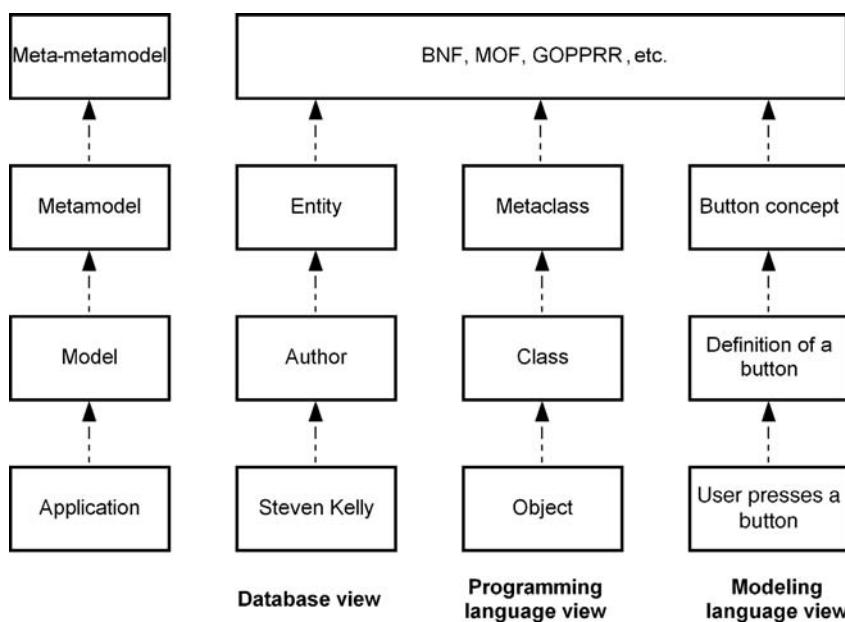


FIGURE 4.6 Four layers of instantiation

Adjacent model layers can perhaps be most intuitively understood by analogy with a database: the upper level is the schema level and the lower level is the database state. Thus the lower level cannot be understood without the upper level. If we inspect the four layers from the database view, we might see “Steven Kelly” as a value in a database of an application. On the model level, we would have a definition of an “Author” as part of the database schema. Reading further up, the schema concept is defined as an “Entity”—a concept specified on the metamodel level. Finally, the meta-metamodel is used to specify the concepts used for database design.

We can identify the same layers from some programming languages too. At the application level we have objects, at the model level we can have classes, and at the metamodel level we can have metaclasses, like in Smalltalk. In a non-object-oriented view we can see program execution, program code, and programming language specification, respectively. In the meta-metamodel level we might see BNF (Backus–Naur Form) as it is used widely to specify programming languages.

If we take a modeling language view, the same layers can again be identified. On the application level, a system is used. Following the digital wrist watch example of Chapter 9, the user pressing a button on a watch is an application level operation. On the model level, a specific button is defined. On the metamodel level, we have the specification of the modeling language, including the button concept. Finally on the highest level, we have a metamodeling language used to specify the modeling language. We usually do this kind of type and instance mapping intuitively as otherwise it would not be possible to understand the models, schemas, and code.

For metamodeling, we can apply different kinds of languages, like GOPRR (Kelly *et al.*, 1996) or MOF (OMG, 2005). If we don’t formalize and use the language to support code generation, it does not matter very much how the metamodeling is done. Following DSM philosophy, a metamodeling language should be made for specifying languages: it should guide language creation, hide unnecessary details, and provide support in producing tools that can follow the specified language. In the end, how the metamodel is specified depends on the tool used. A typical approach is to use a language to specify a modeling language that is then translated into a format that configures a generic modeling tool. A more advanced approach is to support modeling and metamodeling in the same tool, as this supports language creation and language evolution. More on tools for language creation will be given in Chapter 14.

Use of Metamodels is Widely Spread Metamodeling not only is important in defining languages, but also is advantageous in systematizing and formalizing weakly defined languages, providing a more “objective” approach to analyzing and comparing languages and examining linkages between modeling languages and programming languages. Metamodeling is also successfully used in building modeling tools, interfaces between tools (e.g., CDIF, XML), and repository definitions.

Metamodels have been used in standardization efforts, but often only in a limited way: they are not made sufficiently precise and formal and there is no reference implementation in terms of metamodel instantiation. For example, past versions of

UML and its derivatives like SysML have been defined using metamodel fragments, related documentation, and example models. As they are not consistent, the language definitions are left with multiple interpretations. Metamodels in DSM are different as they are formal and have tool support to enable model-based code generation. They also have a reference implementation, although it might be the only implementation of the language.

4.3 MODELS

In DSM, models are the primary source in which developers create, edit, and delete specifications of a system. Although changes to the language, generator, and domain framework are possible, most developers typically focus on modeling. Domain-specific models are then used to directly generate the code. In DSM, we should avoid situations in which models are translated to other models for future editing. This approach did not work in the past with code and it won't work with models either. Use internal to a tool as an intermediate product is a different story. But you should not modify the results of the transformation or generation process. This is discussed in more detail with guidelines for DSM definition in Section 10.5.

Every model is based on some implicitly or explicitly defined language. Generally speaking, models expressed in DSM have similar characteristics as other models. There are some notable differences, though. In DSM, the modeler works by using the concepts of the domain. These are given by the language definer. As discussed in Chapter 3, in DSM models are also formal, based on higher abstraction than coding, follow the rules of the domain, and are based on concepts familiar to the developers working in the domain the language targets.

4.3.1 Model is a Partial Description and Code is Full?

A model is a description of something. Usually a single model, like a diagram, represents a selected view of a system, a program, or a feature. To specify the complete system, we need several models and modeling concepts that specify different aspects of the system. Some may argue that a model is just a simplified description of the system and that code made with some programming language is closer or better at describing the reality. While this may be true for modeling languages that don't carry enough information to specify running systems, it does not hold for models made with DSM.

In DSM, models are formal and backed by a code generator, domain framework, and underlying target platform. These provide the necessary lifting work to make models first class citizens so that models are an adequate specification to develop complete systems from a modeler's perspective. This is nothing new compared to manual coding: programmers easily forget that compilers, linkers, and libraries actually make manual coding possible. By following this analogy, we can see that C code is just a partial model for a compiler developer.

4.3.2 Working with Models

Use of a modeling language naturally leads to working with models. And in true model-based development, we end up having a lot of models. A “lot” is not necessarily the best word here as there is clearly less specification work in models when compared to manual coding, or visualizing code with UML or other modeling languages originating from the coding world. This is simply because the abstraction in DSM is always at a higher level. But we still end up having more models than we usually may have experienced.

In DSM, the role of models in the development process changes:

- **Models are versioned, not code.** In DSM, models are now the source—first class citizens which we don’t throw away as the development progresses. We version the models. As the code can be generated at any time, there should be no need to version source code separately. We don’t version object files either during C compilation. However, we may still save the code if the other development tasks so require, for example, if the build process can’t execute code generators.
- **Modeling and generators cover some tasks that earlier belonged to testing.** We do more tasks related to testing at the design stage. This is simply because a proper modeling language knows the given domain and ideally does not allow making designs that are illegal or would lead to poor performance. Supporting testing right at the modeling stage is important because it is far cheaper to prevent errors early during design.
- **Debugging is done largely at the model level.** If the specified functionality is not working as expected, the trace from execution can be provided in the model instead of debugging on the lower level of generated code.
- **Models can be used for communication.** Models expressed in domain concepts are used more for communication as we can trust the models; they are not separate from the actual implementation. Having domain concepts directly in the language makes models easier to read, understand, remember, and verify than modeling the domain using programming concepts. Compare, for example, the difference in Chapter 1 between the class diagram and sequence diagrams, and the mobile phone DSM model.
- **Models are the input for multiple different artifacts.** Models not only are the input for generating code but also can be used for many other purposes, such as producing documentation, test cases, and configuration data. What later stages are used depends on the objectives. Some of the generation targets are discussed in Section 4.4.
- **Models can be expressed in different representational styles.** Depending on the model user, visualization needs, and analysis needs, models can be expressed in different representational styles, such as graphical diagrams, matrixes, tables, forms, or text.
- **Modeling with DSM is agile.** Compared to heavy up-front specification work before the actual implementation starts, DSM is agile: Only those aspects are modeled that are relevant. With code generation, we can quickly get response

and feedback to the models. If desired, the modeling language can be designed particularly to add agility to the development. For instance, a modeling language can first be used partially, to produce a prototype to review the functionality of the product. Later, the same models can be extended to finalize the specification and generate production code, possibly into a different programming language from that used for the prototype.

4.3.3 Users of Models

Often we focus on making models for code generation, so the typical language users are application developers. In DSM, the possible user base of the models can easily be broader: A higher abstraction level and closer mapping to the domain allow customers and other end users to be better involved in the development process. They can read, accept, and in some cases even change the specifications. This is very important since the success of the project is often directly related to the level of customer involvement. DSM allows people other than software developers to create specifications. Domain experts, who often don't have software development background, can specify applications for code generation. The insurance case discussed in Chapter 6 belongs to this category: insurance experts use models to specify insurance products and generate Java code for a the web portal.

Since the role of models changes, the border of requirements and implementation can also change. For example, domain experts can specify models for concept prototyping or concept demonstration and application developers can then continue from these models. The work can then be based on using other languages or by extending the existing models with additional details for implementation.

A DSM solution can also be built for a user group other than traditional application software developers. One class of DSM use targets test engineers: they create models that produce test cases, test scripts, or programs that run the tests. Another group is the configurators who handle product deployment, installation, and service. They can work with models that apply concepts directly related to specific characteristics of configuration, like specifying deployment of software units to hardware or describing high-availability settings for uninterrupted services with redundancy and reparability for various fault-recovery scenarios. Yet another group of model users is those specifying services that are then executed in the target environment. For example, the Call Processing Services case in Chapter 5 describes how service engineers can specify IP telephone services using domain-specific models. Dedicated DSM solutions can also be built for architects specifying the application architecture within a specific application domain. For example, languages like AADL (SAE 2004) and AUTOSAR (2006) component and runnable diagrams target the general architecture of automotive applications.

4.4 CODE GENERATOR

In DSM, code generators transform the models into code for interpretation or compilation into an executable. By providing automation, they contribute to the

productivity and quality gains of the DSM approach. The generated code is typically complete from the modeler's perspective. This means that the code is complete, executable, and of production quality; in other words, after generation, the code needs no manual rewriting or additions. This is possible because the generator (and modeling language) is made to satisfy the demands of a narrow application domain—used inside one company. We must emphasize that this does not mean that all code used is generated. That's why we have a domain framework and a target environment. They may be generated from different models or, as is most likely today, programmed manually. The generator itself, like the domain framework and target environment, can be largely invisible to the developers in the same way as black-box components or compilers are not visible.

Code generators can be classified differently and perhaps the most used is dividing them into declarative and operational, or a mixture. This classification is based on the approach used to specify generators. In the declarative approach, mapping between elements of the source (metamodel) and target programming language is described. Operational approaches, such as graph transformation rules, define the steps required to produce the target code from a given source model.

Although it is possible to view and edit the generated code in DSM applications, developers usually do not need to inspect the results of the generator. Editing generated code is (or should be) analogous to manually editing machine code after C compilation, which typically is unnecessary. In DSM, modifications are made to the models, not to the generated code, which can be treated simply as an intermediate by-product. That has been the recipe of success for compilers, and code generators can achieve the same objective. How to specify these generators as a part of your DSM solution is described in Chapter 11.

4.4.1 Generator Principle

Basically, a code generator is an automaton that accesses models, extracts information from them, and transforms it into output in a specific syntax. This process depends on and is guided by the metamodel, the modeling language with its concepts, semantics and rules, and the input syntax required by the domain framework and target environment. We introduce the role of domain framework and target environment later in Section 4.5.

Accessing Data in Models Generators access the models based on the metamodel of the language. They can start navigation based on a certain root element, seek for certain object types, or be dependent on the various relationship types and connection types the models have. Even more navigation choices are available if the generator uses instance values, like choosing based on a certain value which model elements to access next. A generator can further navigate connections or submodels, depth- or breadth-first, or apply some order for navigation and access.

While usually most of the model data for accessing and navigating models are the same as the design information, additional model data also can be used. These can include:

- spatial location of model elements, such as size, or location relative to other elements;
- model administration data, such as creation time, version, or author;
- model annotations that guide the generator but are not needed for finding a solution in the problem domain. These can include selection of target environment, compiler, and output directory.

If there are multiple models based on different languages, navigation can still be based on the same principles as accessing just one model. If the languages use integrated metamodels, a generator can then treat separate models as integrated. If models are treated separately from each other, the generator integrates the models during generation, for example, by using string matching or annotations in the model that show links between models.

Extracting Model Data While navigating in the models, the generator extracts design data and combines it with a possible domain framework. Again the code generator can only retrieve information from models that was provided for in the metamodel. In the simplest case, each modeling element produces certain fixed code that includes the values entered by the modeler. Generators can also extract data by analyzing combinations of model elements, such as the relationships connecting them, the submodels an element has, or other linkages between model elements.

Transforming Models to Output Code While navigating in models the data accessed are combined for the purpose of code generation. Here the generator adds additional information for the output as well as integrating with the framework code or making calls to the underlying target environment and its libraries. Consider the generated C++ code in Listing 4.1 below. The code is generated from the mobile phone application design described in Fig. 1.6.

Listing 4.1 Sample code for the application described in Section 1.3.

```

01 // -----
02 // void CAknConferenceRegistrationView::welcome()
03 // -----
04
05 void CAknConferenceRegistrationView::welcome()
06 {
07   CAknInformationNote* aNote;
08   aNote = new (ELeave) CAknInformationNote(ETrue);
09   aNote->ExecuteID(_L("Conference registration: Welcome"));
10 }
```

Typically a generator produces the syntax for the generated code: C++ here, but it could equally well be some other target language, like Python, as discussed in depth in Chapter 8. The generator also transforms model data, and sometimes also metamodel data, to the output code. The notification type “Information” in lines 7 and 8 and the note text “Conference registration: Welcome” in line 9 are taken from the properties of a model element. These define the text be shown and set the note element type and icon to

be shown to be “Information”. Similarly, “welcome” is the name given for the note element and the “ConferenceRegistration” is taken directly from the name of the diagram. A developer has specified all these values either by selecting among existing notification types or by entering the values. The rest of the output is produced by the generator. Most notable is the call to the Symbian and S60 UI framework in line 9.

The structure of the generated code is then dependent on the requirements of the implementation. The examples in Part III show different kinds of structures for code output, such as serialization, function calls, case switches, and transition tables. Here with the C++ code the output follows exactly the same structure that the tutorials on C++ use for Symbian application development, or that experienced developers in a company use when writing the code manually.

A code generator can also use a translation mechanism to change the information entered into the models into a format applicable in the implementation language. A typical case is removing or replacing spaces from values given in models that are used as variable names in the generated code. The string “ConferenceRegistration” could then have been entered in the model with a space between the words. Also, if the generated programming language uses some specific conventions for naming variables, classes, operations, and so on, the generator can use translations for them, like starting names with a capital letter.

4.4.2 Quality of Generated Code

Varied opinions exist concerning what kinds of code one can generate and with what level of quality. For example, automation to produce static declarative definitions from common designs such as interfaces or database schemas has been a reality for many years, so multiple off-the-shelf generators are available. However, the situation is different when it comes to generating behavioral, functional code. Consider the use of UML in the phone example in Chapter 1. It required extensive and detailed UML models to specify the behavioral side—yet still not adequate in detail for code generation. In DSM, the code generation can tackle both the static code and the behavioural, functional code. Since static structures such as schemas, interfaces, and declarations are usually easier to generate, we focus in this book mostly on generating behavioral and functional code.

Can We Trust the Generated Code? Many developers have had bad experiences with third party generators because the generator vendor has fixed the method of code production. Despite the existence of multiple ways to write code for a certain behavior, the vendor has chosen just one of them. The vendor’s chosen way is not always likely to be ideal for your specific contingency, taking into account the target language generated, programming model used, memory used, and so on. Third-party generators often don’t know enough about an organization’s specific requirements to generate ideal code, so it is not surprising that many have found generated code unsatisfactory. Because modifying the generated code is usually not a realistic option, organizations end up throwing away the generated code. The value of the generated code is then limited to prototyping and requirements gathering.

Because of these disappointing experiences, developers sometimes have little confidence in generated code. This lack of confidence, however, changes radically when developers are asked if they trust generators they have made themselves. Not having to give up control of the code generation process, from design to output format, to a faceless tool vendor makes a big difference in the acceptance of generated code. Here DSM changes the rules: an experienced developer, usually within a company, defines the automation process and output for the rest of the developers in that team.

Is the Generated Code Efficient? The major argument against generators is the claim that the generated code cannot meet the strict requirements of size and runtime efficiency that are fundamental issues when developing software for devices with limited memory and processing resources. When comparing generated code and manually written code, we should not forget that the compiler performs further optimization. In one application area, a company conducted an analysis of hand optimization in speed and memory use in the assembly language produced by a compiler. After careful analysis of the code, the comparison team did not find any substantial differences. This study was conducted in an embedded area where code size and memory use matter. While you may have bad experiences with the quality of code produced by conventional code generators, they are not valid for DSM. You now have control and can update the generator if needed. Several examples, presented in Part III, especially target embedded software development with relatively strict requirements for the code.

When code is produced by a generator made by an experienced developer, it will always produce better code than the average programmer writes manually. The memory management, optimization, programming model, and styles are applied consistently. Ideally, the generated code should look like the code handwritten by the experienced developer who defined the generator. The generated code can also follow other requirements, like corporate coding standards. These may feel important in the beginning but become less relevant when the shift to using DSM has happened. The format of the generated code then matters most for the generator developers who need to debug and check the code generated during generator test runs.

4.4.3 Different Usage of Generators

In this book we focus on producing production code, although generators can be used for many other purposes too. If we can generate production code, then other kinds of outputs can be produced as well. One obvious extension of this is automating the generation process in which the code generator also creates build scripts, calls a compiler, deploys the generated code, and executes it in a target environment. It is even possible to generate traceable code that includes a link to the models: running the application then allows to trace back to models to visualize the execution. Below we consider other kinds of generators.

Model Checking In DSM, generators are also used for checking the consistency and completeness of designs. This is needed because it usually does not make sense, or is not even possible, to put all the rules in the metamodel and check them during each modeling action. This is especially true when checking partial models, when there are multiple models, or when integrating models made by different developers. Generators for model analysis can also be used for guiding modeling work and informing about actions needed. A typical such scenario is to look if a model, or models, is incomplete and report possible actions needed to make the model complete. Such model checking can be run similarly to generators: when needed or after conducting certain modeling actions.

Metrics When moving toward model-based development, code-based metrics can still be applied; they are now calculated from the generated code instead of from the manually written code. As platform code is already available, the metrics may also cover platform functions or libraries used instead of focusing only on the generated application code. Use of code metrics based on models is easy as it does not require much change to earlier practices, but it is not likely to be the most effective use of metrics.

In DSM, code-based metrics no longer measure the amount of human work needed. Metrics, like function point analysis (FPA, Albrecht and Gaffney, 1983), to analyze program size and to estimate required development effort are no longer relevant: the application is often already ready when we can calculate these metrics. DSM uses metrics that are based on the specific domain and thus uses the metamodel data. For example, a metric for software used to control production processes in a paper mill can calculate valves, motors, pumps, and their characteristics rather than produce general metamodel-independent metrics on system complexity (e.g., cyclomatic complexity by McCabe,) or program length (Halstead, 1977).

Prototyping and Simulation Generators can also be used to produce prototypes instead of production code. Prototyping is usually used to give early feedback, and generators may produce code for a totally different target platform and in a different programming language than the production code. Here generators don't necessarily need to optimize the code, but to enable functionality, usability, look and feel or other characteristics relevant to prototyping. The modeling language, however, can be the same for developing both the prototype and the production code. The model can also be used for simulation: code generators then provide output in a syntax required by a simulator.

Configuration, Packaging, and Deployment Generators can be limited to producing component glue code, such as integrating existing components together. When seeking better automation, a more typical case is to produce configuration data and other packaging information for application deployment and installation. In other words, the application and its installation configuration can be generated

simultaneously from the same models. This reduces the work needed as well as making the process safer and easier to repeat.

Documentation Generators can also be used to produce documentation, inspection reports, management status reports, and so on, from the same source models from which the actual code is produced. The obvious benefit is that there is no need to manually update documentation to keep it up-to-date with development. It is also worth noting that the generated documentation is not only about the implementation but also covers the solution described in domain terms. After all, in DSM, models specify mostly the problem domain, not the solution.

Testing and Test Suites Models can also be used for testing, especially formal models like DSM; they rarely have other models' problems of inconsistent information or inaccurate specification of the product. How could they if the product can be generated from the same models? The domain rules in the language make many normal tests redundant simply because they are already "tested" in the models. Similarly, automated production of code wipes out many typical errors found in manual coding, such as typos, missing values, or incorrect references. Models used to generate code are usually poor sources from which to generate tests: would such tests really prove anything? Data from models can however be used to select from existing tests those that are applicable to the developed system. This is especially relevant for large systems where thousands of tests can exist. DSM can also be used specifically for testing: test engineers can define tests using testing and product concepts to generate test cases and applications running the tests.

Language Use and Refinement Information We can also shift the focus from reporting about models to reporting about metamodels. Generators can produce information about how the language and generator are used. This helps to reveal patterns of language use, which concepts are not used, which models use older version of the language, and so on. Modeling languages can also include some elements with open semantics that provide possibilities for modelers to express things that can't be captured with the current language. A generator can report on possible uses of these concepts to find out where users of the language find it lacking and identify areas for further improvement. We discuss this more in Chapter 10 together with language evolution and maintenance.

With multiple generators, we start to enjoy having a single source but multiple targets: developers need to change only one place and the generators take care of the rest. Generators can also be combined so that they are not executed alone but in relation to others. For instance, a model checking report can be executed automatically before generating the code. If errors are found, the actual code generation can be stopped. Similarly, generated documentation can include information produced by a metrics report. Another kind of generator, although rare in practice, is a generator that aims to produce other generators.

4.5 DOMAIN FRAMEWORK AND TARGET ENVIRONMENT

A domain framework provides the interface between the generated code and the underlying target environment. It is not needed in all cases though: for example, if the generated code can directly be the input for some engine or call the interface of the target environment. More often, though, a DSM solution uses some framework utility code or components to make the generated code simpler.

4.5.1 Target Environment

Before inspecting the domain framework let's first look at the underlying target on which our generated applications run (Fig. 4.7). We may view the target environment as consisting of different infrastructure layers. We need at least some of these layers regardless of whether our applications are programmed manually or generated automatically. All these layers exist partly because they improve developer productivity. However, they also increase the level of abstraction from the bottom up, on the implementation side, whereas in a DSM language, the level of abstraction is also raised on the problem domain side.

At one extreme, the target environment can be plain hardware, but more typical are those with a component framework, a library, or even off-the-shelf engines or standalone programs, like messaging servers or databases. Which of these are used depends on the case.

The support provided by programming languages, virtual machines, or operating systems is usually leveraged in a similar way to libraries. They are used by the created application. Libraries provide predefined building blocks but do not enforce a particular design for an application. A framework can be considered to be more than a library. It is an abstract specification of a kind of application. It expects a certain programming model that the developer must follow. Frameworks are generally classified as white box or black box. In a white-box framework, the user adds

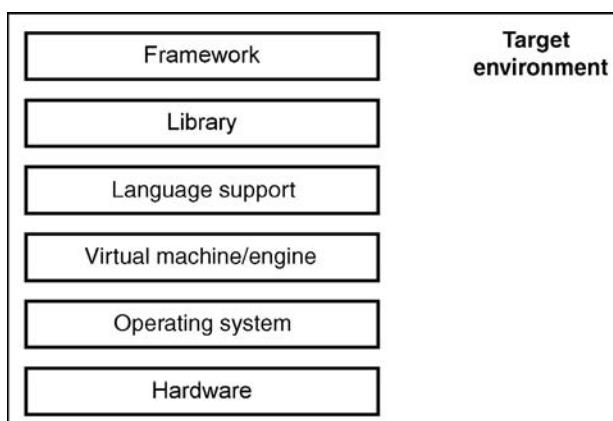


FIGURE 4.7 Layered architecture of a target environment

functionality to existing code. This necessitates that the implementation details must be understood at least to some extent. In a black-box framework, existing building blocks are applied as such and their internal implementation is not visible. Separation between libraries and frameworks can sometimes be difficult since they form a continuum: sometimes a library can provide more sophisticated support for certain application functionality than a framework.

In DSM, all these possible layers of a target environment are invisible to application developers. They are not even black-box components since modelers don't need to know anything about them. DSM hides them yet makes their use automatic. However, the experienced developers creating the DSM solution need to know them well. In Chapter 12, we discuss in detail how the target environment and existing code can be integrated with models and generated code.

Target Environment Already Narrows the Focus For DSM, a third-party target environment is often not enough; it is usually too generic. We are not building all the different kinds of applications we can run on a PC, J2EE, or .Net: we are always more focused. Let's consider the case of developing mobile phone applications as illustrated in Chapter 1. As a target environment the Symbian operating system alone is too generic as we are building a certain kind of application. So we have a narrower domain. It could be gaming, browsing, camera, or enterprise applications as in Chapter 1. Depending on the choice, there are different levels of language support, libraries, and components or even existing frameworks, all providing a set of prefabricated software building blocks that programmers can use, extend, or customize for specific systems or applications. For a game we could use an existing engine and for enterprise application we could use an existing workflow server that is not running in the phone but is accessed via the network.

With the phone case, we could focus on developing personal productivity applications, like a phonebook or a diary. Even then we would still have different options, such as UIQ or S60 frameworks. Both of them provide additional components to the Symbian OS for application development. These frameworks also expect different programming models and limit our choices of a programming language. Now we have restricted our focus to a narrower domain and creating a DSM solution becomes possible. Even at this level, more choices are available to raise the level of abstraction and narrow the focus on the implementation side. For instance, we could focus on implementing our applications with Java MIDP, C++, or Python. These implementation languages may have their own libraries with which a DSM solution may integrate. In Chapter 8, we describe one mobile application case in detail, targeting both Python and C++ libraries.

4.5.2 Framework Code and Generated Code

In most cases, the optimal way to improve the use of a target environment is to build an adequate domain framework on top of it. This domain framework can range in size

from components down to individual groups of programming language statements that commonly occur in code in the selected domain. They are also usually developed manually, similar to libraries and frameworks for the target environment. Code for the domain framework, however, has different purposes:

- **Remove duplication from the generated code.** Applications tend to have similar structures that are specific to the type of application we are building, yet not provided by the target environment. Rather than including these in the generator, they can be made into domain framework code that generated code calls. This keeps the generator simpler. For instance, it could be that every application or feature needs the same data structure or has similar behavior.
- **Provide an interface for the generator.** The domain framework defines the expected format for the code generation output. The output is not defined as concrete code but more as an example or template that the generator then produces. In a simple case, when generating XML we could consider the schema to define the structure for the generated code. In a more complex case, the domain framework can define data structures that generated code then fills.
- **Integrate with existing code.** Rather than directly calling the services of the library and its interfaces, a domain framework may be used to integrate with existing code. For instance, framework code may provide basic behavior as abstract classes and the generated code creates the subclasses or implements stack management that the generated code uses.
- **Hide the target environment and execution platform.** The domain framework can be used to support different implementation platforms. The models and generated code can then be the same and the choice of domain framework decides the execution platform. For such a case see Chapter 9, in which Java code can be executed in applets or midlets (MIDP Java for mobile phones) by choosing the right framework code.

We describe how to implement domain frameworks in more detail in Chapter 12. It is important to note that the domain framework is not necessarily an extra burden required only by the code generator. Actually, in most cases the underlying software architecture already utilizes various libraries, components, or other reusable parts that can also support the generated code.

4.6 DSM ORGANIZATION AND PROCESS

DSM distinguishes two different roles in the development organization: those creating applications with DSM and those developing the DSM solution. This separation is nothing new: we find it applied already in many companies. In component-based development, some people make components, or whole organizational units can act as component factories, and other people use those components to create applications. Similarly, in product line engineering, some people make the

common platform for all projects, and some develop products using the assets of this common platform. Software development is moving from the idea of having generalists to that of having specialists, similar to other development organizations and industries.

4.6.1 Organization and Roles

This separation of two roles in DSM does not mean that the people are necessarily different too. Usually those developing the DSM solution are also using it, at least to some extent. What is crucial is that the more experienced developers are making the DSM solution. Experienced developers can obviously specify the automation in terms of languages, generators, and domain frameworks better than those less experienced. They also have the necessary authority among their colleagues.

In DSM, we can identify the following roles:

- **Domain experts** are people who have knowledge about the problem domain—the area of interest where DSM is applied. They know the terminology, concepts, and rules of the domain and often have actually created them. Application developers also qualify here if they have developed multiple similar kinds of applications in the past; creating just one is usually not enough as there might not be enough domain expertise to make generalizations to find higher abstractions. When developing business applications, such as the insurance product portal described in Chapter 6, domain experts are insurance experts and product managers. In technical domains, like the mobile phone applications discussed in Chapter 8, domain experts are the architects and lead developers of the target environment.
- **DSM users** apply the modeling languages. This group is usually the largest. Among DSM users, the most obvious subgroup is those creating models to generate applications, but other subgroups of users also exist. Models that operate at a higher level of abstraction can be used widely in supporting communication, for example, with test engineers, product managers, Q&A, deployment engineers, sales, and customers. In addition to pure communication aid, test engineers can use the models to plan their test cases and produce test suites. Deployment engineers can produce installation programs and product configurations using the DSM models. They will use different generators but the models will be the same. Also, managers can get reports and metrics on the status of the development from the models.
- **Language developers** specify the modeling language. They formalize it into a metamodel and provide guidelines on its use, such as manuals and example models. Language developers work closely with domain experts and with key DSM users. Usually, just one or two people are responsible for the language specification, especially if metamodel-based tools are used when creating the DSM solution. Such tools can automatically provide the necessary modeling tools, and then their implementation does not require traditional programming for tool development. They allow domain experts to easily participate in DSM

creation by using the modeling languages in development work. Chapter 10 gives detailed guidelines for language specification.

- An **ergonomist** can help the language developers improve the usability of the language. While usability is always relevant, this special role can be particularly significant in certain cases. For example, when creating a DSM solution for UI (user interface) or HMI (human-machine interface) applications, it can be important for models to correspond closely to the actual product. The person who defined the UI styles can then also support the language developers. The role of the ergonomist can also be apparent if the language is used by nonprogrammers or if the users span multiple continents and cultures.
- **Generator developers** specify the transformations from models to code following the formats and reference implementations given by architects and developers of framework code. Often the generator developers are the same people as those defining the domain framework. Chapter 11 gives guidelines for generator development as part of a DSM solution.
- **Domain framework developers** are usually experienced developers and application architects. They can provide reference implementations and can specify how the target environment should be used. Typically they are the people who are already making reusable assets, like component frameworks and libraries. In Chapter 12, we give guidelines on defining the domain framework code.
- **Tool developers** implement the modeling languages and code generators. Depending on the tools used, this group may not be needed at all since modern metamodeling tools provide modeling editors and generators automatically from language and generator specifications. If automation and proper tooling are not used, it is usual to need more than five people to implement the tooling for DSM. Creators of DSM solutions should thus use automation for their own work too. We discuss tooling for DSM in detail in Chapter 14.

The above list does not distinguish the developers of the target environment and reusable components or customers and managers. They exist regardless of whether the DSM approach is used. Similarly, application engineers and architects already exist but now their work changes in part. Most application engineers can now apply higher-level models to create and maintain applications. Architects can now formalize the rules to be followed into a DSM solution and thus be more certain that they will also be followed.

Although the above list is extensive, most roles dealing with DSM creation are handled by just a few people. In fact, it is not exceptional that a single person defines the language, generator, and domain framework. In practice, several people participate somewhat even then by giving requirements and testing the DSM solution. There are several reasons for this. It is unlikely that a single person can find the right abstraction and develop the necessary architectural elements for DSM. A larger development team also makes sense from a human resource point of view: the DSM

solution becomes one of the most valuable assets of the organization and it is good that several people understand it intimately.

4.6.2 DSM Definition Process

The DSM process can perhaps best be viewed as consisting of four main phases: initial development, deployment, use, and maintenance.

The Proof of Concept As the domain-specific approach is often being applied for the first time in the company, it may be necessary to first demonstrate the feasibility of DSM as an approach. For such a demonstration, a narrow and well-known area is usually selected so that the proof of concept can be done quickly. Also, the DSM solution is not necessarily large but just large enough to give a concrete demonstration within the domain. The concept demonstration also helps in identifying candidate domains and in eliciting requirements for the actual project implementing a DSM solution.

The Pilot Project When the feasibility of the DSM approach has been verified, a pilot project follows. In the pilot project, the very first version of the DSM solution is defined, implemented, and tested. At this stage, DSM users get more involved and some of them also use the created solution in a representative example of a real-life project. The pilot helps in estimating the effects of DSM and prepares for introduction in the organization, such as making tutorials and sample designs, training material and tutors with some experience of DSM use.

Use of the DSM Solution The deployment of DSM occurs when it is adopted in full production. From here on, the company starts obtaining the benefits of a higher level of abstraction and automation on a larger scale as discussed in Chapter 2. Also, the role of DSM grows here with increasing use of models and generated code.

DSM Maintenance The DSM solution seldom stays fixed; it needs to be updated when requirements change and as the organization finds better ways to use the DSM solution. Typical cases are generating a larger body of code or using generators for other kinds of outputs too, such as metrics, simulation, and so on. The developers of the DSM solution update and share the related languages, generators, and domain frameworks to modelers until they remain static. Then the DSM solution is reaching the end of its life cycle, typically because the applications are no longer actively developed, just their bugs are corrected. Most likely there is a new domain or the target environment for the application has changed beyond the reach of the original DSM solution.

4.7 SUMMARY

In DSM, there is no single place to raise the level of abstraction to provide automation. Rather it is the task of experienced developers to divide the work between modeling languages, code generators, and a domain framework.

For application developers, modeling languages are the most visible part of DSM: they provide abstractions that are suitable for problem solving in the domain. Generally, the major domain concepts map to modeling language objects, while other concepts will be captured as object properties, relationships, submodels, or links to models in other languages. The specification of the modeling language, a metamodel, also covers rules. Rules are relevant to guide modeling work, prevent errors early, and make the models created suitable for code generation. A language also needs to have a notation. Here DSM tries to mimic representations of the true domain whenever possible, making models more acceptable and easier to create, read, remember, understand, and maintain.

Generators transform the models into code for interpretation or compilation into an executable. A generator basically defines how model concepts are mapped to code or other output. In the simplest case, each modeling symbol produces certain fixed code that includes the values entered into the symbol as arguments by the modeler. Generally, generated code is linked to existing libraries and other code available in the target environment, as often all such lower-level code for the application cannot and does not need to be generated from the domain-specific models.

A domain framework is often created to make code generation easier. The domain framework provides a layer between the generated code and existing code in the target environment. It provides code that helps avoid repetition in the generator (and models) and minimizes the complexity of the generated code.

PART III

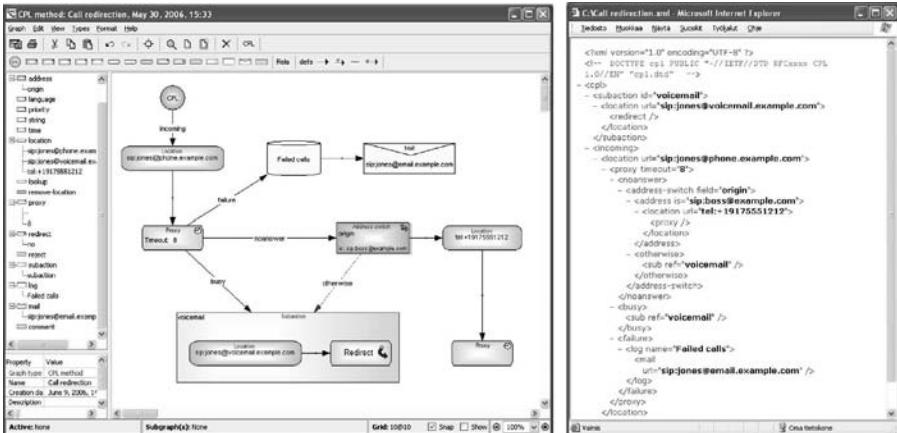
DSM EXAMPLES

In this part of the book, we illustrate Domain-Specific Modeling (DSM) examples from practice. Following the DSM architecture, we describe for each case the language definitions, how the code generators work, and how the services of the underlying platform and domain framework are used. Later, in Part IV, we refer to these examples to demonstrate guidelines for DSM implementation.

Every domain is different, and so every DSM example is different. We have chosen five examples that cover different problem domains and generation targets. The problem domains range from insurance products to microcontroller-based voice systems, and they illustrate modeling languages based on different models of computations. The generation targets cover the whole spectrum from Assembler to Java and XML; some use a purpose-built component and domain framework whereas others don't use any supporting framework.

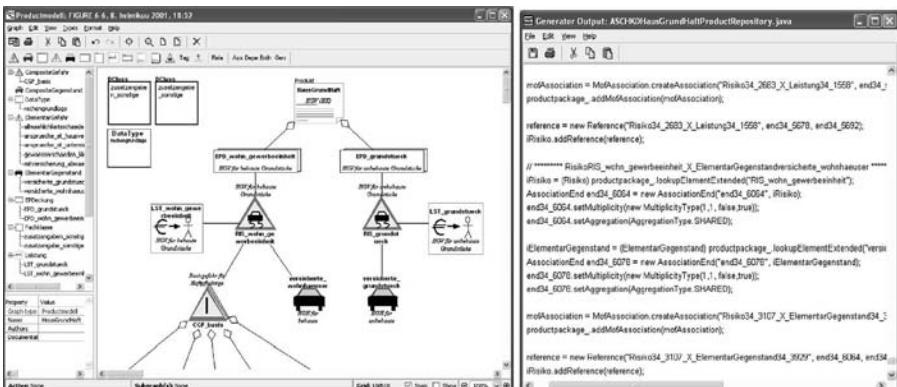
For the purposes of this book, we selected examples that are easy to understand and grasp completely in a limited space. Although we have been working with larger domain-specific languages, some having twice as many concepts as UML, showing just the parts of these that would fit would not show the whole of DSM. The principles described in this book also scale to large DSM solutions. For the sake of readability, we also selected application domains that are relatively well known. Below is a summary of the cases showing example designs with the domain-specific language and part of the generated code.

Chapter 5: IP telephony and call processing



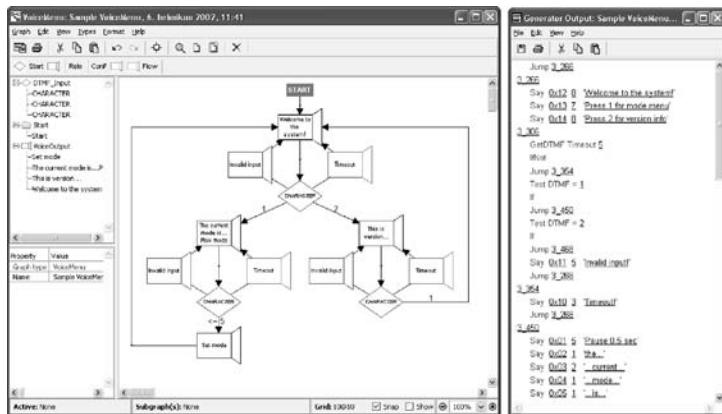
We start with the case of a language based on an XML schema. Chapter 5 illustrates a case for service creation: describing IP telephony services using flow models and generating a service description in XML. A service engineer draws models like the above to define different telephony services, and then the generator produces the required service descriptions in XML for execution in a telephony server. From a language creation perspective, this example is the best to start with as it is closely related to an XML schema that almost completely defines the language.

Chapter 6: Insurance products for a J2EE web site



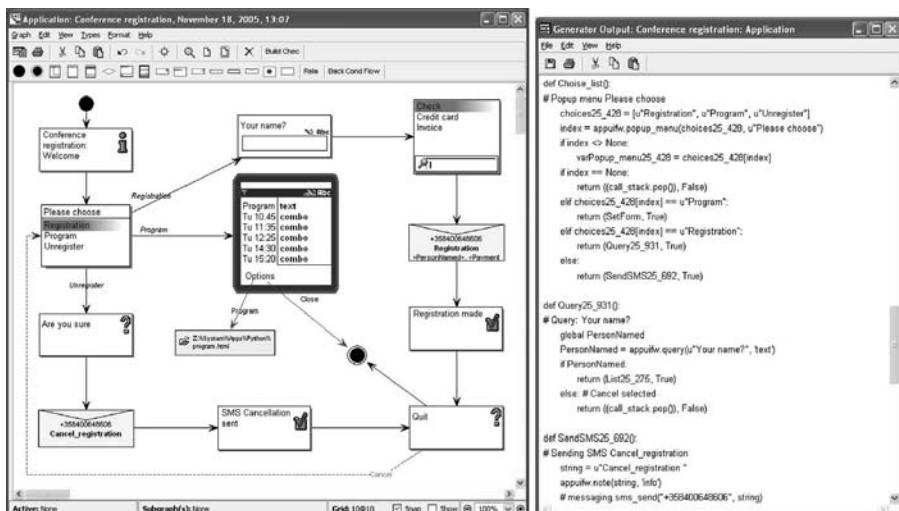
The second example illustrates a case of capturing insurance products using static declarative models. An insurance expert, a nonprogrammer, draws models to define insurance products, and then the generators produce the required insurance data and code for a J2EE web site. As the generated code covers only static aspects, it is perhaps a good place to start for those used to generating database tables or class stubs.

Chapter 7: Microcontroller applications specified in 8-bit assembler



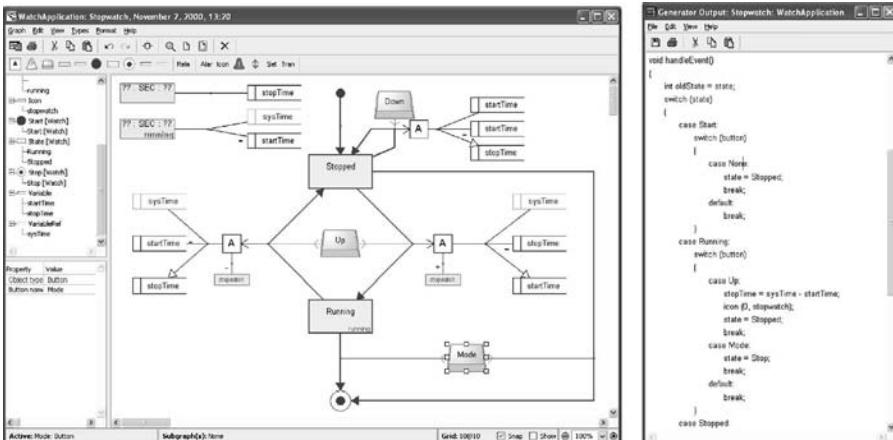
Chapter 7 shows a case developing a voice menu system for an 8-bit microcontroller. The models show the flow-like execution of the menu system. The generator produces assembly language code for memory addressing, calculation, and operations needed within the voice menu domain.

Chapter 8: Mobile phone applications using a Python framework



This example illustrates application development for mobile phones. DSM uses the widgets and services of the phone as modeling concepts, following the phone's UI programming model. The generator produces full code as Python functions. The generated code calls the phone's platform services provided via an API and executes the result in an emulator or in the target device.

Chapter 9: Digital wristwatch applications in Java/C



Chapter 9 describes state machine based Java and C code generation for embedded devices using a familiar domain, a digital wristwatch, as an example. This case describes how product line development can be supported by modeling variation among products. It also shows how different kinds of languages, static and behavior, can be integrated during modeling and used when generating code from multiple different kinds of design models.

All these cases applied fully model-based development: the models created form the input for code generation. Thus, the DSM language was created not only to use models to get a better understanding, support communication, or create documentation, but also to automate development with domain-specific generators. Actually, in all of the cases the generators, together with the supporting domain framework, aim to provide full code from the modelers' perspective. The code produced is fully working and also covers the application logic and behavior, not just the static structures, which are usually easier to model and generate. The only case where models are used to capture only static structures is the case of insurance products.

To describe the modeling languages consistently and precisely, we will use the metamodeling language documented in Appendix A.

The modeling languages, generators, and example models from Part III can be downloaded from <http://dsmbbook.com> along with a free MetaEdit+ demo.

CHAPTER 5

IP TELEPHONY AND CALL PROCESSING

In this chapter we describe DSM solution made for specifying Internet telephony services. Recently multiple protocols, such as SIP (Rosenberg et al., 2002) and H.232 (ITU, 2003), have been defined to provide a session initiation protocol for telephony over IP networks. These protocols also offer the possibility to decentralize the control of user-specific call processing services. In traditional telephony systems, network-based services were created only by the service providers, such as operators. Service creation has required special knowledge of the telephony system in use and use of a variety of often proprietary tools, and most importantly many of the user-specific service customizations were simply not available. Internet telephony changes this as protocols are open and allow even telephony users to define their own customized services. Here, a DSM solution is created to allow easy specification of call processing services using telephony service concepts.

5.1 INTRODUCTION AND OBJECTIVES

One of the major changes IP-based technology offers for telephony is call processing applications. Many of the telephony services typically depend on a user's status and their creation does not necessarily require intermediary organizations. In fact, the cost of their production would be less if they were made closer to their user. To illustrate the

application development, let us look at some of the typical telephony services illustrated by Lennox and Schulzrinne (2000):

- Call forwarding if the receiver is busy or does not answer: For instance, when a new call comes in, the call should ring at the user's desk telephone. If it is busy, the call should always be redirected to the user's voice mail box. If, however, there is no answer after four rings, it should be redirected to the user's voice mail unless it is from a supervisor, in which case it should be proxied to the user's cell phone if it is currently registered.
- Information address service: An example service could be made for a company that advertises a general "information" address for prospective customers. When a call comes in to this address during office hours, the caller should be given a list of the people currently willing to accept general information calls. If it is outside of working hours, the caller should get a web page indicating what time they can call.
- Intelligent user location: For example, when a call comes in, the list of locations where the user has registered should be consulted. Depending on the type of call (work, personal, etc.), the call should ring at an appropriate subset of the registered locations, depending on information in the registrations. If the user picks up from more than one station, the pickups should be reported back separately to the calling party.
- Intelligent user location with media knowledge: One service could be that when a call comes in, the call should be proxied to the station the user has registered from whose media capabilities, such as video call, best match those specified in the call request. If the user does not pick up from that station within a specified number of attempts, the call should be proxied to the other stations from which the user has registered, sequentially, in order of decreasing closeness of match.

The call processing framework and language (Lennox et al., 2004) present an architecture to specify and control Internet telephony services such as those described above. The language part is of interest to us here. The purpose of the Call Processing Language (CPL) is to be powerful enough to describe a large number of services and features but at the same time to be limited in its power so that it can run safely on Internet telephony servers. The limited scope makes sure that the CPL server's security will be ensured. Looking from the DSM creation side, the domain is well restricted and the requirement of complexity hiding is highly relevant. Also, generators offer obvious advantages here as quality and correctness of services are of great importance. Nobody wants to miss a call just because the service definition had errors.

A computational model of a CPL specification is a list of condition and action pairs: if the system condition matches a condition specified in a service, then a corresponding action or actions are performed. A typical system condition in CPL is, for example, that a call arrives and the line is busy or it is a certain day of the week. Example actions redirect the call to the user's mobile phone or reject the call.

If conditions are not specified in the CPL specification, then the server's standard action is taken.

5.1.1 CPL Architecture

Architecturally, a call processing service is executed in a signaling server. Signaling servers are devices that relay or control signaling information. In the SIP world (Rosenberg et al., 2002), examples of signaling servers are proxy servers and redirect servers. A signaling server also normally maintains a database of locations where a user can be reached. A call processing service makes its proxy, redirect, and rejection decisions based on the contents of that database. A CPL specification replaces this basic database lookup functionality; it takes the registration information, the specifics of a call request, and other external information it wants to reference and chooses the signaling actions to perform. To put it simply, CPL describes how devices respond to calls and how a system routes calls.

Services in CPL are normally associated with a particular Internet telephony address. When a call arrives at a signaling server that is a CPL server, the server associates the source and destination addresses specified in the request with its database of CPL services. If one matches, the corresponding CPL service is executed.

At this point, we should emphasize that creation of call processing services is not bound to any specific user type or organization. Services can be created by different kinds of people as follows:

- An end user can make or modify a service by defining a CPL script and uploading it to a server. This type of user is typically a nonprogrammer.
- A third-party service provider can create or customize services for its clients.
- The administrator of an IP telephony server can create services or describe policy for the servers they control.
- Middleware software could create and modify the services, and CPL would be the back-end for their execution. Here, a CPL service could then be a component for other services.

5.1.2 Why Create a DSM Solution for CPL?

The underlying objective for creating a DSM solution was to allow services to quickly and reliably specified and generated to be executed in a CPL server. The DSM language was not targeting end users as their service needs were considered to be solved better by choosing predefined services. The target users of the language were thus service providers and administrators. The actual service specification for the server is specified in XML. The definition of the language for defining services was available as an XML schema (Lennox et al., 2004)—a common starting point for defining modeling languages. It is not a bad starting point as many domain concepts are already identified. However, and as Lennox et al. also acknowledge, XML schemas alone are not adequate to specify correctness of specifications made with the language.

5.2 DEVELOPMENT PROCESS

Development of the language started from the interest of one company to find an easy way to specify CPL services. Writing the services in handwritten XML was considered difficult, error-prone, and as easily leading to internally inconsistent specifications. Having generators produce the specifications in XML was anticipated to give significant productivity and quality improvements.

The starting point for modeling language development was the idea of using graphical models, already suggested in the CPL specification. The language notation was briefly outlined there with the example model shown in Fig. 5.1. Although only some modeling concepts were presented in this sketch example, the idea of using flow models as a computation model was a clear starting point. The rest of the CPL specification, however, focused on domain concepts and their semantics, not on modeling language concepts.

DSM was soon discovered to be of interest for two different kinds of companies: an operator and a manufacturer of telecom equipment. The actual language specification was largely the same because the starting point, the domain of call processing, was the same. The differences were in the process of use and in the extension and integration with other domains. The operator wanted to integrate CPL specifications with other designs, such as specifying voice messages during the call using VoiceXML. The equipment manufacturer wanted to generate Java code based on the SIP framework used in its product platform. This code would configure hardware along with CPL. While when discussing the DSM solution creation in detail, we focus on the common and publicly available CPL specification. Possibilities for extension and integration with other domains are, however, also described along with the case.

The first working draft of the DSM language and generator was made in 2 days. The draft covered about 80% of the language concepts. Implementing the generator took another 2 days and most of time that was spent on handling special cases of generating default path code. More about this is discussed later. The first draft was made by a

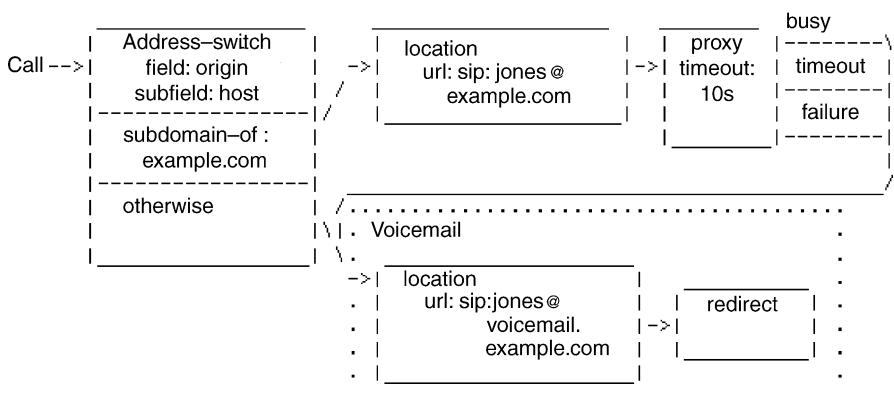


FIGURE 5.1 Idea for the modeling language notation (Lennox et al., 2004)

consultant using the public specification and the rest of the language was completed by the operator. The quick development time is largely explained by having a formal and adequately detailed specification of the domain: the XML schema. The specification also provided examples of CPL service specifications that served as test cases for the DSM solution. About 20 different kinds of services were made and compared against these examples. This testing was largely done by the external consultant using normal XML validation tools, as the CPL servers were not yet available during language creation. On the basis of the feedback, some minor changes to the language were made and model checking was implemented to support the specification process.

5.3 LANGUAGE FOR MODELING CALL PROCESSING SERVICES

DSM for CPL was implemented in small increments using existing IP telephony service specifications as test cases. These test cases were specified by the customer and were partly available in the CPL specification itself (Lennox et al., 2004). In fact, the implementation of the language was done in much the same order that the CPL language was described in this original CPL specification documentation. After adding the first few concepts to the modeling language, the generator was extended to cover the same concepts. This allowed immediately making an equivalent service specification with the DSM language and producing the CPL script for comparison with the relevant test cases. This process was followed until all the modeling concepts, and thus the full CPL specification, were handled.

The structure of the language was also taken directly from the CPL framework. The XML schema provided the language concepts and many of the constraints. A call processing action is structured as a tree that describes the operations and decisions a telephony signaling server performs when a call is made. Each node in the tree has parameters that specify the precise characteristics of the node. The nodes usually also have outputs, which depend on the result of the decision or action. The starting point for the language could thus be based on already established domain-specific concepts. The names used by the XML schema were followed exactly as this made the task of the generators easier. The naming needed for the generated output could be used straightforwardly in the modeling language. Thus the generator could be simpler and there would be no need to translate the names of the modeling concepts to those needed in XML. Later, during generator implementation, this was discovered to not be fully true as legal naming in XML required some string manipulation (see Section 11); for example, spaces are not allowed in the as names and special characters given in the model can be treated as XML tags.

Often, the specification, service examples, and XML schema used different names for the same domain concepts, such as “signaling actions” and “signaling operations” that both mean the same thing. This is a typical situation when the language is not formally defined and there is no reference implementation. Thus, creators of a DSM solution need to make choices and have a major influence on naming policies and establishing a vocabulary.

5.3.1 Modeling Concepts

The modeling language was constructed based on a flow model, with a special root element to start the script. After the root element, other modeling concepts were call processing concepts that usually pointed to the next stage in the call path. This service flow forms a tree structure and ends when the last element in the chosen path is reached. Constraints were given to prevent cyclic structures as discussed later in Section 5.3.2. The modeling concepts were divided into four categories based on the classification of their nature already found in the CPL framework: signaling operations, switches, locations, and nonsignaling actions.

Signaling Actions Signaling actions cause events that a CPL server can perform. A server can proxy a call setup, respond with redirecting information, or reject a call setup. Depending on the signaling action, different properties must be defined. To guide this, CPL specifies its own attributes for each action type. These are added to the modeling language as their own constructs. The concept definition needed for the modeling language construct could be taken directly from the XML Document Type Definition (DTD). Consider the following case of Proxy definition in the DTD:

```
<!-- The default value of timeout is "20" if the <noanswer>
output exists. -->
<!ATTLIST proxy
    timeout      CDATA      #IMPLIED
    recurse     (yes|no)   "yes"
    ordering    (parallel|sequential|first-only) "parallel"
>
```

This piece of DTD specifies that a proxy concept has three properties. The same concepts could be added to the modeling language concept too:

- The time-out interval specifies how long to wait before giving up the call attempt, that is, a number of seconds. There is no default value to be used in the language for this property, but in case a call is not answered, a default value of 20 second is used. This default value is already set in the CPL server.
- Recurse specifies if the server should automatically make further call attempts to telephony addresses when responses are returned from the server that initiated the call. In the language, this was defined as a list of two values (yes, no) with a default value of the parameter (yes) chosen.
- Ordering specifies in which order the user locations should be tried. In CPL, a location set is maintained and there are three ordering possibilities. These were implemented to the modeling language similarly to recurse: parallel, sequential, and first-only are available as a list from which one must be chosen. The default value, parallel, is already chosen as a prefilled value when a proxy object is added in a model. These choices minimize the modeling work and reduce typing errors.

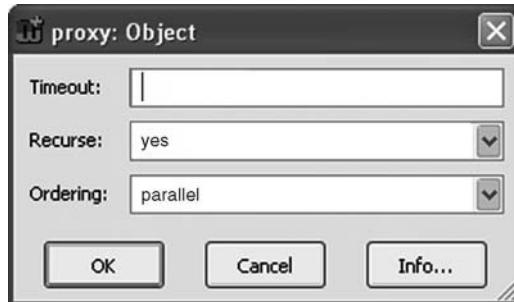


FIGURE 5.2 Specifying a Proxy in CPL using the Proxy language concept

What does the above look like in the modeling language? Figure 5.2 illustrates the use of the Proxy modeling construct when specifying a proxy instance. Having now defined the Proxy concept in the metamodel, all CPL models for specifying proxies follow the DTD. By entering the values based on the language element, a proxy causes the triggering call to be forwarded to the currently specified set of locations. We discuss these connections and constraints with other concepts later.

Other signaling actions, Redirect and Reject, were specified similarly. Redirect has just one property to choose: should the redirection be considered permanent or temporary? Reject states that the call attempt is rejected by the server. The Reject language construct was defined to have two properties: a mandatory status to inform why rejection was made and an optional Reason to describe why the call was rejected. Both Redirect and Reject immediately terminate the call processing execution, so these concepts were defined to have just incoming flows, that is, they form the leaves of the tree.

During language definition, the proxy concept was found to be a suitable place in the language to integrate with another domain, VoiceXML. VoiceXML is used to specify voice messages during call processing. For the operator, VoiceXML was added to the language as its own modeling concept and called by the Proxy via a relationship. This allowed a Proxy to connect the call based on its conditions and, for example, play instructions defined with VoiceXML. The actual voice was already defined elsewhere, and thus the VoiceXML concept just referred to an existing voice specification. Another option considered was integrating metamodels: putting the languages of CPL and VoiceXML together. This was abandoned by the operator as it was seen that different people would create these specifications and there were no real integration needs other than calling VoiceXML.

Switches for Specifying Choices Choices are expressed in CPL with a switch concept. The choice arguments are taken from call requests or from items independent of the call. These are usually entered as property values for the switch. If an entered value is matched during CPL service execution, the choice meeting the condition is chosen. In the case of switches, the condition is simply True or False. The choice results are specified by connecting a switch to another modeling concept using a

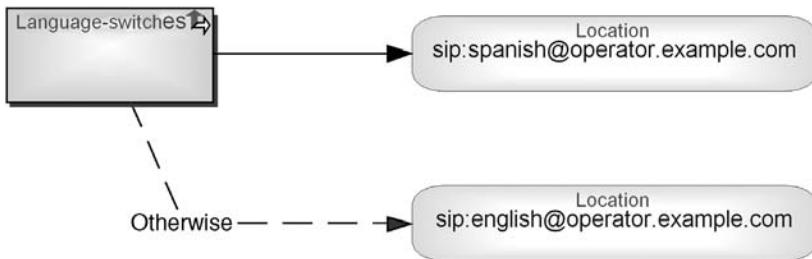


FIGURE 5.3 A specification describing a choice expressed with a switch

relationship. These relationships are divided into two different kinds: a Default relationship, which specifies the result when the condition is met, and Otherwise which specifies the result when the condition is not met. The name of Otherwise was taken directly from the CPL terminology. Figure 5.3 illustrates this: the Default choice is made when the caller wants to communicate in Spanish; otherwise English is chosen.

CPL recognizes different kinds of conditions. These were represented in the languages using different switch types, that is, with dedicated object types as modeling concepts for each switch type. Each switch type thus had different kinds of properties to be entered. Their definition was taken directly from the DTD as with the signaling actions discussed earlier:

- An address switch specifies decisions based on one of the addresses present in the original call request. For this purpose the modeling concept needs two property types: Field and Subfield. The value for Field is mandatory and is implemented as a list of fixed values, where the most common value is already given as a default value. The Subfield is used to specify which part of the address is considered. To support the modelers, a predefined list of possible address parts is provided in the language. Because the Subfield property is optional, no default value was specified. If no subfield is specified, then the entire address was used when evaluating the switch. The results of the address switch were represented along with the Address switch concept: a string value to be matched and a matching policy. An alternative approach would be to show these values by placing them into a relationship pointing to the next element. One reason favoring this choice would be to support reuse of switches as then condition values that depend on the case would not be reused, just the switch condition. This was abandoned because other switches would behave differently during modeling. For example, the Language switch, discussed next, has just one value for comparison and would then be empty.
- A Language switch specifies the communication language the caller wants to use. In the modeling language, this was implemented by adding a property type for entering the preferred language. This property was defined as a string, although predefined values could be added to speed up modeling. If the value entered is matched during call processing, execution of the specified language

choice is made. If a match is made (True), a choice specified using a Default relationship path is followed. See Fig. 5.3 for an illustration.

- The string switch is needed to specify decisions based on free-form strings present in a call request. It has one property to give the string value and two additional fields to specify in more detail how the string matching is done and what protocol is used (e.g., who initiates the call). To support capturing these, two property types were added to the String switch modeling concept.
- The time switch is needed to define time- and date-dependent execution information. For example, during holidays calls may be directed differently. In the modeling language, different time and date values were specified as property values of the Time switch. As it was possible to define 19 different kinds of properties, out of which actually very few are used at a time, the timing properties were divided into two categories. Two mandatory values had their own property types and the rest of the timing values were defined using an optional list. The list was defined as pairs of a timing property and its value. The property types were named as in the XML schema for CPL to keep the generator simple.
- The Priority switch allows a CPL script to make decisions based on the priority specified for the original call. This priority switch concept was defined so that the modeler could choose among predefined priorities and conditions. An example of a priority is “less than an emergency.” The value “normal” was defined as the default priority as in CPL, to be used when no priority is given. In this respect, the priority could also have been left empty, or the generator could replace empty values with the default value during the generation. Showing the default value during modeling was considered to make the specification process more transparent and service specifications easier to understand.

Although each switch was described with its own language concept, an alternative possibility considered was to have only one switch concept that would refer to different switch types. Then a modeler would first explicitly choose to add a switch into a model and after that choose which kind of switch is needed. The benefit of this approach would be that the existing switch instances could be changed to new types without deleting the main switch element. This approach is useful when the rules and relationships are the same among the (switch) types. As a single CPL design was considered relatively small, less than 21 main concepts, it was considered more straightforward to use distinct types. This also meant that the switches could be used similarly to other concepts that had been implemented as their own types.

Location Modifiers for Accessing Location Data The behavior of many signaling actions is dependent on the current location set specified. For example, if the user location is Spain, a call can be directed differently than if the location is something else (see Fig. 5.3). The set of locations to which a call can be directed is kept in a CPL server. To modify a location, the modeling language was extended with constructs called location modifiers. They allow adding and removing locations from

the location set. Following the XML schema, three different modeling objects were added to the language:

- Explicit Location are set by adding the given URL address to the location set. The location can be further refined with its priority in the location set and with an optional property to choose if the existing location set is cleared before adding a new location. Thus the property specified here is used as an argument to access the CPL server's service.
- Locations can also be set from external sources. For this purpose, CPL has a location lookup concept. This was represented as its own concept in the language. Location lookup has three properties: Mandatory Source to give a location from which the CPL server can ask for the location, Time-out to specify the time to keep trying before abandoning the lookup, and whether the current location set needs to be cleared before adding the new location.
- A call processing service can also remove locations from the location set. For this purpose, the Location removal concept has a property to specify the location as a string value. If the value is left empty, the default in the concept, all locations are removed from the location set.

The locations were also specified as their own concepts because their connections and rules are different. Subtyping a single location concept, for example, with a property type having different location types as values would not make sense. In this respect, language definition followed the choices already made with switches. The main exception between these choices was that the metamodel did not allow such rich possibilities for choices. For example, Location lookup has the possibility to define three alternative choices, whereas switches allow only two (default and otherwise). Another difference is that lookup, can not be followed by another lookup, whereas switches can be next to one another in the call processing path.

Nonsignaling Actions To record actions that were taken, a server can send mail to the user or log information about the call. In the language, both were defined as their own modeling objects since they had different properties: The Mail concept in the language was defined to include a “mailto” URL property, whereas the Log concept contained information about the log name and its more detailed description.

During language creation, the concepts acting as modeling objects were formalized into a metamodel. The metamodel in Fig. 5.4 illustrates the definition of the object types of the language in the metamodeling language described in Appendix A. All the main domain concepts were defined as subclasses of an abstract Node element to follow the structure of the CPL specification. The object types, however, were defined incrementally, and thus the metamodel shows only the final result.

Model of Computation In addition to the domain concepts presented above, the flow model needed additional modeling concepts. A root node was added to the language to specify a starting point for the call processing service. This modeling concept did not have any properties but just marked the initiation of the call session—making or receiving a call. Although it would be possible to identify the start of the

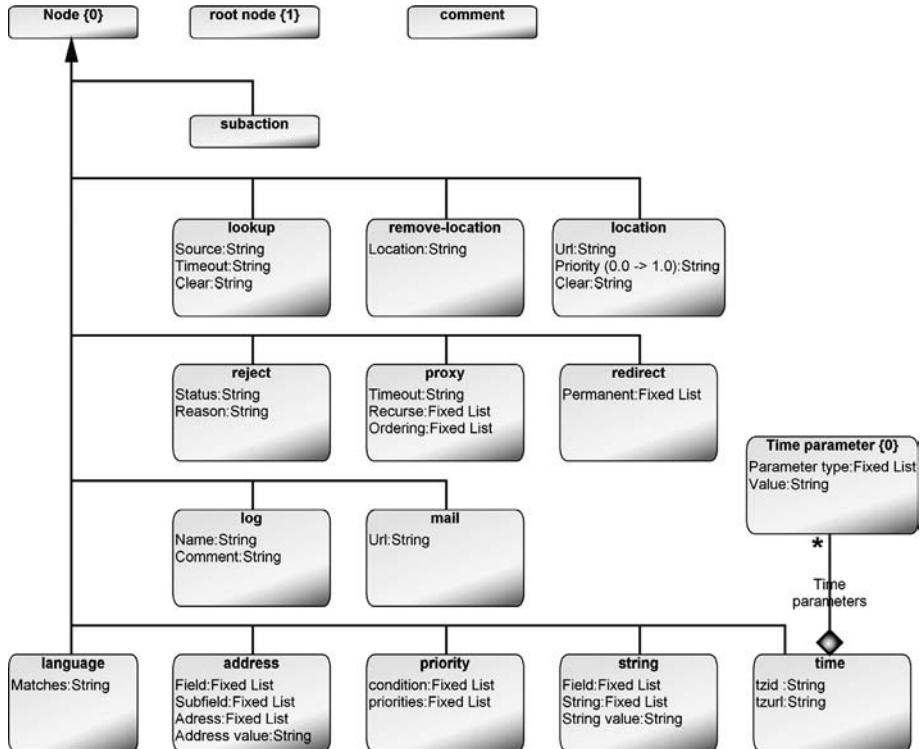


FIGURE 5.4 Metamodel specifying the object types of the language

service from a concept in the model that had no incoming relationships, this policy could easily lead to conflicts if a model is not complete. It was also considered easier to understand the specifications when there was a single starting point.

The start of a call processing service with the Root node was implemented in the language with a Session Phase relationship from the Root node. This relationship was further classified into incoming or outgoing calls. This information is important in CPL for performing any necessary initialization, such as emptying the location set in the server. The modeling language did not include any explicit service end concept. The execution of the service ends when the last element in the flow is reached and performed. At this stage, a CPL server takes the resulting action and the service ends.

The flow of a call was divided into two parts. The main service flow, described using a directed relationship, was called Default path. The name was taken directly from the CPL schema. To model alternative binary choices with the switch concept, an Otherwise relationship was added to the language. Later, dedicated relationships for both Proxy and Lookup were added to extend the default path to specify the predefined kinds of outputs. Default or Otherwise relationships could not be used as they did not have properties to specify outputs. Adding them to the current relationships was abandoned as that would make the language more complex to use: a modeler would be asked about design data that could be unnecessary. To make modeling easier and hide complexity, these choices were put directly into the metamodel.

Model Layering with Subactions CPL identifies two layers to support reuse and modularity. The main action is treated as a topmost action that is triggered, via the Root element, when events arrive at the server. As many services have common functionality, these could be treated as reusable services to be called by other actions. These subactions were defined in the language by enabling an action to decompose into another model treated as a submodel. As the subaction could be defined using the same language concepts as the main CPL definition, there was no need to have a separate language for the subactions. In the metamodel, this was achieved by defining decomposition from a Subaction concept to the Call Processing graph. This language structure also supports reuse well as any Call Processing specification can be used as a subaction elsewhere. Figure 5.5 illustrates possible subaction calls: A design model becomes a subaction when it is called by a subaction in another model. This choice influenced the generator very little: the root element specifying the initiation of the call processing service was just ignored during generation. The model was the same independent of its use as a subaction or main action.

The choice of whether the subaction should be moved to the main specification model or kept in its own separate model depends on the modeling process: if changes in a subaction need to be propagated to all specifications using it, the subaction should be kept in its own model. This capability to support reuse was thought to be a useful solution for service providers and server administrators. The end user was thought to prefer a simpler approach as chances for reuse would not be so common. Also, the end user would be less experienced and having to understand reuse and model hierarchies might require learning things that are rarely needed. To outline the content of the subaction, the language and related tool support were implemented so that the main

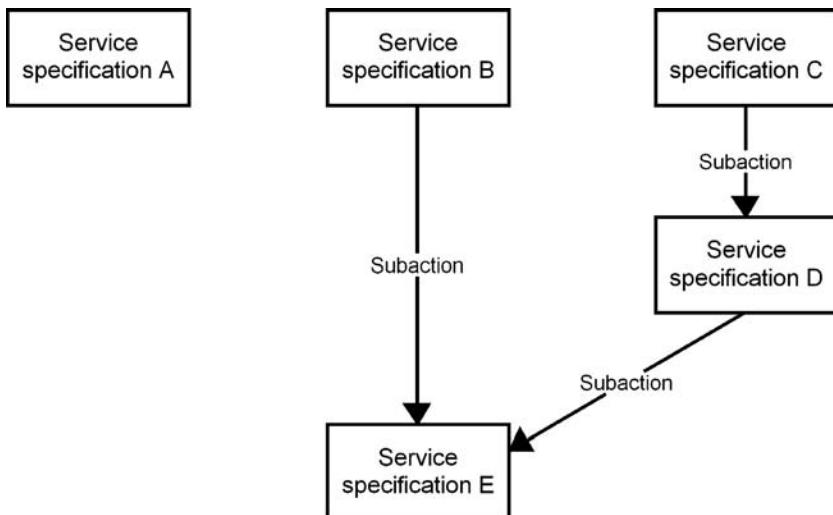


FIGURE 5.5 Reusing a model as a subaction

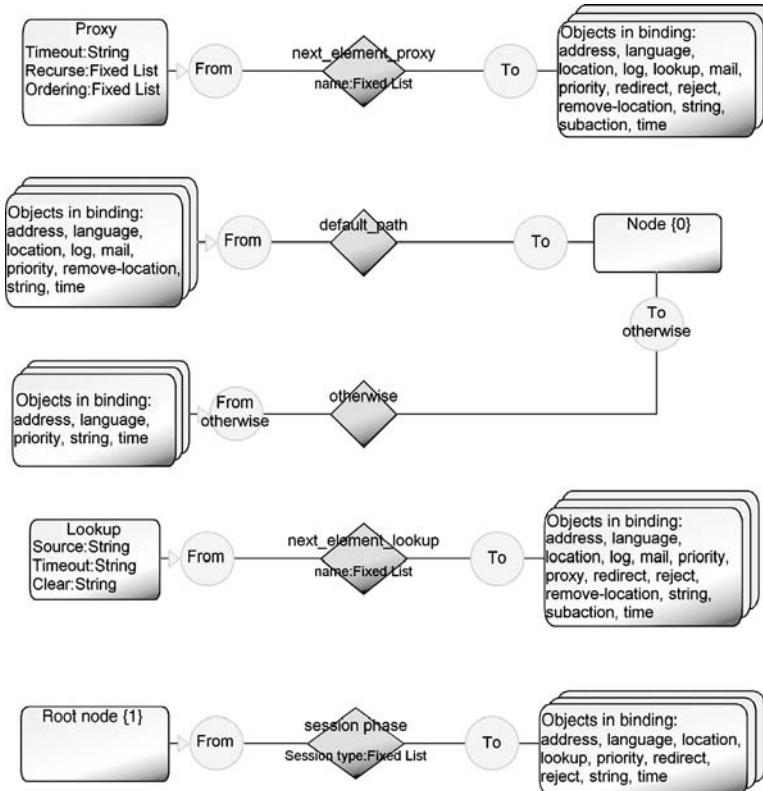


FIGURE 5.6 Metamodel of the language

contents of the subaction were summarized in the main model within the subaction object that referred to a subaction model.

5.3.2 Modeling Rules

Along with identification of the modeling concepts, many of the constraints and model correctness rules were identified. Whenever possible, they were defined to be a part of the metamodel to be checked actively in the course of time. Otherwise they were supported in related model-checking reports. Figure 5.6 gives an overview of the call processing language as implemented in the MetaEdit+ tool. The metamodel shows the modeling objects and their bindings: possible connections via relationship types and role types.

Tree Structure By definition, CPL scripts should never include loops or recursions. This, however, does not mean that models should always form the structure of the script. Model elements should be made reusable, and links to already existing services could be provided to support efficient modeling work. Most constraints ensuring a tree structure were enforced in the modeling language. During

code generation, remaining illegal structures for XML (but legal for the model) were identified and forced to follow a tree structure. To support the call process view, the modeling language had the following rules defined:

- A root element could occur only once in a model, that is, only one start element for each service specification was allowed. In the metamodel, the root element further has a constraint that a root can only be in one relationship to start the call process. Reusable subactions could have their own root element, which would be omitted during generation.
- As both Redirect and Reject immediately terminate the call processing execution, these concepts were defined to have just incoming flows; that is, they form leaves of the tree.
- Cyclic structures between the objects were reported as errors.

Choices and Conditions in the Call Path The metamodel was also extended with concepts and rules to enable choices. This was done by having special relationship types for connecting choice types, and related rules called bindings, that specify the legal elements as source and target. The rules included the following:

- The call processing path was initiated with a dedicated relationship (session phase) from the root element, specifying if the call was incoming or outgoing.
- For most choices two alternative path relationships were added, called default and otherwise. The otherwise path was defined to be legal only for selected call elements, namely, address, language, priority, string, and time. See Fig. 5.6 for a detailed metamodel.
- As lookup had three alternative types, their value was added to the dedicated relationship type (called next_element lookup). This relationship could only be initiated by the lookup element.
- Similarly for the proxy: a relationship with a property type for proxy output was added. This relationship could only be started from a proxy element and had five different output values. Use of the same values multiple times for the same proxy instance gave a warning. Having just a warning was preferred, as it would allow changing output values without first deleting existing values.

Rules for Model Hierarchy To support reuse of other call processing specifications, the language provided a subaction concept. Each subaction object could include only one subaction specification. During code generation, subactions were produced first, followed by the main call specification.

Model Checking Reports In addition to constraints and rules attached to the metamodel, separate model checking reports were also made. These reports analyzed models for the kinds of rules that cannot or should not be checked after each modeling action. For instance, an unreachable part of the call model and a missing default path are best warned about with reports. Similarly, if choices based on a location lookup

include the same lookup value multiple times (e.g., more than one success choice), their occurrence can be reported.

Model checking results were accessed in two different ways: separately from a tool menu and by placing the current status and warnings next to a separate model element. A third option would show the warning next to each model element having an error, but combining them in one place was preferred.

5.3.3 Modeling Notation

Lennox et al. (2004) proposes a very basic notation for the language: directed arrows between boxes (see Fig. 5.1). This is a typical starting point when defining a concrete syntax for a language. The notation was made by a consultant. During the notation definition, no customer feedback was given, perhaps because there was no established practice in either of the companies—anything was considered better than writing specifications directly in XML. First, the notation was just boxes but later symbols that clearly distinguished the concepts were used.

The key principle in choosing the notation was that it would be possible to read the whole specification from the visually represented model. This was a realistic objective because each concept had relatively little data. The notation applied a visualization pattern based on the type of concept: similar kinds of language concepts were represented with the same shape. For example, all switches were represented as rectangles and locations as rounded rectangles. To further improve the readability, small icons were used in symbols to illustrate the concept: all switches have an icon representing choice; for example, the time switch has a small watch icon. Similarly, the relationships used to mark the Default path and Otherwise path were also graphically distinguished: the usual default path, including root start and signaling actions, was represented with a solid line and the Otherwise path was represented with a dotted line. The relationships also showed the path values, such as choices made for proxies and location lookups. Root was illustrated using a circle similar to the start state in many state transition diagrams.

To make the notation aesthetically pleasing, the language was finalized by adding different colors; fill colors for the symbols and icons made the notation easier to read. For example, all signaling actions were represented in light brown. As colors do not work well on monochrome printers, their value in the notation was limited to computer use: inspecting models in the modeling tool or generated documentation.

5.4 MODELING IP TELEPHONY SERVICES

5.4.1 Example Model

The structure of the CPL language maps closely to its behavior so that a service developer who understands the domain concepts can also understand and validate the service from the graphical models. Figure 5.7 illustrates the specification of a

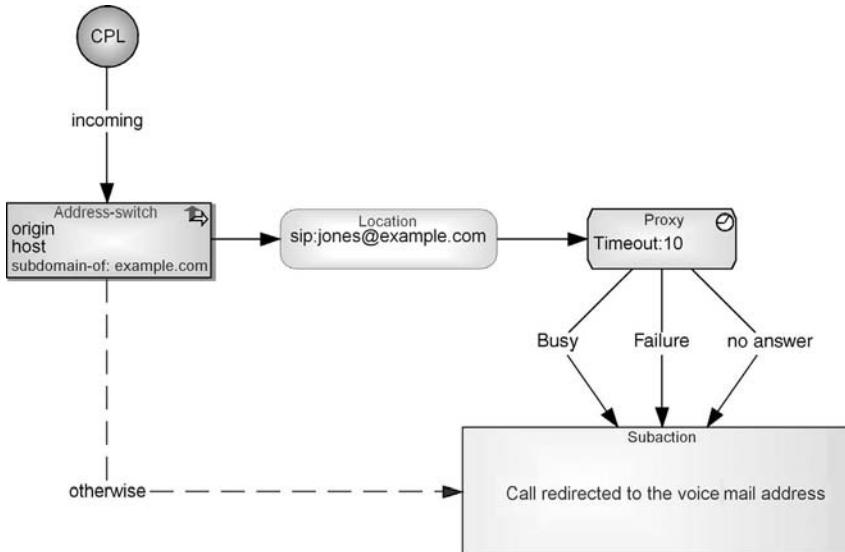


FIGURE 5.7 A sample of call redirecting service expressed in CPL

service: here, all incoming calls originating from “example.com” are redirected to the address “sip: jones@example.com,” and if there is no answer, a line is busy, or a failure occurs, the call is redirected to the voice mail address “sip: jones@voicemail.example.com.” All calls that originate from addresses other than example.com will be redirected immediately to the same voice mail.

5.4.2 Use Scenarios

The DSM solution was made to support the reuse of services as a subaction (see Fig. 5.5, model layering). In Fig. 5.8, the same subaction is reused as in Fig. 5.7. In this case, the user attempts to have his calls reach his desk; if he does not answer within 8 second, calls from his boss are forwarded to his mobile phone, and all other calls are directed to voicemail. If the call setup fails, the failed call information is stored in the log and sent to the email address. This kind of structure was seen to allow providing a library of basic services or their parts to speed up service creation based on existing subactions.

5.5 GENERATOR FOR XML

Defining generators to produce XML is often quite a straightforward process: elements in a model and their connections are described by XML tags. This becomes even easier if the modeling language maps well onto the XML schema. This was true in this case, too, since both are designed to be a good way of describing the same domain. Where XML schemas have had to sacrifice understandability to cope with

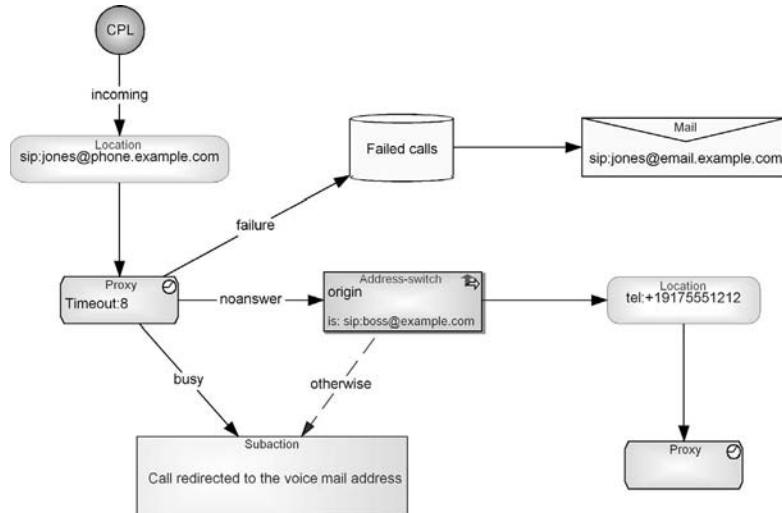


FIGURE 5.8 Call processing service reusing the same voicemail subaction

the limitations of XML, the modeling language can do things in a more natural way. In this case, the generator will do a little extra legwork to produce the verbosity and duplication required in XML.

The first version of the generator was made as a proof of concept for the customer. Later, the generator was completed based on customer feedback. More effort was needed for defining the default behavior for different types and defining the XML tags for the ending elements correctly. The generator was defined in parallel with the modeling language: when a new modeling concept was added to the language, a generator module related to the added concept was implemented too. This approach allowed testing the language using the example specifications given in the original language specification. Further testing of the generated result was done using basic XML schema validation tools, as there were no actual CPL engines available.

From the designs expressed in the models, the generator produces code that can be parsed and then executed in the CPL server. Let's look next at the structure of the code generator and then samples of the generated call processing scripts.

5.5.1 Generator Structure

The generator visits each element in the model, calling a generator module for that element's type (see the visitor pattern in Chapter 11). Figure 5.9 illustrates this generator structure by describing the generator modules and call structures between them. The generator modules that handle the CPL concepts are emphasized with a gray rectangle. Most of the other generator modules provide generator functionality that was common and applied at many places during the generation.

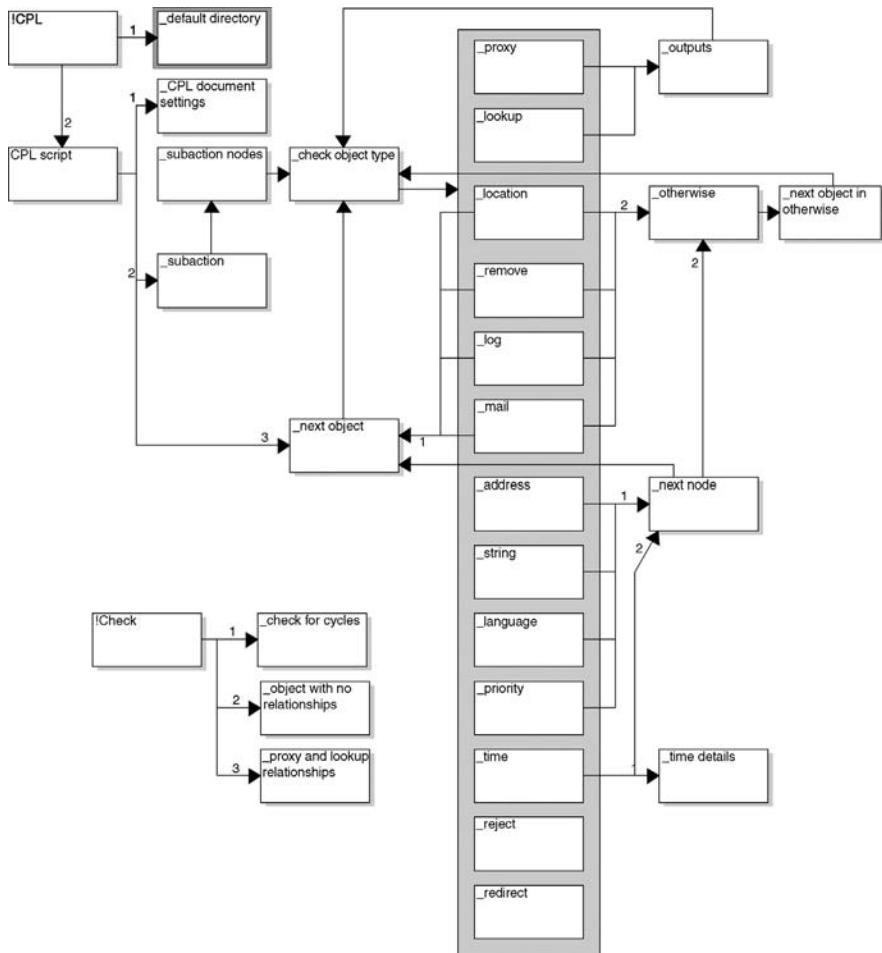


FIGURE 5.9 Structure of the CPL generator

Generators were defined for three different purposes: producing the service specification into a file, producing it into an output window, and checking the service specifications. In the figure, these main generators are represented on the left-hand side. The difference between “!CPL” and “CPL script” is minimal as the only difference is how the generated result is shown: saved into a file in a default directory or shown in the generator output editor. Therefore, these two generators use the same generator modules.

After having defined the document settings (XML header), the generator looks for subactions to be generated. This was needed because CPL expects all subactions to be defined in the beginning of the service specification. This is followed by choosing the domain concepts to be generated: the “_check object type” module chooses the domain concept found in the call processing specification and runs the right domain-related generator module (inside the gray box in Fig. 5.9).

So each concept had its own generator module. As some concepts resembled each other (e.g., all switches), a single generator module could serve more than one concept. This was mainly done because the generated outcome looked the same for all modules. Generator modules named after domain concepts then called other generator modules—mostly those applied to navigate further in the specification, such as the generators “_next object” and “_next node.” Embodying common behavior, they were thus defined to be reusable.

The naming of modeling concepts was taken directly from XML to make generators easy to make: a generator uses domain concept names while accessing the model and producing the required output. As XML required a special naming policy, domain concept names using a capital letter were changed to lower case. Generating XML is problematic for reserved words and white spaces in entries. For the former, regular expressions were used to describe legal text strings and for the latter a generator was defined to make entry values legal in XML (e.g., remove spaces from domain concept names).

To produce the conditions in the same order, the generator first produces the default path information followed by the “otherwise” path. The generator implements this by using the metamodel data—access first the default path and then the otherwise path. For CPL this was an adequate and simple solution. If different path execution orders had needed to be specified, then the path choice could have been given to the relationship as a property describing the path. Giving a number to the relationship path is one approach. If there are only two paths from the switch, or if it is relevant to describe the execution order only for one of the many paths, such as default, first, and so on, marking just one path is enough. The path order could also be taken from the order of specification—the path specified first in the model is executed first—or from spatial information—the path specified highest is taken first. These choices, however, could restrict the modeling process, and reuse of larger model chunks between different specifications would not necessarily work—the context would be different between models in a library and their use in design.

As not all design rules can be defined in the modeling language to be checked at modeling time, model checking generators were defined. Rather than calling individual model checks, they were all called by one checking generator named “!Check.” This generator checked a service specification and created warning reports about cyclic specifications. The result of the warning report was defined to be shown in a separate window, although another possibility would have been to place the warning information as a visible part of the designs. One approach considered was showing in the model that there were inconsistencies, but not showing the details; the details could then be seen by running the complete model checking report.

5.5.2 Generator in Action

The task of the generator is to follow the nodes via their connections and create the corresponding CPL document structure in XML. The listing below shows an example of the generator output. The XML is produced with the generator from the service specification illustrated in Fig. 5.7.

Listing 5.1 Call redirecting based on location of caller origin.

```

01  <?xml version="1.0" encoding="UTF-8"?>
02  <!DOCTYPE call SYSTEM "cpl.dtd" -->
03  <!DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL
04  1.0//EN" "cpl.dtd" -->
05  <cpl>
06  <subaction id="voicemail">
07    <location url="sip:jones@voicemail.example.com">
08      <redirect />
09    </location>
10  </subaction>
11  <incoming>
12    <address-switch field="origin" subfield="host">
13      <address subdomain-of="example.com">
14        <location url="sip:jones@example.com">
15          <proxy timeout="10">
16            <busy><sub ref="voicemail" /></busy>
17            <noanswer><sub ref="voicemail" /></noanswer>
18            <failure><sub ref="voicemail" /></failure>
19          </proxy>
20        </location>
21      </address>
22    <otherwise>
23      <sub ref="voicemail" />
24    </otherwise>
25  </incoming>
26 </cpl>
```

The generator starts by going through all the subactions of the service specification. This example contains only one subaction, the `voicemail` box at the bottom right corner of the model, for which the generator produces lines 4–8.

```

04  <subaction id="voicemail">
05    <location url="sip:jones@voicemail.example.com">
06      <redirect />
07    </location>
08  </subaction>
```

This “voice mail” subaction defines a location element (line 5) as well as a redirect element (line 6), which activates that new redirection automatically.

After producing subactions, the generator starts to specify the main call processing actions. It goes through the service specification from a service start (the “CPL” circle in Fig. 5.7). The generator crawls the connections from the CPL circle through the “Incoming” relationship to the Address-switch object. It produces the properties of the Address-switch node as element attributes in the generated output (lines 10–11).

```

10    <address-switch field="origin" subfield="host">
11      <address subdomain-of="example.com">
```

The generator continues to follow the main flow path arrow to the next object and produces the location definition (line 12).

```
12           <location url="sip:jones@example.com">
```

The path continues and the proxy handling is generated on lines 13–17: first the timeout attribute (line 13), followed by three alternate connections from the proxy element.

```
13           <proxy timeout="10">
14             <busy><sub ref="voicemail" /></busy>
15             <noanswer><sub ref="voicemail" /></noanswer>
16             <failure><sub ref="voicemail" /></failure>
17           </proxy>
```

Finally, the generator produces lines 20–22 for the cases where the call origin has an address other than example.com.

```
20           <otherwise>
21             <sub ref="voicemail" />
22           </otherwise>
```

The generated code forms a complete service whose validity has already been checked at the design stage. Because the modeling language contains the rules of the domain, service creators can only make valid and well-formed design models. The modeling language can also help service creators with consistency checks and guidelines, for example, by informing them about missing information (such as no call redirect being specified). These rules are highly domain-specific and thus can be handled only with a domain-specific language.

The generator producing the call processing scripts, and related model checking, was tested against several test cases. First, script samples were taken from the CPL specification, but further cases were made to cover situations that were not available in the specification documentation. In total, about 40 different kinds of service specifications were used and found to be enough to test the possible combinations. Then the generator was executed and the result was compared first for legality as XML and then against the test cases.

5.6 FRAMEWORK SUPPORT

The CPL generation did not require any framework code to make model-based code generation possible. As the service specification is made based on the CPL specification, the schema largely defines what the generated code should look like. Similarly, for the VoiceXML extension for the operator, just a simple link to external files was enough. If the DSM were to be extended to also cover VoiceXML, most likely no framework code would be needed there either.

The generated code, the call processing service specification, can thus be immediately executed in the CPL server. This target execution environment provides default behavior for situations where the CPL specification is left open and for cases where a failure may occur. For example, if locations are not found for the call target,

the failure is handled by the server. Here, service execution would be terminated and the default behavior is provided. In this sense, CPL already provides a good target platform for specifying services.

For the equipment manufacturer, the target configuration and generated Java would obviously need some framework code, but these will not be discussed here. We discuss Java code generation in two other chapters with different framework support. We show first in Chapter 6 how static structures are generated into Java code and then in Chapter 9 how framework support can be applied to code generation that also covers the behavioral side.

5.7 MAIN RESULTS

The DSM solution for CPL allows call processing services to be specified graphically. Services can be created with a language that almost solely applies call processing concepts—concepts that every service engineer must know anyway. A code generator reads the created service specifications to produce service specifications in XML.

The DSM approach provided the following benefits:

- Service creation became significantly faster. Making small services with the created language and comparing it to earlier manual practices showed that DSM lead to processes that are 6 times faster. It was further expected that when the specification size and complexity become larger and subactions can be reused in the model levels, the difference would be even bigger. This is natural as XML is not really made for reuse.
- Service creators do not need to master XML. With the created DSM solution, service engineers need not deal with XML concepts at all. XML was just used internally as an intermediate output for passing call processing specifications to the CPL servers.
- Specifications could be checked during modeling. Checking the correctness of the specifications was valued highly, especially because call processing services for IP telephony were new. The simple statement that “if the specification can be drawn, it will execute in a server” was very powerful and easy to understand by personnel. Model checking was mostly done at modeling time, but some checking by generators was also found useful.
- The quality of specifications was better. The first tests showed significant decreases in errors and using generators totally removed most typical errors.

5.8 SUMMARY

This first case of DSM creation shows how a language is created based on a well-bounded domain specified as an XML schema. Schema concepts are mapped almost one-to-one to language concepts. Thus the modeling language concepts are directly

based on the key elements and the structure described in the CPL specification. Language elements such as Switches, Locations, and Signaling actions and their attributes are specific for processing Internet phone calls and services. These concepts automatically become familiar to the service developer and they form a comprehensible working environment for CPL service developers.

The modeling language was built before any CPL servers were implemented, so the platform was not yet ready. However, the CPL DTD and later the company-specific extensions allowed building the DSM solution in advance. Adequate specifications of the language enabled this parallel development of the DSM and of the platform. This case shows that there is no need to wait until the target environment and platform are ready: DSM can be made ready beforehand. This can be useful when service specifications are needed to be made before they can be applied.

On the code generation side this case was very clear-cut. This was mainly because the language concepts are already defined as elements in XML, and the property values of the modeling constructs are attributes of the XML elements. CPL servers already provided an interface specification and thus showed the expected input.

CHAPTER 6

INSURANCE PRODUCTS

In this chapter we describe the specification of financial and insurance products for a web portal. Unlike the other cases presented in this book, we focus here on modeling only static structures: The platform already provided the implementation for the behavior. The allocation of the specification work is interesting in this case. The modeling language is used by insurance experts, and thus by non-programmers, to specify various insurance products. The same people then use a generator to produce the required implementation as Java code for a J2EE web portal.

6.1 INTRODUCTION AND OBJECTIVES

The starting point for this case was a business decision: A company acting as an information provider and broker was building a portal for handling various financial products and insurance products. The portal aimed to compare information about insurance products from different providers and share the data with insurance companies and other financial service providers. The nature of the service was to target business users rather than consumers. To codify the product information (like health insurance), various insurance details such as insurance coverage, indemnities, payments, covered risks, damages, bonuses, and tariffs needed to be formally specified for each product and made available for analysis and comparison. The platform then provided the common parts, such as those dealing with money and

payments. The specification of insurance products was not simple since insurance products have a lot of detail and differences among similar kinds of products from different insurance companies were of great interest. In addition, each product is usually related to other insurance products from the same provider and thus combined insurance packages needed to be specified too.

The specification work was seen as considerable since the portal would include hundreds of products, and they all would need to be specified formally so they could be compared and analyzed in the future. The domain of insurance products also evolves, requiring modifications to already defined products and also new products that are defined based on market situation, legislation, and so on. So in terms of software engineering, the specifications would need to be maintained over time.

To implement the portal, the first option considered was to ask programmers to write the product specifications in Java. This step would most likely be supported by insurance experts who would first codify relevant parts of the insurance products into requirement documents. These would then be used by programmers to write the implementation. This step was taken first and applied to implement a few product specifications. The second option, used in production, was creating a Domain-Specific Modeling (DSM) solution that would allow domain experts, that is, insurance people, to specify the products completely and then generate the needed implementation. This choice led to significantly faster development, allowed specifications to be checked early, leading to fewer errors, and reduced the amount of resources needed. Next we describe the development process and the DSM solution in more detail.

6.2 DEVELOPMENT PROCESS

The creation of a DSM solution started because the company was seeking an efficient way to capture financial and insurance products and implement its portal-based service. Manual programming was simply seen as too costly, time-consuming, and leading to errors that would later cause problems when insurance information would be analyzed and compared. These typical problems of manual programming had already become very visible when the company had implemented its first specifications of existing insurance products.

For the implementation of the DSM language, generator, and related tooling, the company invited bids from two external companies. The key requirements for the selection were the time needed to implement the DSM solution and its development costs. The company decided to use a commercial metamodeling tool and an external consultant to implement DSM. The actual implementation work took 11 man-days. The project was started in early August and the DSM solution was delivered to the company at the end of September. In October a pilot study was conducted with a few modelers and the DSM solution was introduced to the organization with a 2-day training course.

Before defining the DSM solution began, a target platform had already been selected: a repository product following the Meta Object Facility (MOF) as a storage format and interface with other tools. As mentioned, a few product specifications

were already implemented as Java code. This code was used to test the other components used, such as the libraries in the framework and the repository. During DSM creation, the manually written Java code specifying the few products was used as a reference for testing the modeling language and the code generator. The main input for the language specification came from the “domain model” that was stated as a common semantic model for all insurance products. Based on the domain model, knowledge of all insurance products was explicitly specified and made available for further analysis, comparison, and modification. The domain model was used to unify the way insurance products are specified, and it offered comparability of the individual products. In terms of language creation, the domain model was a metamodel: All products were instantiated from the domain model. Looking at this another way, no insurance product could have data about an item not mentioned in the domain model. The domain model was largely defined before creation of the DSM solution began, but during the process it needed to be updated and complemented because the language specification required details that were lacking in the original domain model.

The domain model was originally formulated as an extension of MOF. It basically consisted of a number of (insurance) domain specific types that were all derived from MofClass or MofAttribute. In addition, it provided a set of rules determining the possible ways in which these domain-specific types could participate in relation to each other (like MofAssociations). These rules ensured that a certain insurance-specific product structure was followed. Tagged values were added to certain model elements to distinguish the role that a model element can play under certain insurance-specific points of view. For example, tagging an attribute with an “A”-tag meant that the attribute is needed for an application, a “T”-tag meant that the attribute is needed for calculating the premium, and so on.

The company wanted to follow standards and therefore the domain-specific language was also expected to be defined directly based on MOF types. This requirement was satisfied by first implementing MOF as a metamodel into a modeling tool and then specializing the defined MOF concepts to make the domain-specific modeling concepts. In the metamodel, the modeling concepts were, therefore, subtypes of MOF types. Rather than implementing MOF completely, just those parts relevant for the language definition were implemented. Figure 6.1 represents the major concepts of MOF defined as a metamodel using the metamodeling language of MetaEdit+, the tool applied (see Appendix A). Each rectangle represents an object type, and connections with a diamond show aggregation relationships, for example, MOFClass may have multiple attributes. Then, these concepts were used in the modeling language directly since many of the modeling concepts (object) types were created by inheriting from MOFClass.

In addition to the MOF elements described above, Association, Generalization, and Dependency relationships were implemented from MOF types. Similarly, an AssociationEnd role type was implemented and used as a supertype for the domain-specific role types different objects can take when connected to other modeling objects. We describe their use in more detail later when presenting the metamodel for the modeling language.

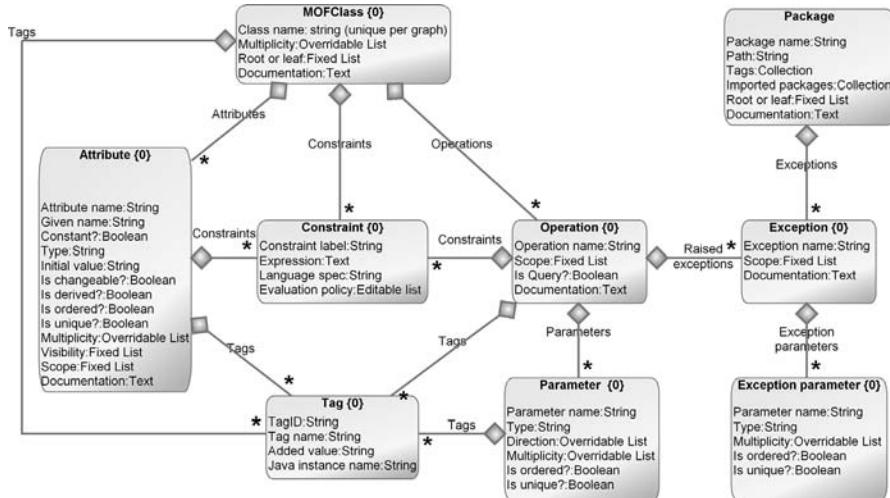


FIGURE 6.1 Metamodel of the relevant parts of MOF

The requirement to apply MOF, however, was later noticed to lead to an unnecessarily complex modeling language. For example, associations and their role names were provided in the language, since they came with MOF, but were not used at all. The requirement to follow the standard for metamodeling also caused additional delay in the DSM creation: In the middle of the DSM implementation, the company realized that the version of MOF had changed, and they needed to recheck the definitions made in the domain model. Following the newer MOF releases was seen as relevant since the target environment used for storing the information about the financial products was expected to change along with new MOF versions. However, for language creation, it was not relevant since the domain-specific language could have been implemented without any relation to MOF. However, to satisfy the company policy for following standards, some terms were changed to the existing implementation of MOF (Fig. 6.1). These changes were mostly cosmetic, and the DSM tool updated the language definitions automatically after the supertypes (i.e., MOF implementation) were changed.

6.3 LANGUAGE FOR MODELING INSURANCES

The starting point for the language definition was an existing domain model: a common semantic model for all insurance products. This domain model was made by the insurance experts of the company, and the task for the developer of the DSM solution was to make the domain model formal and usable as a modeling language. In that respect, the modeling language creation was easy since the domain was already well bounded and had established semantics. Many of the modeling concepts could be derived directly from the domain model, as could some constraints. The missing

constraints were added and refined in collaboration with domain experts. This process was rather simple since most questions were easily answered by domain experts. A contributing factor was that the questions were related to the already known domain model. The questions dealt with missing relationships between objects, cardinality constraints, having binary relationships instead of n-ary relationships, and organizing specifications larger than one diagram.

6.3.1 Modeling Concepts

The analyses of the domain model showed that most differences among the insurance object types were related to the relationships they may have with each other rather than to their individual property types. In fact, almost all the object types had the same property types, those inherited from MOF. Figure 6.2 illustrates part of the original domain model.

The large number of rules related to relationships between domain concepts led to the definition of multiple different object types. Each domain-specific object type then had related roles and constraints. Another alternative considered was having just one object type with a classifying stereotype along with a constraint definition for connecting objects. This approach, however, was impractical since some objects

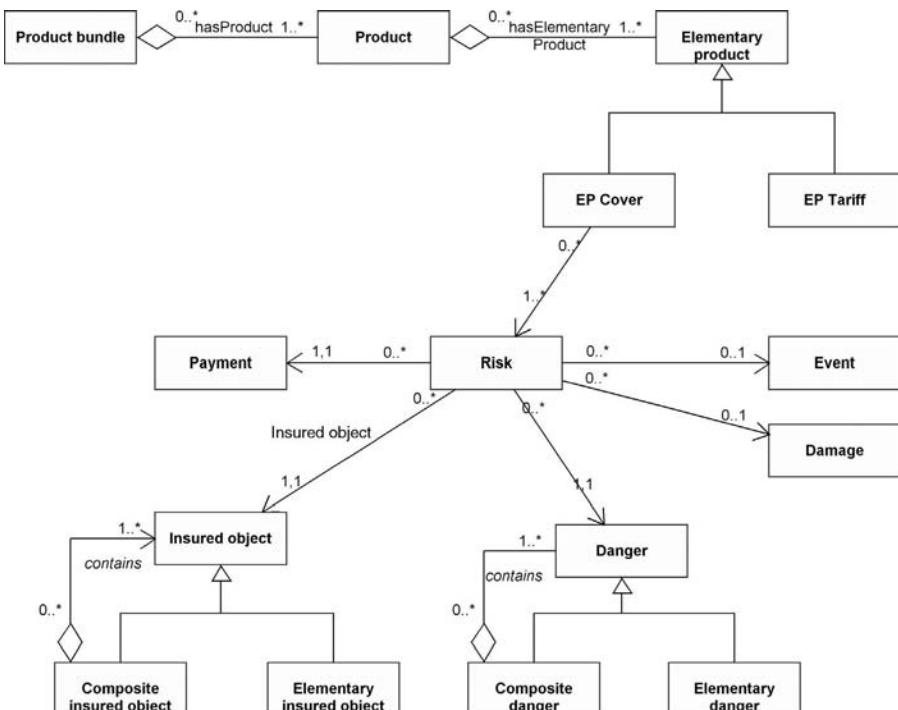


FIGURE 6.2 Part of the domain model

had additional properties, and it would make the constraints too complex to define and maintain. The original domain model, having different object types for each domain concept, also favored creating individual object types in the language.

Language definition started by categorizing the domain concepts into groups and implementing the root concepts first. In the case of insurance products, these root concepts are an insurance product and a bundle of insurance products. Since both of these had common properties, a new supertype, called Domain Class, was added to the metamodel. This supertype had the common properties like Given name and Type. Figure 6.3 shows the main modeling concepts in relation to indemnity insurances. For the modeling of life insurance products, the language needed more modeling concepts. At this point, it is worth mentioning that the complete metamodel was defined as one diagram, and for the sake of its presentation here, we have divided the metamodel into separate figures.

While the Product bundle and Product refer to whole insurance products, the rest of the domain concepts are used to describe their characteristics. When specifying the metamodel, all the candidate language concepts already had an existing definition. For example, an event was defined as a process of the real world, such as achieving a given age, thunderbolt, theft, suicide, natural death, or buyback, and a damage was defined as an insurance-technical classification of cases of damage as effects of events (e.g., destruction of house contents, loss of cash, disablement, damage to vehicles). The names for the modeling concepts were chosen directly from the domain model to make the language easier to learn and use.

Some domain concepts were further characterized with specific property types, such as followings:

- An insured person has a property type Role to specify the related policy outline, such as connected life insurance or add-on widow insurance.
- Calculation basis has a property type Computation purpose, which could have values like a balance and a calculation.
- Surplus may be based on different types, such as a bonus, an immediate risk bonus, and an immediate premium deduction.
- Payments could be further characterized by their type.
- Tariff has two property types: Type for specifying the tariff type used, such as new contracts, compensation, and dynamic, and a property type Variant for specifying if different tariffs are used over the time.

All these properties had some predefined values that could be directly used as a basis for selection. Therefore, the property types were defined as different kinds of lists, such as an overridable list for either selecting one of the predefined values or typing one-off values, or an editable list where the modeler can add values to the list for future selection. The language was defined mostly to use editable lists and it was decided that their use would be analyzed later once the language had been used for defining a larger number of products.

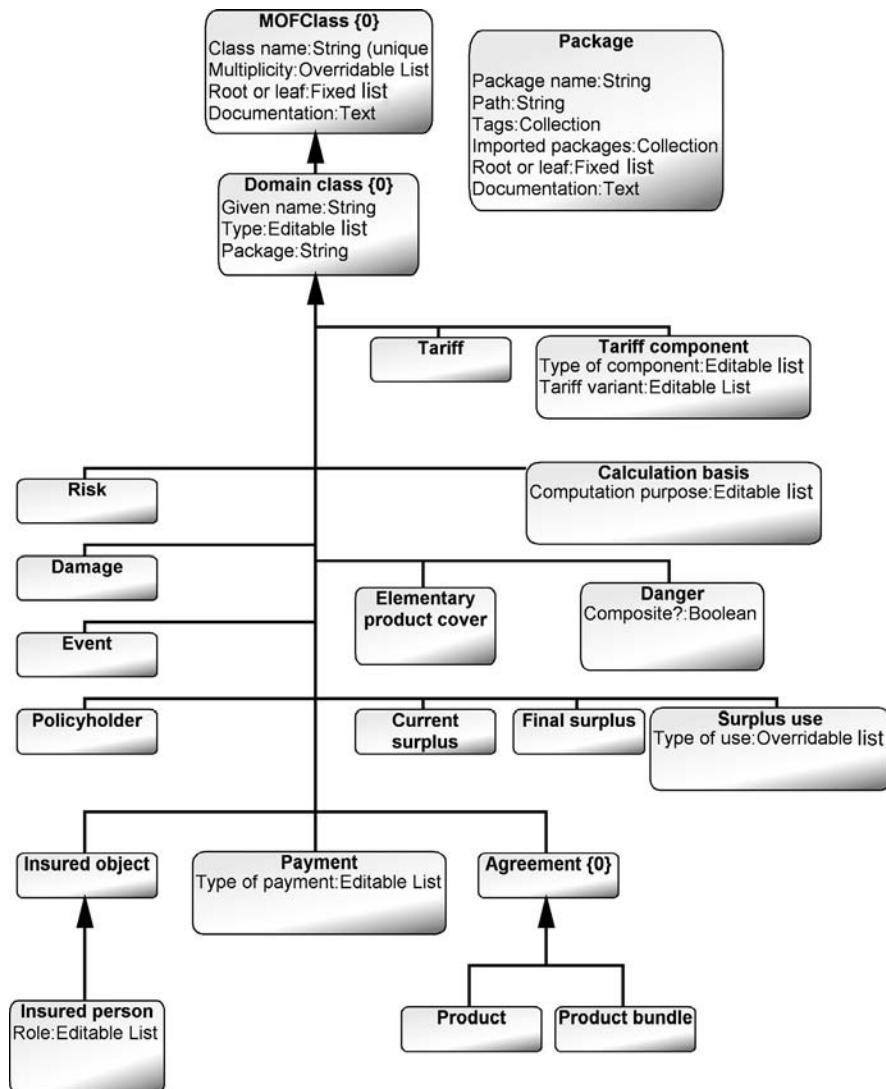


FIGURE 6.3 Insurance-specific objects inheriting from MOFClass

Each modeling concept needed a reference property type for specifying the package to which it belongs. This property was added to the common supertype Domain class (see Fig. 6.3). Its data type was made an object so that it referred directly to a Package model element instead of having just a mapping with a string value of the package name. This package reference was needed only for cases where a product specification represented in a single diagram had elements from multiple packages. Otherwise, all elements in a diagram belonged to the same package: the model hierarchy specified the package and product they belonged to.

6.3.2 Modeling Rules

In addition to object types and property types, various relationships and their related constraints were identified and defined. Depending on their nature, the constraints were either implemented directly into the metamodel or checked using generators. Most of the relationships deal with insurance-specific rules, such as that insured objects can be related to risks and that surpluses can be connected to tariff components. Figure 6.4 illustrates some of these relationships, such as that a Product bundle can consist of Products, which can further consist of either Elementary product cover or Tariff elements. This allows us to describe insurance products from a tariff-centric or a product cover point of view.

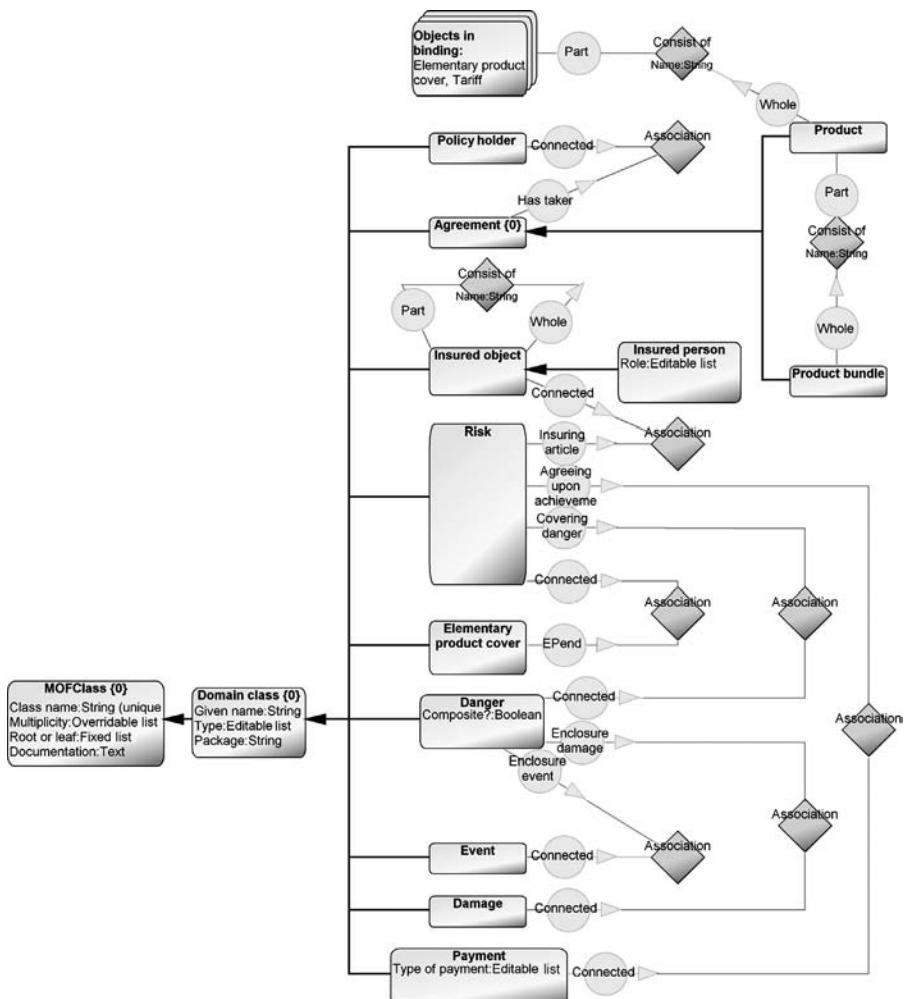


FIGURE 6.4 Relationships between insurance object types, Part 1

Rather than using different relationship types for connecting specific object types, the relationship types from MOF, namely, Association and Aggregation, were used. When the metamodel was used in the modeling tool, the right relationship type was chosen by the tool. If multiple relationship types were possible, the modeling tool asked the user to choose among the possible ones.

The legal connections were specified in the metamodel by using specific role types. Most of the role types were again inherited from the MOF AssociationEnd, and therefore had property types like name, multiplicity, and navigability. The latter two had predefined values for speeding up the modeling work. For multiplicity, the most typical values (0,1; 1,1; 0,M; 1,M) were added to a predefined list for quick selection. For navigability, the selection list was defined as having a mandatory value with a default value of having no navigation. The other possible values that a modeler could choose were restricted to “Is navigable” and “Is navigable and references.”

All the relationships were defined as binary, although n-ary relationships would require less modeling work: with an a-ary relationship just one aggregation

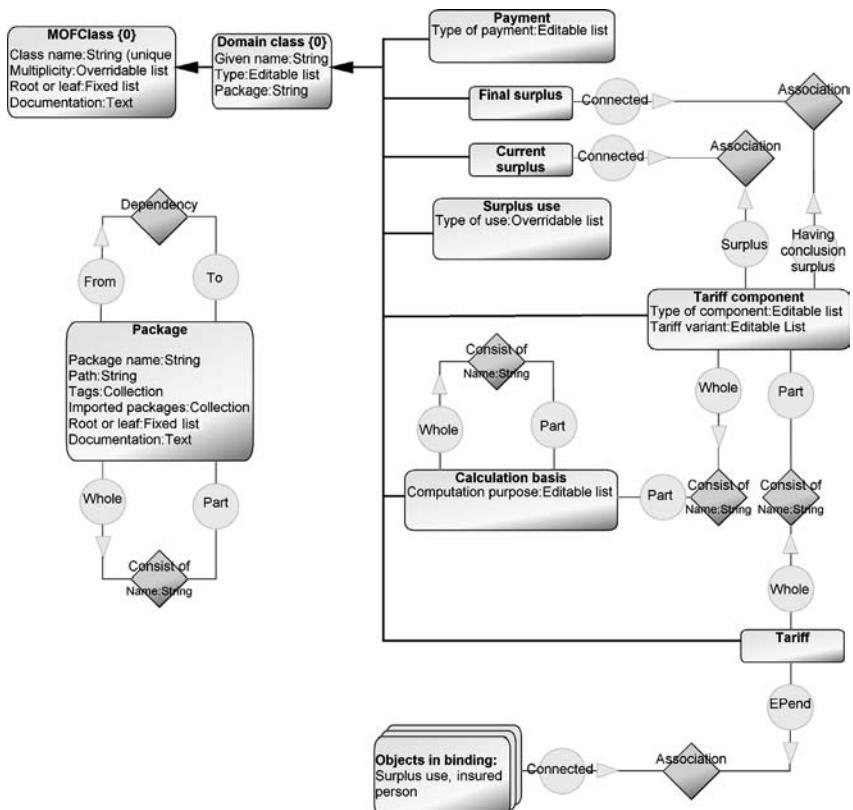


FIGURE 6.5 Relationships between insurance object types, Part 2

relationship between the Product bundle and Products would be needed, rather than drawing each aggregation as a separate relationship and filling the same aggregation information multiple times. Since the users of the language were not experienced in modeling, the choice to make model creation simple was emphasized. Figure 6.5 represents the remaining object types and their connections.

Along with the bindings between the object types, role types, and relationship types, constraints related to these domain concepts were defined. These constraints added to the metamodel and include the following:

- A Danger could be connected to only one Event.
- A Danger could be connected to only one Damage.
- A Product can have only one Insurant.
- A Product bundle can have only one policy holder.
- A Risk can be connected to only one Danger.
- A Risk can be connected to only one Payment.
- A Risk can be connected to only one Insured object.
- A Tariff component can be connected to only one Surplus.

Implementation of the metamodel also revealed some mismatches in the original definition of the metamodel. These were partly because MOF was used. For example, the inheritance relationship originating from MOF was changed in the metamodel so that inheritance was only possible between similar types of domain elements: risks could inherit properties from another risk element but not from a product or an insured person. Implementation of the modeling language also revealed missing information from the domain model, such as which properties need to have values, which are legal data types, and which parameters (inherited from MOF) must be defined as returns. All the additional constraints were added to the metamodel and the original domain model was not updated. The metamodel of the modeling language became more detailed and precise. It could also be tested as a language once instantiated, unlike the documentation of the domain model. Testing of the language definition was done by modeling the reference applications and generating their implementation code.

Model Hierarchy To model large or complex insurance products, the modeling language was extended to support model hierarchies. This was done by using a package concept: each package could be described in detail with a submodel. The submodel was based on the same modeling language, so all the modeling concepts available in the higher level model could be used in the submodel. In the metamodel, an optional decomposition link was defined from each Package object type to the product modeling language (see Fig. 6.6). In addition, package structures could be specified in one diagram using a Consist of relationship, and the language also had support for specifying dependencies among packages. These latter two are described in the metamodel represented in Fig. 6.5.

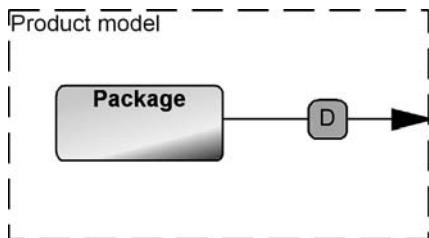


FIGURE 6.6 A part of the metamodel specifying model hierarchies by allowing a decomposition from an object type *Package* to the Product modeling language

Model Checking Not all domain rules could be included in the metamodel since their checking would not make sense at modeling time. For example, each insurance product needs to include at least one elementary product cover, but this kind of rule can't be checked at modeling time, since immediately after adding a product object to the model the design would be invalid. Various generators were implemented for checking such rules. These included, for example, that every Product cover needs to refer to a Risk object and that each Product bundle must refer to at least one Product. Although these model checking topics were detected while creating the metamodel, they were implemented last when it was possible to test them by using the language to specify some insurance products. The available models then acted as test material for the model checking generators. In addition to model checking, additional generators were made for producing documentation and generating an overview in HTML format.

6.3.3 Modeling Notation

Since the modelers were not software developers, language visualization (e.g., the visual appearance of the notation), ease of use, and user friendliness were emphasized. To gain better acceptance for the introduced language, the notational symbols were asked to be defined by the users of the language, the insurance experts. Most notational elements were taken from existing signs related to events, risks, and payments, such as traffic signs and currency. Figure 6.7 shows the notation for some modeling concepts when using the language to create specifications.

A specific question on the notation dealt with showing detailed properties of model elements. Since insurance elements usually had 5–10 properties, showing all would make large symbols that took most of the modeling space. Instead, the functionality of the modeling tools was used to hide the details and make them available via browsers and dialogs related to the model element. The symbols were used to show just the most important information, such as name, given name, and composite information. Since the modeling concepts were new in the beginning, each symbol showed its type name too. Later, once the language had been used, this extra cue for identifying the different types was removed.

Since it was not always desirable to specify inheritance with an explicit relationship, each domain element was modified by adding an inheritance property type. Its value was then the possible supertype. This property type was defined as an object, instead of using name matching, so that the details of the supertype could be

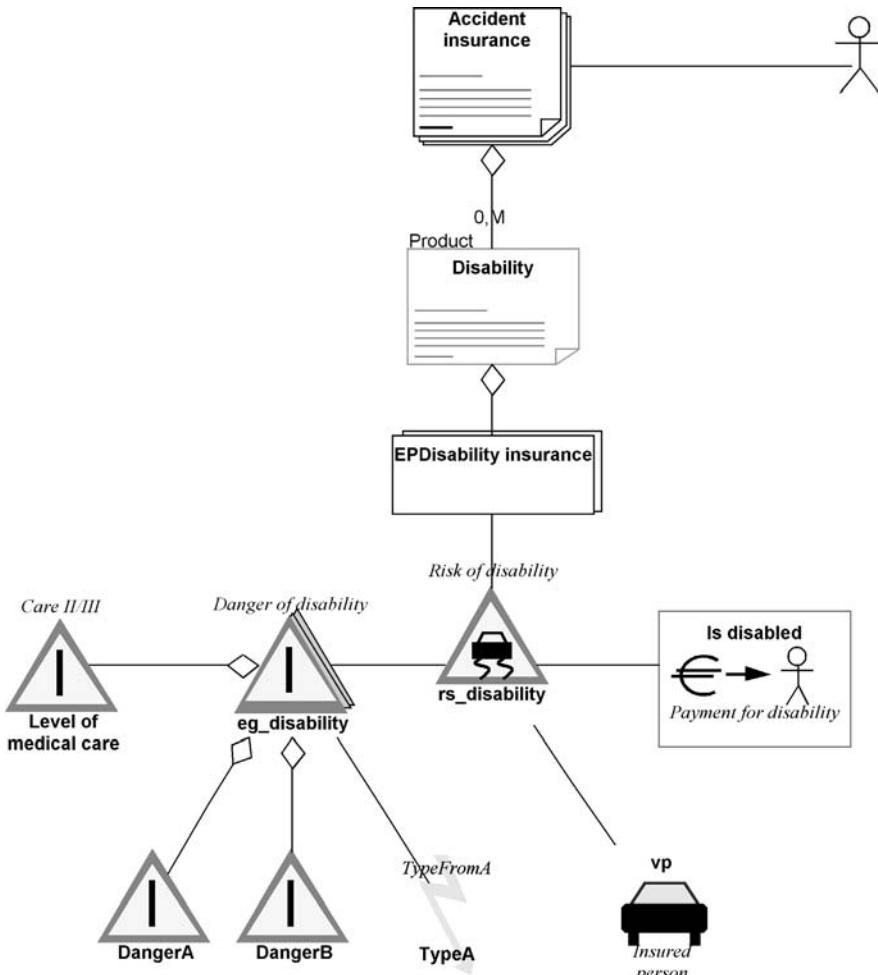


FIGURE 6.7 Sample insurance product (partial model)

viewed during model creation. This language structure also allowed creating new supertypes where such did not yet exist.

6.4 MODELING INSURANCE PRODUCTS

6.4.1 Example Models

The defined modeling language was used to specify the characteristics of products, and therefore, the language was called a product modeling language. The first models created were related to testing the language during its definition. The language was tested in a realistic situation by specifying the insurance applications already implemented by hand in Java. Figure 6.7 illustrates the specification of one small insurance product.

6.4.2 Use Scenarios

The modeling language was used by insurance experts. These nonprogrammers drew models similar to Fig. 6.7, specifying insurance products, and then executed a generator to produce the required code for a J2EE web site. The language clearly raised the level of abstraction since insurance experts could apply the already known insurance concepts. Actually, in the beginning the modelers did not even realize that they also generated the Java code. Later we will discuss the code generation in more detail.

During the introduction of the DSM solution a 2-day course was given to train the insurance experts in using the modeling language. For the training, the sample reference application was used. After using the DSM solution, the number of created specifications increased quickly, and after 6 months several hundred products had been specified covering well over 500 risks. To cope with similarities among products, a template specification was created for each insurance type to be used when specifying similar products from different insurance companies. Product templates were used as incomplete specifications and models made from them could be changed as needed while creating the final specifications.

Evaluation of the Language Later, analysis of the models revealed that similar patterns occurred in models depending on the insurance type, for example, indemnity products were modeled differently from life insurances. To minimize the modeling need, it would have been possible to create metamodels for each insurance type. Although the effort needed to create and maintain multiple similar languages would not have been large, the company decided to use just one language to specify all the insurance products. It was thus accepted that modelers needed to draw some structures almost identically for all insurance products of the same category. The tool helped here as it allowed the same structures to be copied from a template library either by value or by reference.

Since the metamodel also included MOF concepts, more for “standard compliance” than real need, it was unnecessarily large. For example, none of the product specifications used a generalization relationship, a concept taken from MOF. The modeling tool allowed removing the unused concepts or just hiding them from the modelers. Particularly useful was that this was possible even when the language was already used to specify dozens of insurance products.

6.5 GENERATOR FOR JAVA

The target platform included a quotation engine that used the information from the models in the form of Java code. Later, the Java code was expected to be replaced by XML. With a DSM solution, this change was not seen as problematic since it would only require changing the generator. Just one person, the generator developer, would have to work to make a new generator allowing generation of the desired XML from the same specification models. The users of the modeling language would not even notice the change in the generated output!

The main starting point for the generator definition was the expected output: the code that was written manually before considering a DSM solution. Although the manually written code was taken as a requirement and used as a test case for the generator, its structure was not implemented in the generator: the generator produced the same functionality but in a different structure from the manually written code. The reasons for not following the manually written code were twofold: the manually written code was unnecessarily long and inconsistent. The latter was especially relevant since similar kinds of features were sometimes written differently even inside the same insurance product. This was partly expected since the company had only specified a few insurance products, and both Java as a programming language and the way to implement them were new. The code was also unnecessarily complex since many classes were implemented in the code by first declaring them and then later defining them. For these reasons, the reference applications were unified to follow one set of best practices instead of using different styles and personal preferences. Let's look next at the structure of the code generator and then samples of the generated Java code.

6.5.1 Generator Structure

Figure 6.8 illustrates the structure of the final generator by describing the generator modules and call structures between them. The numbers on the lines connecting the generator modules indicate the execution order of the generator modules: the first, 1, produces the main class based on the package, and the last, 12, the associations between model elements.

Implementation of the generator, however, did not follow the execution order. Instead, the implementation started from the main insurance product concept that produced the main product definition. This was followed by implementing each individual domain concept based on the metamodel structure, that is, first, all elements related to the product concept were handled and then following the connections they have until all domain concepts were handled. Finally, generators for producing the common infrastructure code, such as package, importing, and data-type definitions, were implemented. This order did not allow testing of the generated code in the target environment since not all infrastructure code was available. This approach was partly a result of using an external consultant to make the generator, without access to the target environment for testing purposes. Therefore, generator testing during development was done solely by comparing the generated code to the manually written reference applications.

The generator was structured so that the handling of domain concepts was checked at the generation time: when MOF classes and their related attributes were generated, the characteristics of the domain concept were used to change the code generation. In this way, for example, the risk related classes were created differently from the payment related classes although the generator modules were the same. This approach was therefore opposite to that of the previous CPL (Chapter 5) or other examples described here that have a generator module for each domain concept. In the insurance case the decision to use the same generator module for multiple domain concepts was

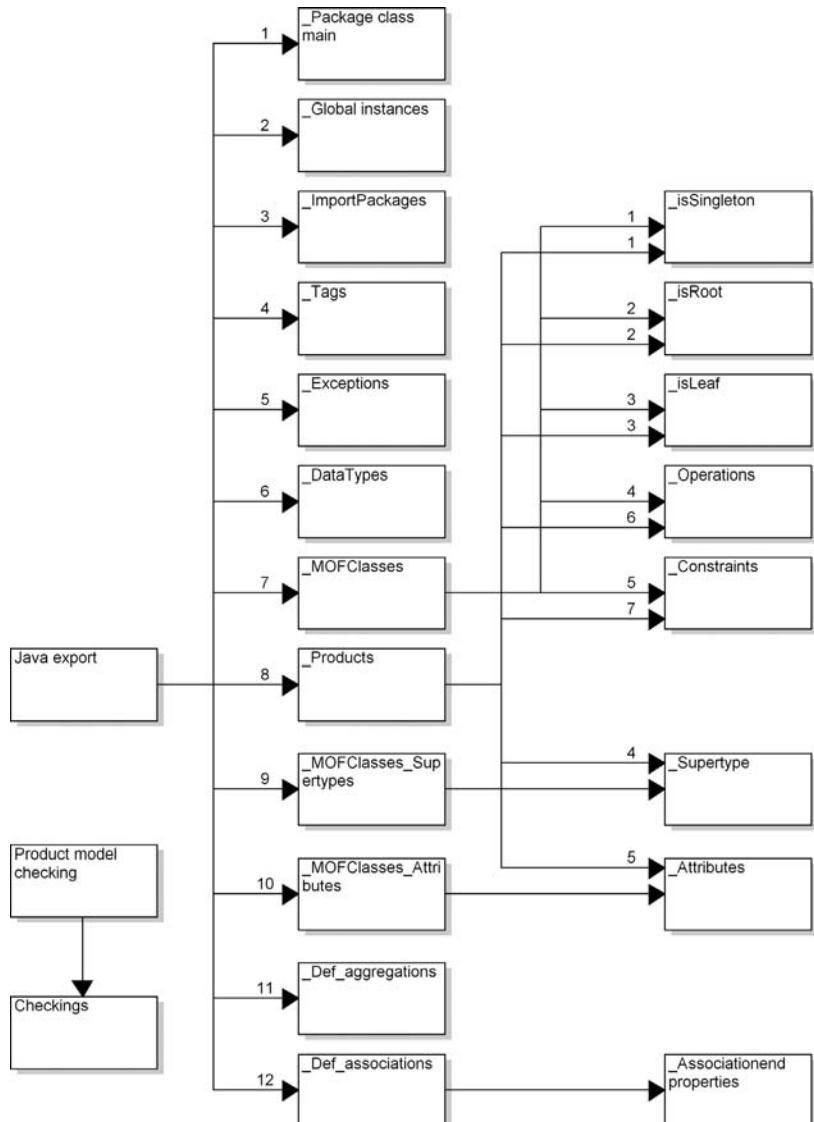


FIGURE 6.8 Structure of the generator

possible since the domain concepts were very similar and their main difference was among the legal connections they could have (see Figs. 6.4 and 6.5).

As shown in the generator structure, a product had its own generator module. The product concept was different from others since it created the product-specific Java class and acted as the main composite concept referring to other insurance-specific concepts. However, since the product concept was inherited from MOFClass and Domain class (see Fig. 6.3), similar to other insurance-specific modeling objects,

parts of the generators were the same. Therefore, those parts producing the common MOFClass related code were put in their own generator modules for reuse. These handled if the concept was a singleton, root, or leaf, and also produced code related to constraints and operations. Listing 6.1 shows the generator for producing the singleton code. In the generator definition the “multiplicity” string in line 3 refers to the property in a model element. Its value in the model element is then used to define if isSingleton is set true or false, “.setIsSingleton(true);” or “.setIsSingleton(false);”

Listing 6.1 A generator for producing singleton code.

```

01 Report '_isSingleton'
02 /*Singleton is true if multiplicity value is 0,1 or 1,1*/
03 if :Multiplicity = '1,1' or '0,1'
04   then '.setIsSingleton(true);'
05 else
06   '.setIsSingleton(false);'
07 endif
08 endreport

```

The names for the code, class name, attributes, operations, and so on, were taken directly from the names used in models. Another alternative considered was allowing the generator to produce unique names. This would work well for modelers since the insurance modelers were not expected to view the code they generated. For implementing the generator, however, it was considered better to use more descriptive names that could be taken from the model. Mapping names closely to the model also made the generators easier to test and later maintain. However, using just the name from the model was not enough since the same name could be used for several things, like the name of a cover and a related risk. For this reason, each name was extended with its type name. If no property names were used, for example, often AssociationEnd had no role names, the tool generated a unique name.

6.5.2 Generator in Action

The generator produces a Java class for each product, one file per specification. If the product specification was more complex, having multiple separate diagrams, all the diagrams in the hierarchy were used to create the class file. The class implements a product creation method that adds all insurance-specific details from the model to the product specification. In other words, it instantiates them as Java objects on the platform based on a predefined metamodel implementation.

From the product specification described in Fig. 6.7, the generator produces 962 lines of Java code. In production use, the average size of generated code was 2000 lines per insurance product and the largest was over 4000 lines. Next let’s examine the generator by looking at the generated code. We will only inspect some parts here: the main product creation code, code produced from a modeling object that is a domain-specific concept part of the insurance product, and code created based on the connections the domain concepts treated as objects in the model have. The sample code is taken from a generated insurance specification, 2188 lines of code in total.

Listing 6.2 shows part of the generated code. Numbers at the beginning of the lines refer to the lines of the generated file. This code is produced by the generator “_Package class main” (see Fig. 6.8). The string “Basis” in line 15 refers to the product name taken from the Product object and its property Given name (see the metamodel in Fig. 6.3). Similarly, the string “BBBasis” in lines 17 and 18 is taken from the name property of the Package object that refers to the insurance specification. All the rest is produced by the generator since they are common for the implementation.

Listing 6.2 Produced code for a product called BASIS.

```

15 public class Basis extends ProductRepository
16 {
17     private static BBBasisProductRepository instance;
18     private BBBasisProductRepository (String name)
19     {
20         super(name);
21         MofPackage productpackage = createProduct();
22         this.addMofPackage(productpackage);
23     }

```

This code is followed by the definition of the variables for each domain-specific type used in the specified product. The generator named “_Global instances” (Fig. 6.8) created the necessary global variables. To follow naming conventions, the variable names had to be written starting with a small letter. For such a convention, the generator could either translate each name for a required naming convention or just add an arbitrary lower case letter in front of the name. Note that it was not possible to ask the user to give a name correctly for code generation or place in the metamodel a rule that the name start with a lower case letter since the same value is used elsewhere starting with a capital letter. Having two properties for each domain object type (one for the type name and the other for a variable name) was not seen as a viable option.

After having produced the code for variables, imported packages, tags, exceptions, and data types, code for each element of the insurance is generated. Listing 6.3 shows the generated code for one danger. The code generator simply iterates through all the domain concepts as modeling objects in the models.

Listing 6.3 Generated code for a danger “extended notification period”.

```

122 // ***** Elementary Danger Extended Notification Period *****
123 elementaryDanger = new ElementaryDanger
("extended_notification_period");
124 elementaryDanger.setGivenName("Extended Notification Period");
125 elementaryDanger.setIsSingleton(false);
126 elementaryDanger.setIsRoot(false);
127 elementaryDanger.setIsLeaf(false);
128 elementaryDanger.setDomainType ("extended_notification_period");
129 selectionViewFalse_.addTaggedModelElement(elementaryDanger);
130
131 productpackage_.addMofClass(elementaryDanger);

```

This code maps to one object of the modeling languages referring to a domain concept Danger (see the metamodel in Fig. 6.3). Line 122 shows the comment using the type name of the model element and its given name. This particular danger is related to the extended notification period and this name is taken from the model and used in different conventions. Lines 125–127 are produced based on the MOF compliance for inputting the specification into the repository. The value “false” is taken from the property value of the respective model element. For example, the code for the singleton in line 125 is produced by the generator shown in Listing 6.1.

Once the domain concepts and their attributes and operations are produced, the generator starts to navigate through any relationships the modeling objects have. Inheritance relationships (or property values for each domain concept) are already used if they existed, but the main relationships usually dealt with the aggregations and associations each object may have. To avoid reporting the same relationships multiple times, the generator started the navigation from relationships rather than from the objects. Listing 6.4 shows the code for an association between a particular risk and danger. This type of relationship was already defined in the metamodel during language design; see Fig. 6.4.

Listing 6.4 Code for specifying association for a risk and a danger.

```

1561 // ****
1562 // Associations
1563 // ****
1564
1565 // ***** RiskRIS_dailyallowance_X_CompositeDangerCGF_basis ****
1566 risk = (Risk) productpackage_.lookupElementExtended
    ("RIS_dailyallowance");
1567 AssociationEnd end19_12014 = new AssociationEnd("end19_12014",
    iRisk);
1568 end19_12014.setMultiplicity(new MultiplicityType(1,1, false,true));
1569 end19_12014.setAggregation(AggregationType.SHARED);
1570
1571 compositeDanger = (CompositeDanger) productpackage_.
    lookupElementExtended("CGF_basis");
1572 AssociationEnd end19_7992 = new AssociationEnd("end19_7992",
    iCompositeDanger);
1573 end19_7992.setMultiplicity(new MultiplicityType(1,1, false,true));
1574 end19_7992.setAggregation(AggregationType.SHARED);
1575
1576 mofAssociation = MofAssociation.createAssociation
    ("Risk19_9291_X_CompositeDanger19_10072", end19_12014, end19_7992);
1577 productpackage_.addMofAssociation(mofAssociation);
1578
1579 reference = new Reference("Risk19_9291_X_CompositeDanger19_10072",
    end19_12014, end19_7992);
1580 risk.addReference(reference);

```

Lines 1565–1569 specify the association a particular risk has and lines 1571–1574 specify the other end of the association, connecting to a danger. The string value consisting of numbers is a unique name for that association end. The tool needs to give

names for associations since role names are neither mandatory nor unique in a model. Unique names are used in line 1576 to create an association by using the defined roles. The name for the association is again derived from the names of the objects types and their unique identifiers. Finally, if an object in an association had a reference, it was added to the object having the reference (line 1580).

The generator always produced the code in a standard manner: as was originally written in the reference applications. The only difference to manually written code was that the generated code used different naming conventions for some variables (generated unique names).

6.6 FRAMEWORK SUPPORT

Creation of the DSM solution was restricted by the already selected components of the portal: the repository to store the product specifications and its quotation engine to analyze and compare the insurance products. Since they could not be changed, the DSM solution itself needed to adapt to the existing target platform. This meant making the generator produce code as required by the target environment. No additional framework code was written and the input format itself formed the interface to the generation.

6.7 MAIN RESULTS

With the DSM solution, the company fundamentally changed its development process: the domain experts not only specified the products but also could test their work immediately by running the produced code in the portal. This was a big difference from the traditional method of first creating requirement documents, which programmers would then use to write the test cases, implement the code, and then finally test the results from a technical point of view and for compliance with the requirements. The CTO of the company saw the change as significant: “Traditional programming has largely disappeared and we can build systems up to five times faster with fewer errors.” These results were similar to the gains discussed in Chapter 2. In particular, the automatic generation made the development process easy and safe: modelers did not need to consider if the correct version of the supporting code was available during code generation.

The insurance experts could start using the modeling language and related tool relatively quickly—after 2 days of training. The main difficulties lay in learning to reuse existing product specifications or their parts and learning to use the MOF-specific parts visible in the modeling language. Later, analysis of the specifications showed that some of the MOF-specific concepts were not needed at all and thus could be removed: they just led to an unnecessarily complex language.

Particularly impressive to the modelers was their capability to generate working specifications that they could immediately see in the portal. In a similar vein, the sheer amount of code generated was considerable, 2000–4000 lines of code per product. This was almost completely a consequence of the input format used by the underlying

repository of the web portal. If the input format could have been changed and supported by additional framework code, the code generators would have been significantly simpler: now the generator produces a lot of MOF-related Java code needed just for the repository. Since the modeling language raised the level of abstraction, the company's plan to move from Java to an XML generator was seen as a minor issue: just one software engineer would be needed to modify the generator to produce a different output format. When planning the change of the target code to XML, it was also soon realized that the main generator structure (Fig. 6.8) would be largely the same—excluding only the MOF specific parts.

6.8 SUMMARY

The case of insurance product specification has shown how a DSM solution can be defined to raise the level of abstraction beyond the technical programming domain to the business domain. With the created DSM solution, nontechnical domain experts can create specifications using terminology they already know and generate the implementation code completely. The DSM solution also allowed the company to start defining the insurance products even when the underlying platform was not yet in use.

Since the company had already defined a domain model and had sample implementations, the DSM solution was largely defined and implemented by an external consultant. The modeling language was co-designed together with the main users, but its formalization as a metamodel and implementation into a modeling language were done by the external consultant. The code generator was developed solely by the consultant. This was possible since the sample applications provided reference implementations and test material for generator development.

CHAPTER 7

HOME AUTOMATION

In this chapter, we will look at an example of a Domain-Specific Modeling (DSM) solution for an embedded system that works closely with low-level hardware concepts. While the DSM solution itself succeeded in its aims, this case encountered a number of difficulties. Rather than only providing examples of the triumphal march of DSM, we hope that looking honestly at these problems will prove useful in helping you to avoid them. Names and minor details have been changed to protect the innocent.

7.1 INTRODUCTION AND OBJECTIVES

The company, which we will call Domatic, worked as a co-manufacturer and solution provider, producing a variety of hardware and software products. The focus was on M2M, Machine-to-Machine communication, applied to domains including energy, home automation, telecommunications, and transport.

Domatic wanted to investigate DSM, to see if and where it could be applied in their work. They were looking for higher levels of productivity through automating parts of the production of software and configuration information. As Domatic had no experience of DSM, and indeed little of any kind of modeling, they engaged a consultant from a company experienced in DSM to help them perform a proof of concept. As an example domain they chose an existing home automation system. Although there were no plans to build a large range of new variant systems in that domain, a few variants already existed, so it seemed a good candidate domain for DSM.

Domain-Specific Modeling: Enabling Full Code Generation, Steven Kelly and Juha-Pekka Tolvanen
Copyright © 2008 John Wiley & Sons, Inc.

7.1.1 Target Environment and Platform

The home automation system chosen for the proof of concept offered control of a range of devices including heating, air conditioning, lights, and security. The focus for the proof of concept was on a telecom module, which allowed remote control of the system over a phone line. In addition to remote control, the telecom module also allowed the remote update of its software, and commands from the main home automation system to dial out to a remote number to report alarms or log other data. The module had already been designed and built, and a few variants of it had been made as part of products for different clients.

The telecom module was operated remotely by a normal call from a phone. The module used voice menus to provide information and offer the user choices, which he could activate by pressing buttons on the phone keypad. The module used a standard telecom chipset to recognize the frequencies of the DTMF tones and translate them back into the simpler form of which button had been pressed.

The voice menus used real speech, sampled and stored in the module. As this was an embedded device, the speech was broken down into reusable sampled units of words or phrases to save memory. An actual sentence was played back as a sequence of these samples.

Clients supplied a sketch of the desired voice menu, for example in simple flow charts. These were fleshed out by Domatic into a spreadsheet format which added the technical details. For instance, sentences were broken down into sample units, and the choices were implemented as jumps to another row in the spreadsheet. Each row of the spreadsheet represented a certain memory address containing one primitive command: play a certain voice sample, jump to a certain memory address, assign a value to a register, and so on. Listing 7.1 shows the spreadsheet for a loop that reads out all five modes in the system, and tells the user which button to press for each.

Listing 7.1 Spreadsheet to read out the list of modes.

Address	Command	Argument
00A1	Load A	00
00A3	Add A	01
00A5	Say	'For'
00AE	SayMode A	
00AF	Say	'press'
00B8	SayNumber A	
00B9	Test A <	05
00BB	IfNot	
00BC	Jump	00A3

As the listing shows, the spreadsheet forms an assembly language program. An in-house assembler processed the spreadsheet into a binary file that implemented the program, running on an 8-bit microprocessor. As opposed to third-generation programming languages such as C or Java, assembly languages are specific to a given microprocessor, and sometimes also to a lesser extent to a given domain of use. This in-house assembly language included a variety of "Say" commands,

which would play a sample. Most samples were specified simply by memory address index and length: the actual samples were burned to an EEPROM. For some frequently used samples, a specific shorter command could be used, for example, “SayNumber B” to play the sample corresponding to the value of register B: “one” for 1 and so on.

7.1.2 DSM Objectives

Unlike other cases, there were no clear objectives for a DSM solution in this domain. The main goal was to use this example to examine the applicability of DSM in low-level embedded software development in general. As Domatic produced solutions based on other companies’ requests, the actual domains varied with each new customer. An important goal was therefore the ability to quickly create a new DSM solution, including the modeling language, generator, and tool support.

Domatic used no specific method for software development. Their developers would sometimes draw simple flow charts or state diagrams, either before or after they wrote the code. Reuse of code from older projects followed the “industry standard” practice of simply copying whole code files and changing parts. Recognizing the problems inherent in this approach, Domatic hoped that DSM solutions would increase the consistency of their software development and the reusability of designs and code.

As their current development relied largely on *ad hoc* or *post hoc* documentation and testing, Domatic were also interested in the fact that DSM models were at a high enough level of abstraction to serve as a communication medium with clients. The models could serve as the formal requirements specification, and at the same time as internal design documentation. Through code generation, the models could also be immediately tested.

7.2 DEVELOPMENT PROCESS

The DSM solution was developed in MetaEdit+ 3.0 by a consultant from MetaCase and an expert developer from Domatic. The consultant supplied the DSM know-how and actually built the metamodel and generators. Domatic supplied the understanding of the domain and the required code, and also made an extension to their spreadsheet assembler. The development of the DSM solution set out to follow the process for a proof of concept described in Section 13.3. As we shall see, however, not all went according to plan.

7.2.1 Before the Workshop and Day 1

Domatic had supplied the consultant with material about their domain and language three weeks before the workshop. The material covered the whole home automation domain, focusing on the telecom module. A week before the workshop they

emphasized a particular description of the whole home automation system and how it interacted with its sensors, actuators, keypad, screen, data modem, and DTMF voice control.

The first day of the workshop was spent building up a shared picture of this wider domain, resulting in a modeling language containing concepts like sensors and actuators. By the end of the first day, it was apparent that this language was too generic. Just knowing that there is a sensor called “smoke detector” connected to the system, and an actuator called “fire alarm”, is not enough to generate meaningful code. The language would thus be useful for describing whole systems, and possibly for configuration, but not for demonstrating DSM with full code generation.

7.2.2 Day 2: If at First You Don’t Succeed...

The second day of the workshop had the hard deadline of a meeting at 1 p.m. to present the results to management. The first part of the morning was spent establishing the area of the domain to be covered. Rather than focusing on the boundary, which is hard to lay down precisely at an early stage, the consultant and Domatic experts identified the central concepts of the domain. The modeling language had to be able to specify a section of voice output built up from text fragments, and a choice based on DTMF input. A small modeling language for this was built in MetaEdit+ in 25 minutes, 10:40–11:05.

Using this VoiceMenu modeling language, Domatic built a small example model and sketched the corresponding code. As both the modeling language and the assembly language were specific to the same narrow domain of Domatic’s telecom module, there was a good correspondence between model elements and lines or blocks of code. The consultant could thus build a basic code generator for the skeleton modeling language in 10–15 minutes.

In the remaining time up to 11:40, the modeling language was extended to handle more than the core cases: what to do when the user did not follow the voice instructions or when the voice output varied according to the state of the system. Concepts and control paths were added for timeouts and invalid input in DTMF, and for system calls to manipulate and test registers. Because of time constraints, the system calls were left free-form: the modeler had to know to use one of several possible assembler commands.

From 11:40 to around 12:00, the code generator was extended to handle the normal usage of these new concepts. The basic rules for how elements could be connected in the models were already specified along with the concepts, but there was no time for even the more obvious finer rules or checks. With an hour left until the meeting, there was time to finalize the example model, eat a hurried lunch, and prepare slides for the presentation to management.

7.2.3 Further Development

After the meeting, there was a little time left to refactor the DSM solution. The modeling language was split into two diagram types, with the top level showing

the voice menu and DTMF interactions. Each voice element there exploded to a lower-level diagram showing how it was built from static text fragments, varied by system calls. After the proof of concept workshop, the consultant finished this refactoring. He also added the missing parts of the code generator based on the sample code provided and sent the results to Domatic. These additions after the workshop took at total of two hours.

7.3 HOME AUTOMATION MODELING LANGUAGE

This particular DSM solution was never taken into wider use at Domatic, and thus has not experienced the normal evolution and rounding off of sharp corners. Some minor adjustments have been made to the version presented here to remove proprietary details. In this section, we will look at the modeling language and its metamodel; readers who prefer to see practice before theory can first take a quick look at the example models in Section 7.4.1.

7.3.1 Modeling Concepts and Rules

There were two modeling languages making up this DSM solution. The VoiceMenu language described the high-level interaction from the point of view of the caller. This language was thus useful not only for specifying the hierarchical structure of the voice menus, but also for discussing this structure with the client, or for providing documentation to the end users. Each part of the model that specified speech or system actions was further detailed in the lower-level VoiceOutput language. This language took the place of the earlier assembly language statements written in the spreadsheet.

Figure 7.1 shows the definition of the VoiceMenu modeling language. The main concepts are the VoiceOutput, where the telecom module says something to the caller, and the DTMF_Input, where the module waits for the caller to press touch tone buttons to make a choice. The normal flow is from a singleton Start object to a VoiceOutput, which gives instructions about possible choices, to a DTMF_Input that waits for input from the caller. The type of input expected is specified in a property of the DTMF_Input object as either “character” or “string” (for simplicity, we will concentrate here on single character input). For each possible input there is a ConditionalFlow relationship to another VoiceOutput. Mostly, this will be a test for equality with a given character specified in the ConditionalFlow, but slightly more complex conditions like \geq could also be specified.

If an invalid key is pressed there is an InvalidInput flow, normally back to the previous VoiceOutput, that is, the instructions for this choice. For cases where no input is received there is a Timeout flow, which specifies how long to wait before it is followed, again normally back to the previous VoiceOutput. A VoiceOutput can also be directly followed by another VoiceOutput, allowing reuse at this level.

In addition to the rules specified here about how objects can be connected with various flows, there are also some more specific constraints. As usual, a Start object can be in just one From role, to prevent ambiguity in the initial control flow. In a

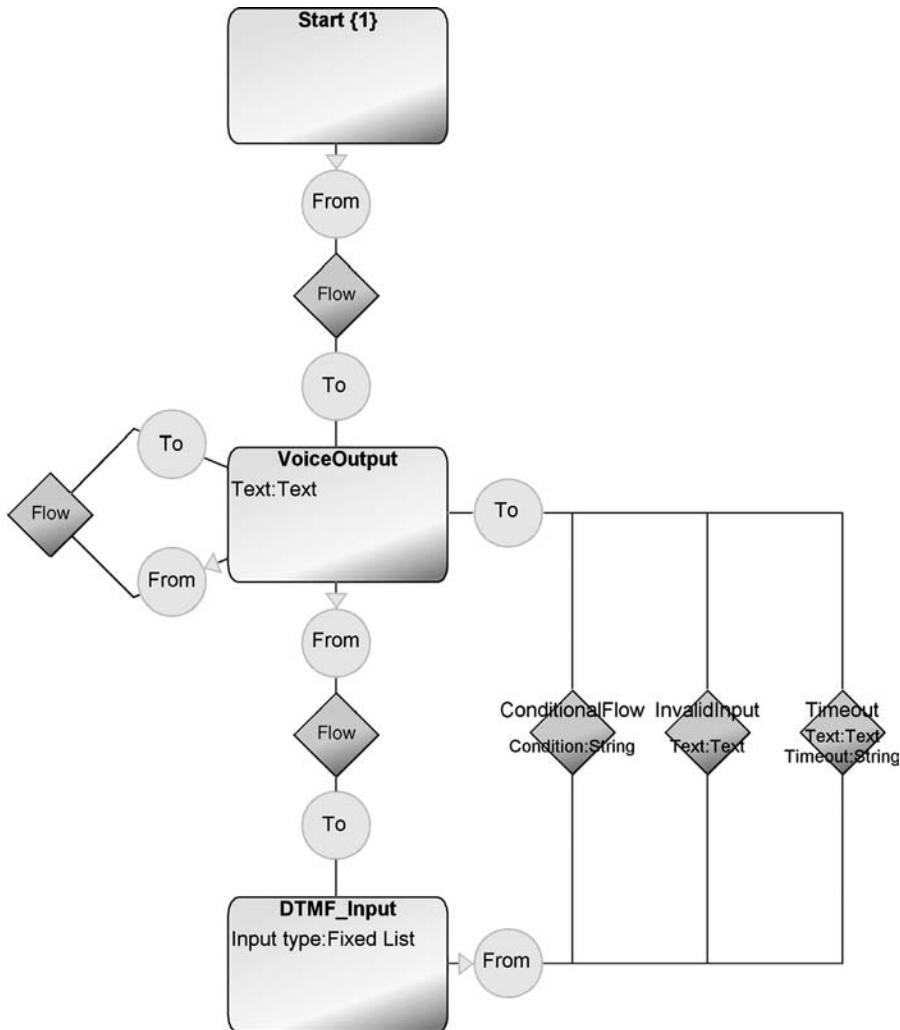


FIGURE 7.1 Top-level metamodel, VoiceMenu

similar way, a DTMF_Input object may only be in one InvalidInput and one Timeout relationship: there would be no way to choose between several. There will however normally be several ConditionalFlow relationships, as they each specify their own Conditions: the various keys that can be pressed.

Each VoiceOutput object, InvalidInput relationship, and Timeout relationship specifies in a lower-level diagram the actual speech it produces: the text property in the elements themselves is a description for the convenience of the modeler. The structure of explosions from the top-level VoiceMenu to lower-level VoiceOutput diagrams is shown in Fig. 7.2. Most often, the speech used for all InvalidInputs will be the same,

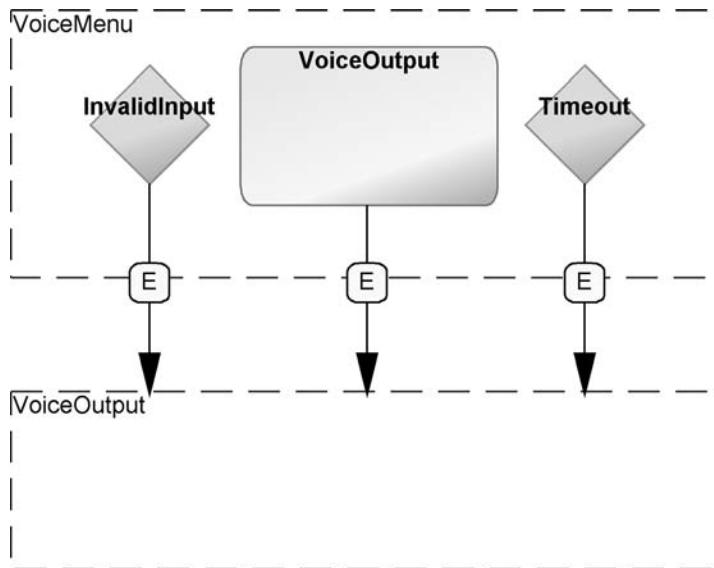


FIGURE 7.2 VoiceMenu elements with a VoiceOutput subdiagram

that is, there will be one “Invalid input” subdiagram, and each InvalidInput relationship will explode to that same diagram. The same applies to Timeouts, but each VoiceOutput object will generally have its own VoiceOutput subdiagram. As there is a limit to the complexity of a usable voice menu, no need was envisaged at this stage for an element in a VoiceOutput subdiagram exploding again to its own VoiceOutput sub-subdiagram.

The concepts of the lower-level VoiceOutput modeling language are shown in Fig. 7.3. The main elements of the language are the Text and SystemCall object types. A Text represents a sequence of TextFragments played one after the other, with no variation. A SystemCall represents a sequence of system commands: register assignments, special speech commands, and so on. The TextFragment and Command objects can only be used inside Text and SystemCall objects, not directly in the model itself.

In order to specify more complex flow control than simple sequential chains of speech and system commands, the language provides conditionals jumps with If and GotoPoint objects. An If object is made up of an Test such as "A >=" and a Parameter containing the value to be compared with. The condition can be inverted through the Boolean property, Not.

The elements in the diagram are mostly connected into a sequential chain of Flow relationships. As the top row of Fig. 7.3 shows, such a Flow can come from several different kinds of objects and go to a slightly different set. Start and Stop can only be in appropriate roles in such a flow, and If cannot be the source of a normal Flow relationship. Instead, If has two different relationships leaving it: True and False. If the result of the whole condition is true, control flow will jump to

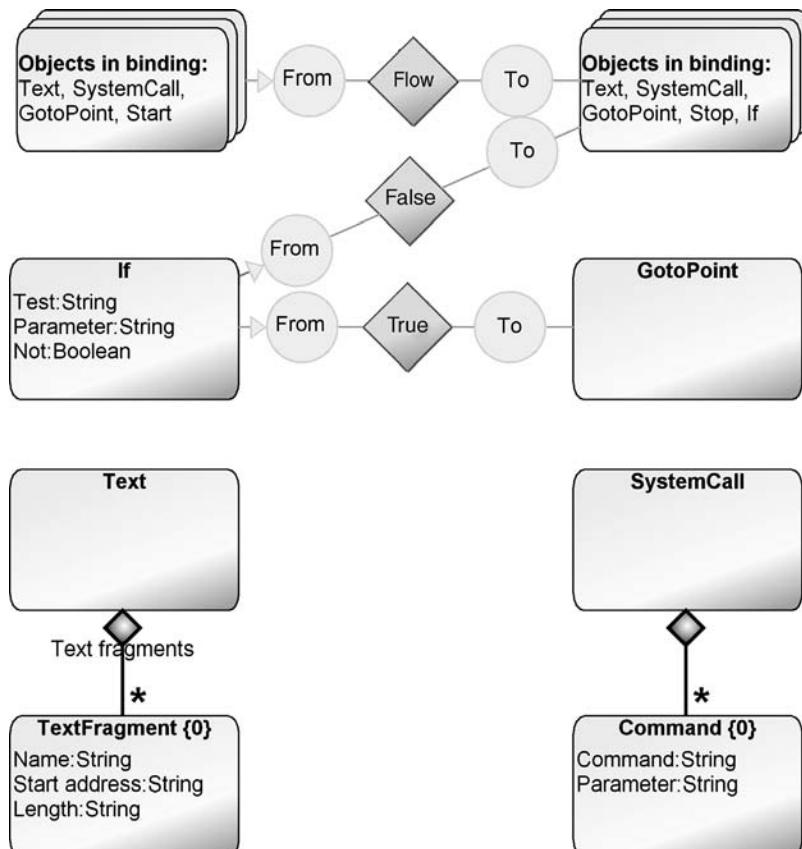


FIGURE 7.3 Lower-level metamodel, VoiceOutput

the GotoPoint at the other end of the True relationship. If the result is false, control flow will follow the False relationship to any of the normal target objects, just as in a normal Flow.

The If construct is thus not a full if..then..else familiar from third-generation languages, but a simpler conditional jump, as is common in assembly languages. GotoPoint has no behavior of its own: when included in a normal flow sequence control simply passes on to the next element. Instead, it serves simply as a label, the target for an If jump.

Once again, there are the normal rules for Start: one instance per graph, and only one From role per instance. This time, a similar constraint on From and To roles applies to most of the object types: only GotoPoint and Stop have no such restrictions. We can allow multiple Stops and multiple incoming To roles for each; the metamodel already prevents From roles leaving it. An If object should have only one True and one False relationship leaving it.

7.3.2 Possible Improvements

As this modeling language was made in such a short time, and has not been developed further, it is worth looking at some areas in which it could be improved. Some of the names for concepts could be fine-tuned, for example, GotoPoint might be better as “Jump Target” or “Label,” and Text should perhaps be “Speech.” These are however minor points, and easy to address at any stage—although it is worth noting that with some tools, changing the names of concepts can have catastrophic consequences: the next time the model is loaded, all instances of those concepts may disappear.

Perhaps the clearest problem is the repetitiveness of the InvalidInput and Timeout relationships in the VoiceMenu models (see Fig. 7.5). If in most cases the same structure will be in a model, but there may be some variation, it is better to assume the default and only use the structure where the model should behave differently. The norm for InvalidInput and Timeout is to return to the previous VoiceOutput after saying a simple message, so that should be taken as the default if these relationships are not specified. This would make the models faster to build, smaller, and easier to read, at no expense in terms of expressive power.

Another difficulty is specifying a watertight constraint for the False relationships leaving If objects. Most objects simply have one From role leaving them, but If can have two, one for True and one for False. Constraining to two From roles does not solve the problem: they could both be either True or False. We can constrain If to have only one True relationship, but the case of False is harder.

If we say an If can be in only one False relationship, we also exclude the possibility of an If followed by an If: the second If takes part in two False relationships, one incoming and one outgoing. The simplest solution would be to make the leaving role for the True case different by creating a new role type, for example, JumpFrom. This would allow us to specify that each If can be in at most one JumpFrom role (for True), and at most one From role (for False).

Looking at the False relationship, however, there may be more that we can do. The False relationship is actually no different from a normal Flow. It would probably be better to have just a normal Flow relationship for it, and add If to the set of source objects at the top of the figure. The True case would be distinguished by a different role. Since GotoPoint can allow several incoming roles, we should probably distinguish those in the normal sequential flow (To) from those that are jumps to this label (JumpTo). That would allow us to constrain GotoPoint to a single incoming sequential role—corresponding to the single line of assembler that can precede the label—but many incoming JumpTo roles corresponding to conditional jumps to the same label from multiple places in the code.

This would make the rest of the constraints more similar and allow the use of inheritance. Rather than having to specify constraints for each object type separately, we could give Text, SystemCall, If, and GotoPoint a common Abstract supertype, and

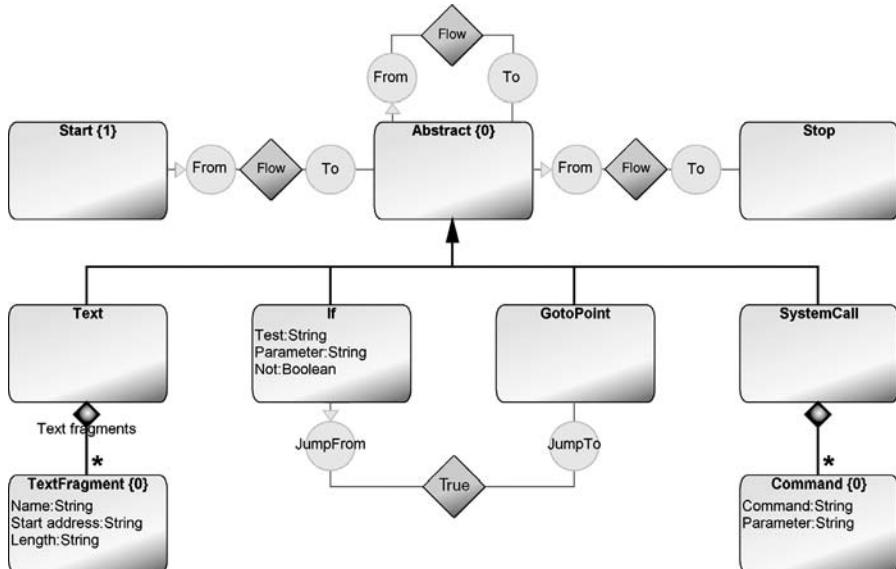


FIGURE 7.4 Alternative metamodel for VoiceOutput

specify the supertype in the bindings and constraints. This way there would only be three constraints: any Object could be in at most one From role, Abstract could be in at most one To role, and If could be in at most one JumpFrom role. The result of these changes would look like Fig. 7.4, but we will stay with the original metamodel for the purposes of this chapter.

7.3.3 Modeling Notation

As Domatic already used some simple flow chart symbols, the notation took these as its basis. Start and Stop were gray boxes containing their name, and conditional points such as DTMF_Input and If were represented as diamonds. In the top-level VoiceMenu diagrams, the schematic symbol for a loudspeaker was used to represent all items containing speech—VoiceOutput, InvalidInput, and Timeout. This was partly a concession to time limitations: separating these symbols only by color might be confusing, particularly since one is an object and two are relationships.

In VoiceOutput diagrams, speech segments were represented by a cartoon speech bubble showing the sequence of words or phrases to be spoken. SystemCall sequences were shown with a traditional flowchart symbol: a cut-off rectangle. The lines from If were labeled as TRUE and FALSE, with TRUE leading to a circle representing the GotoPoint.

7.4 HOME AUTOMATION MODELING LANGUAGE IN USE

For each home automation system type, there would normally be one set of diagrams specifying how the user could control it over the phone. At the top level would be a VoiceMenu diagram, and each VoiceOutput object in that could be exploded to its own VoiceOutput diagram.

7.4.1 Example Models

In our example application in Fig. 7.5, the telecom module responds to the call with the main menu: an initial welcome message and list of the options. To keep things simple, here there are only two options: pressing 1 takes the user to the mode menu and pressing 2 to the version info. The version info is simple: it reads out the version info and waits for the user to press 1 to return to the main menu. The InvalidInput and Timeout relationships are even simpler: each simply says “Invalid input!” or “Timeout!” and returns to the previous menu as shown.

The mode menu is more complex: it reads out the current mode and a list of all modes, telling the user which key to press for each. The DTMF_Input for the mode

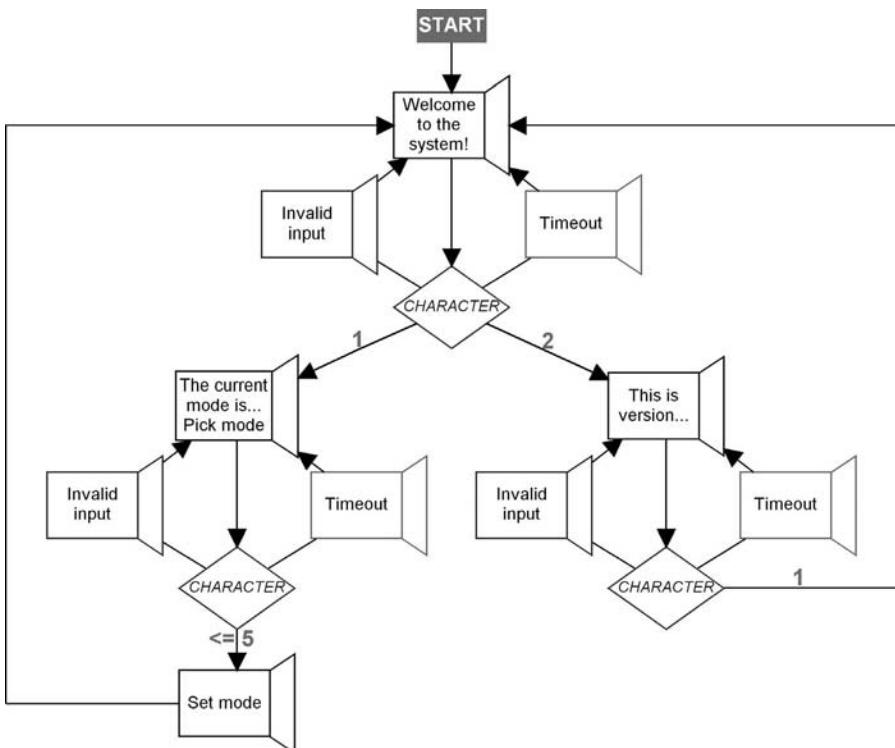


FIGURE 7.5 Sample VoiceMenu model

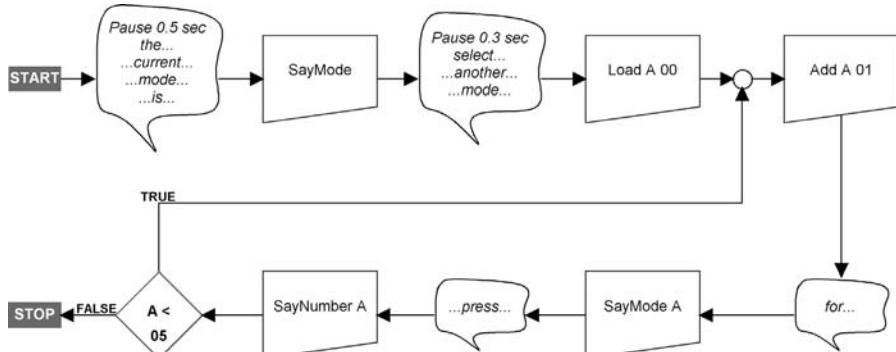


FIGURE 7.6 Sample VoiceOutput model for mode menu

menu allows the user to press a key corresponding to a mode. As there are five modes, the key must be from 1 to 5 (setting the mode to 0 does nothing). If legal input is received, the “Set mode” object’s subdiagram uses a SystemCall to change the system mode to match the key which was pressed.

The details of the mode menu are described in the VoiceOutput subdiagram in Fig. 7.6. After initially stating the current mode and telling the user to select another mode, the application initializes a counter variable, register A, to zero. After the GotoPoint, A is incremented and we move to the bottom row of the diagram, heading left. The number of modes, five, and their names are built into the system, so the system can say the name of the first mode and “press 1.” In the If object we check that A has not yet reached the value of the last mode, five, and if so we jump to repeat for the next mode from the GotoPoint. After the fifth mode has been read and the test in If fails, we exit via Stop back to the VoiceMenu diagram, where we wait for the user to press a key corresponding to a mode.

7.4.2 Use Scenarios

As we mentioned above, the two modeling languages aimed to provide a natural way to describe and specify the desired behavior of the voice menu, and of the voice elements and system calls that made up each segment of speech. The VoiceMenu language was specific to the domain of voice menus, which was a common basis shared by Domatic’s developers, clients, and clients’ end-users. It could thus easily be used as a medium of conversation between all these stakeholders, and allowed specification of systems at a high level of abstraction.

A likely use scenario would have been for a Domatic employee to work with a client to design the voice menu, directly using the VoiceMenu modeling language in the DSM tool. If example texts were specified in the top-level elements, a slight modification of the generator would allow working prototypes to be built and tested immediately.

The VoiceOutput language was based on the domain-specific side of the in-house assembly language, which was in turn based on the features offered by Domatic's hardware platform. This modeling language was designed for use by Domatic's developers, although simpler cases could be handled easily by nontechnical personnel. In the current state of the language, the direct inclusion of assembly language commands would have made using the whole language on more complex cases too complicated for anyone unfamiliar with the assembly language. Looking at more examples of the usage of that language would probably have allowed the use of higher-level constructs, for example, to replace the three steps in the above model—Load A 00, Add A 01, and A < 05—with a simpler single “For A = 1 TO 5” construct.

As things stood, the modeling languages would allow the creation of the complete range of applications that existed for that framework. Speech elements could be reused across multiple models, keeping memory requirements down in the finished product. As part of this reuse, it would be useful to know the total set of speech fragments used in a given application. This could be produced by a generator, guaranteeing that the set of samples for a product included all of those that were needed, and only those.

Often with reusable components, it is also useful to create an explicit library of reusable components. This helps prevent developers inadvertently reinventing the wheel because they did not know of the existence of a previously made component. This could be accomplished with a simple little modeling language that would contain a set of TextFragments. When developers wanted a text fragment, they would pick it up from the library, adding it there if nothing suitable existed. An example of such a library is shown in Table 7.1. Here we have included the start address, to fit with

TABLE 7.1 Library of Text Fragments

Label	Start address	Length
...press...	0×00	1
Pause 0.5 sec	0×01	5
the...	0×02	1
...current...	0×03	2
...mode...	0×04	1
...is...	0×05	1
Pause 0.3 sec	0×06	3
select...	0×07	2
...another...	0×08	3
for...	0×09	1
Timeout!	0×10	3
Invalid input!	0×11	5
Welcome to the system!	0×12	8
Press 1 for mode menu	0×13	7
Press 2 for other menu	0×14	8
Welcome to the other menu	0×15	10
Press 1 for the main menu	0×16	11
This is VoiceMenu v1.2	0×17	11

Domatic's practice. In actual use, it would probably be better to omit this and instead have the generator automatically create sequential numbering separately for each product. A useful addition would be a property for each fragment that pointed to the actual sound file, so developers could listen to it directly from the model and even emulate whole sections of speech.

7.5 GENERATOR

The generator produced the necessary code for the whole application in the assembly language that Domatic used. As the generator was based on existing best-practice

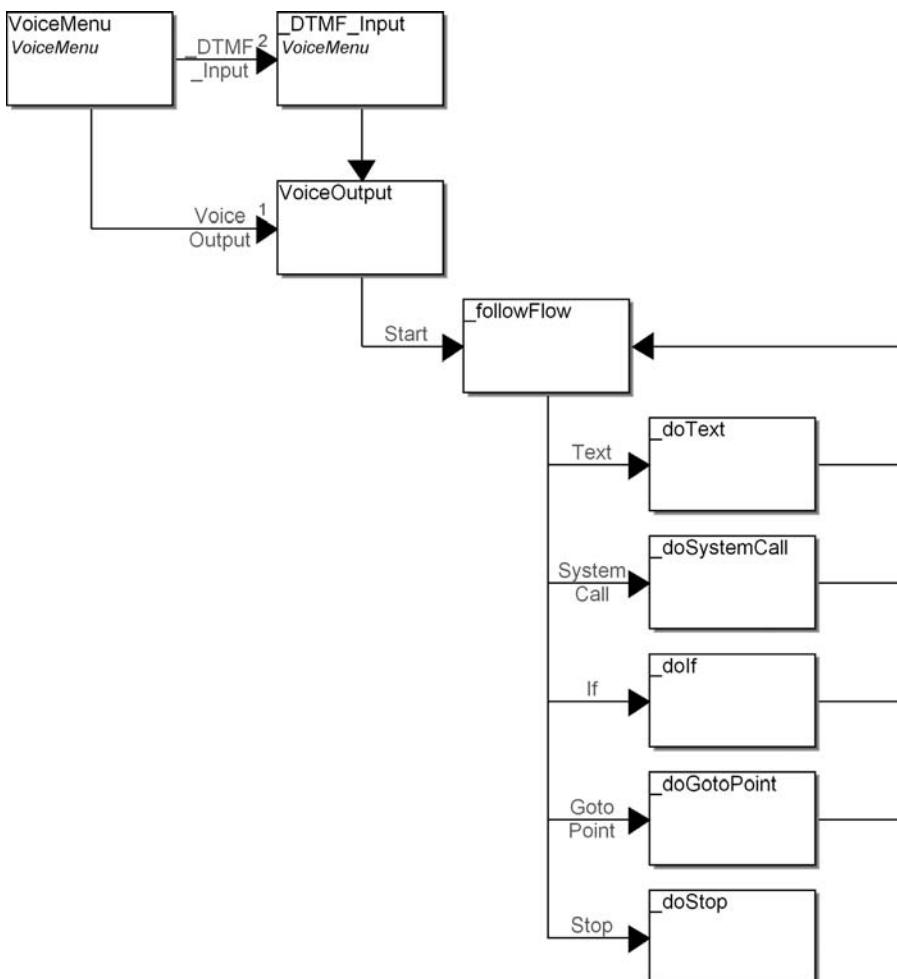


FIGURE 7.7 Home automation generator structure

code, the output was virtually indistinguishable from handwritten applications. One concession was made to the time constraints: it would have been hard to generate the correct absolute memory addresses for jumps, as this would have required calculating the byte length of each assembly instruction. Instead, labels were generated as part of the output, and jumps were directed to the labels. A quick change to the assembler made these jumps function properly.

7.5.1 Generator Structure

The generator was divided between the two modeling languages in the obvious way. Figure 7.7 shows the parts of the generator and the calls between them.

At the top level, a VoiceMenu generator started off the generation for the top-level VoiceMenu diagram, iterating over each VoiceOutput object and each _DTMF_Input object. The handling of DTMF input, invalid input and timeouts was all generated at this level. For the sample VoiceMenu from Fig. 7.5, the code output for the first VoiceOutput and DTMF_Input is shown in Listing 7.2.

Listing 7.2 Generator output for first VoiceOutput in sample VoiceMenu.

```
:3_266 /* code for "Welcome to the system!" */
  Say 0x12 8  'Welcome to the system!'
  Say 0x13 7  'Press 1 for mode menu'
  Say 0x14 8  'Press 2 for version info'

:3_306
  GetDTMF Timeout 5
  IfNot
    Jump 3_354
    Test DTMF = 1
    If
      Jump 3_450
      Test DTMF = 2
      If
        Jump 3_468
        Say 0x11 5  'Invalid input!'
        Jump 3_266
:3_354
  Say 0x10 3  'Timeout!'
  Jump 3_266
:3_450 /* code for "The current mode is... Pick mode" */
...
:3_468 /* code for "This is version..." */
...
```

The first block, labeled 3_266, is specified in the lower-level VoiceOutput subdiagram. 3_306 is the start of the DTMF handling: wait for DTMF input, timing out after 5 seconds (this period is taken from the Timeout relationship). If

we time out with no input, jump to 3_354, say the voice output specified in the Timeout relationship's reused VoiceOutput subdiagram, and jump back to the start of the first block.

The meat of the DTMF handling is a series of Test-If-Jump blocks, which compare the DTMF tones received with the characters specified in the ConditionalFlow relationships, jumping to the appropriate VoiceOutput block if there is a match. If no match is found, we fall through to the "Invalid input!" section at the end of the 3_306 block, which is generated similarly to the Timeout section.

Moving on to the VoiceOutput section of the generator, Listing 7.3 shows the output for Fig. 7.6. The first block shows the code for the flow up to the GotoPoint, which is label 3_844. The code after 3_844 is essentially the sample code from Listing 7.1: the generator fulfills its requirements.

Listing 7.3 Generator output for Mode menu.

```
:3_450
  Say 0x01 5  'Pause 0.5 sec'
  Say 0x02 1  'the...'
  Say 0x03 2  '...current...'
  Say 0x04 1  '...mode...'
  Say 0x05 1  '...is...'
  SayMode
  Say 0x06 3  'Pause 0.3 sec'
  Say 0x07 2  'select...'
  Say 0x08 3  '...another...'
  Say 0x04 1  '...mode...'
  Load A 00

:3_844
  Add A 01
  Say 0x09 1  'for...'
  SayMode A
  Say 0x00 1  '...press...'
  SayNumber A
  Test A < 05
  If
    Jump 3_844
```

The VoiceOutput generation was handled by a generator in the VoiceOutput modeling language. As the basic sequential flow control was the same for all object types, that was handled with one generic _followFlow generator. This followed the relationship to the next object and called the generator for that object type. The name of the subgenerator to be called is formed on the fly from the name of the type, allowing a new type to be added to the modeling language simply by specifying one small subgenerator for it. Listing 7.4 shows the generator

definition. A From role, Flow or False relationship, and To role are followed to the next object: (A|B) specifies either type A or type B, and () specifies any type. The name of the generator to call is formed from the output between the `subreport` and `run` keywords, that is, `_do` followed by the name of the object's type, for example, `_dolf`.

Listing 7.4 “`_followFrom`” generator definition.

```
Report '_followFlow'

do ~From>(Flow|False)~To.()
{
    subreport '_do' type run
}
endreport
```

The generator begins from the Start object and calls `_followFlow` from it. Each object type's generator handles its own line or lines of output and then calls `_followFlow` again, making a simple recursion step through the model. The generator for the Stop object does not recurse, and thus the generation finishes there.

The `_dolf` generator handles the FALSE path as normal sequential flow with the `_followFlow` generator. The TRUE path, however, is handled by outputting an assembly language test statement followed by a conditional jump to the label specified by the `GotoPoint`, without any recursion to follow the code on from the label. This avoids the problem of an infinite cycle or duplicate code (see Section 11.3.5 for more on this issue in general).

Text and SystemCall objects may contain several TextFragment or Command objects, leading to several lines of output, for example, the first five lines of Listing 7.3. These are handled simply by iterating over the contained objects as shown in Listing 7.5. The listing also shows the final `_followFlow` call.

Listing 7.5 “`_doText`” generator definition for Text object type.

```
Report '_doText'

do :Text fragments
{
    ' Say ' :Start address; ' ' :Length ' ''
    ' Say ' :Start address; ' ' :Length ' ''
    :Label '''' newline
}

subreport '_followFlow' run

endreport
```

7.6 MAIN RESULTS

The presentation by Domatic's experts to their management revealed that the proof of concept was partially successful. The modeling language and generators were seen to have accomplished the objectives set for DSM:

- The models visualized application structures well
- The modeling language and generators forced developers to do things right
- Applications which previously took a day could be made in an hour or two
- Better reuse possibilities within and across products
- The models provided consistent documentation
- Test plans could be integrated with models

However, Domatic's needs were not just for this single domain, but for applying DSM in a range of domains. The time constraint of effectively three hours meant that only simple cases had been handled in this domain, leaving uncertainty over whether DSM or the tool could cope in other domains. It is also likely that time constraints forced the consultant more into doing things himself, rather than having the time to explain everything and let Domatic's experts try their hand at doing things themselves. Developers are loath to tell management something can be done unless they are certain of their own ability to do it. Domatic's experts thus also raised a number of concerns that the proof of concept had not been able to answer:

- Coding such simple applications was not challenging for average developers
- Uncertainty about 100% code generation
- Uncertainty about backward compatibility with existing code
- Are code generator facilities flexible enough?

These are indeed often concerns after a proof of concept: even in the best case, in two days it is simply not possible to prove these beyond doubt. Normally, however, sufficient progress has been made that the experts feel these have been demonstrated as well as could be in the limited domain, and certainly beyond what they themselves had expected.

The extra two hours of work after the workshop are obviously not reflected in this bullet list from the presentation. This is unfortunate, as those two hours addressed three of the main concerns. First, the refactoring of the modeling language allowed the creation of more complex applications. Second, finishing off the code generator proved its ability to generate 100% code. Finally, these extra parts of the generator required some of the more powerful features of the generator facilities, demonstrating flexibility that had not been needed in the earlier parts.

7.7 SUMMARY

This case provides a good example of how DSM can be used at the lowest end of the abstraction spectrum, generating assembly language for an 8-bit microprocessor directly from high-level models. It is interesting to see that even here, the productivity increase fits within the normal range of a factor of 5–10. At first sight, there seems little benefit in replacing a line of code ‘Say “Press”’ with an object containing the word “Press.” Looking more closely, we notice that in both cases we need to also specify the starting address and length of the speech sample. If we were writing the assembly language by hand, we would have to look these up somewhere and copy them into the code. Here we can simply reuse an existing “Press” TextFragment object, and should any of its details change later, those changes will be automatically reflected here. Similarly, rather than having to fathom our way through the spaghetti of Goto jumps and labels, the models make the control flow instantly and intuitively clear.

The fact that the modeling language and generators were built in such a short space of time also provides good evidence for the use of efficient DSM tools. It is particularly important to be able to quickly iterate through several cycles of adding a small part to the modeling language and testing it, without existing models being rendered unusable.

We can learn several lessons from the less successful parts of the case. With regards to the first, abortive, day of the workshop, there is always a danger when trying to build a modeling language that we invent something that is really more of a model, or simply allows us to draw architectural diagrams. A good rule of thumb to apply is whether we can realistically generate code from the models made with this language (assuming code generation is desired). Another test is how many models of this kind would we be likely to build. If the answer is “one diagram per product”, the language does not go deep enough.

Second, even if people talk about the choice of example domain not being important, it still has a large effect on how the overall feasibility of DSM is perceived. Seeing DSM work in one domain holds little credibility, if that is not a domain the customer intends to use it for. People are naturally afraid that the actual domain they want will turn out to be more complicated than the example shown. Looking at examples from other companies helps little: for instance, Domatic were familiar with Nokia’s use of DSM, but dismissed cell phone development as not being true embedded development—far too simple. The truth of the matter is that most development, when looked at from 10,000 feet, is simple: the complexity is often in the languages, frameworks, and tools we use. The power of DSM is that it can leverage the abilities and experience of an expert developer to create a development environment that more closely resembles the view from 10,000 feet.

Finally, DSM is a hard sell to companies who do not control their own products. If the problem domain changes with each client, even proving that DSM works for an existing range of products will rarely be enough. Instead, before the company starts using DSM, it must be convinced that it can itself build a new DSM language and

generators for a sizable proportion of the new domains it may find itself in. This will in many cases be perfectly possible, but the belief in its possibility can often only be acquired by experience from a few successful applications of DSM. This cycle of chicken and egg will make DSM a nonstarter for this kind of company, unless they possess an unusually high breadth of experience or depth of understanding to see and understand how DSM works in both theory and practice.

CHAPTER 8

MOBILE PHONE APPLICATIONS USING A PYTHON FRAMEWORK

In this chapter we describe Domain-Specific Modeling (DSM) for modeling and generating enterprise applications for mobile phones. The selected target environment is Symbian smart phones based on the S60 mobile phone platform (Nokia, 2005). The DSM solution was made to work with a Python framework, but to demonstrate that models in DSM can be independent of the generation target, we show how native Symbian applications in C++ were later generated too. The modeling language and thus the application design models are the same, only the generator and platform code are different. You may consider the Python framework as a subdomain of Symbian applications (Babin, 2005) as it can be used to make typical administrative applications but not, for example, graphical games or device drivers.

8.1 INTRODUCTION AND OBJECTIVES

Before discussing DSM we will briefly describe the Python framework, Python for S60 (Nokia, 2004). This framework provides a set of Application Programming Interfaces (APIs) to access the platform services and expects a specific programming model for the user interface (UI). At this point, we need to emphasize that in the world of mobile application development, UI is understood to cover logic and the whole application functionality, not just the look and feel. The modeling language described

Domain-Specific Modeling: Enabling Full Code Generation, Steven Kelly and Juha-Pekka Tolvanen
Copyright © 2008 John Wiley & Sons, Inc.

here uses the widgets and services of the phone as its constructs, and follows the phone's UI programming model. The generator produces Python code calling the phone's platform services and executes the application in an emulator or in the actual phone device. Application execution requires that the Python for S60 package be installed, which includes the Python interpreter and S60 UI application framework adaptation connecting Python for S60 UI programming and accessing Symbian UI applications (such as phonebook, calendar, and camera). In this case, the phone platform and the Python framework were taken as given: there were no possibilities to modify them during DSM definition.

Target Environment and Platform The Python framework was developed to make application development easier when compared to traditional native S60 development based on C++. An easier entry is essential for opening mobile application development to a larger developer base, supporting possibilities for innovation, and lowering the barrier to entry for new developers. In addition to creating stand-alone S60 applications written in Python, Python for Series 60 also enables prototyping and concept development. The primary user target is Python programmers, but a separate development kit is also available for Perl (Hietaniemi, 2006) and a Mobile version of the Java Mobile Information Device Profile (MIDP) can be applied too. These SDKs thus target developers based on their preferred implementation language; with DSM, a single higher-level language can be used regardless of the underlying programming language, as discussed later in this chapter.

The Python for S60 release is offered as an extension to standard Python libraries (Rossum and Drake, 2005). It provides a subset of the underlying phone services and UI elements. For example, instead of over 10 different kinds of lists in native S60, the Python framework and its appuifw module provided only a few. In addition to supporting selected native Graphical User Interface (GUI) widgets from the S60 platform, the Python framework offers networking support for General Packet Radio Service (GPRS) and Bluetooth, Short Message Service (SMS) sending, and accessing other phone applications, such as camera or calendar. Python for S60 follows closely the S60 architecture and its UI programming model. The architecture of the Python for S60 environment is illustrated in Fig. 8.1 (Nokia, 2004).

The built-in appuifw module provides the key API. The Content_handler object type facilitates interfacing to other UI applications and common high-level UI components. It is based on the notion that UI application interaction can be reduced to operations on Multipurpose Internal Mail Extensions (MIME)-typed content by designated handlers. Via the Canvas object type, general-purpose graphics toolkits can attach to the UI of a Python application. During creation of the DSM solution, Canvas support was not yet available.

Before going into the case, a few words about the application domain: Mobile applications must be built to fit the relatively small size of the phone display and the layout of a S60 application. Figure 8.2 shows the user interface layout and its relation to the services available in the appuifw API. Reading the UI layout figure from the top, the main application window may be set up to be occupied by a UI control/widget. A

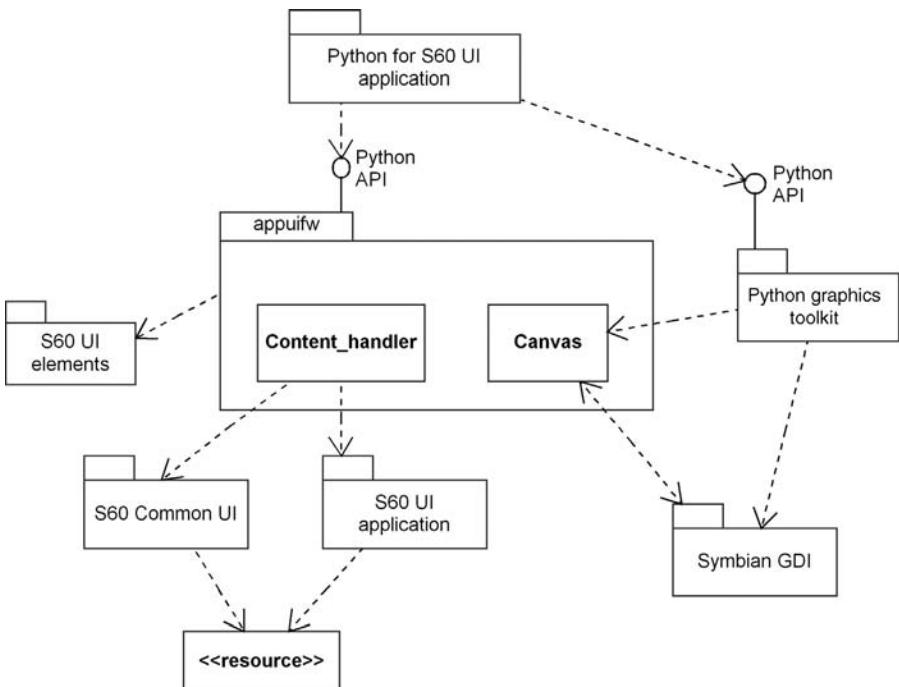


FIGURE 8.1 Architecture for application development with Python for S60

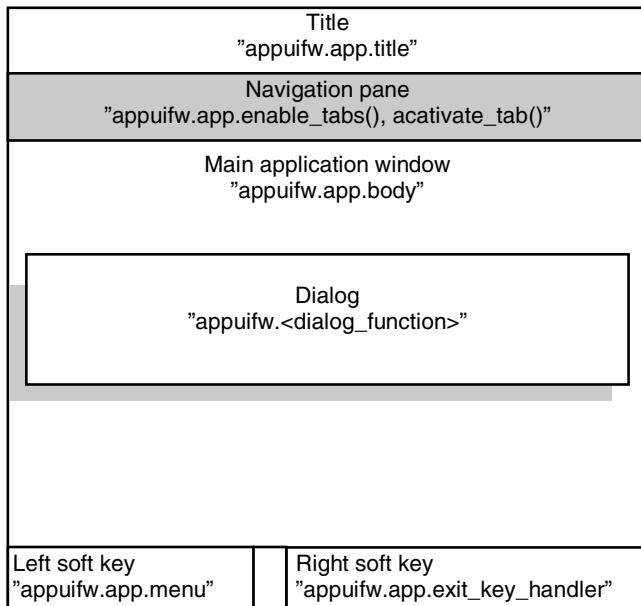


FIGURE 8.2 Python for S60 UI layout (Nokia, 2004)

multiview application can show its different views as tabs in the navigation pane and react as the user navigates between tabs—typically, by pressing the left or right selection buttons. Dialogs always take precedence over other UI controls and appear on top of them. The application uses two soft key buttons (left and right soft keys) to navigate in the application. Usually, the left soft key action in the dialogs is to accept, and the right soft key to cancel or navigate back.

DSM Objectives The objective of the DSM solution was to make application development possible for a fundamentally larger audience: people having little or no experience in programming Symbian/S60 applications. Following the original idea behind the Python framework of making application development easier, the DSM approach was seen to make it even easier while still guaranteeing that the framework is used correctly. The main idea was to get a situation where the developer could draw a picture of the application and then generate it to be executed in a target device. This quick and easy development approach was considered important for companies who wished to make some of their in-house applications mobile but did not have enough experience in mobile application development. The language would have the rules of the architecture and UI programming model so that a developer could focus on application functions and design, not on their implementation details.

8.2 DEVELOPMENT PROCESS

DSM definition started when the first public release of the Python framework was made available as a technology preview. During the DSM definition process, the framework's API became larger and about 15% of the APIs changed before the official release (version 1.0, Nokia, 2004). Since then new services have been added to the API but the basic structure and programming model have remained the same.

The definition process followed a prototyping approach, first making an example as a proof-of-concept and then gradually extending the language to support the whole Python framework. The first prototype was made by choosing a sample application to be modeled and generated. The sample application was the same as used in the documentation of the technology preview, so its design and implementation code was available. This application supported car pool users and used only two widgets, one asking the user to enter travel information and the other informing the user on the application state via a note dialog. On the application service side, the application used SMS to send text messages via the operator's network.

Development of the prototype took 1 day, during which the language and generator were directly defined into a metamodeling tool and the sample application was modeled and generated. At this stage, the modeling language had only three objects and a single relationship to connect them. The generator followed a simple flow navigation (similar to the Voice menu application in Chapter 7) to produce the application code. This code was structured similarly to the manually written application code made available as an example in the documentation of the Python for

S60 technology preview. The DSM implementation did not really cover application navigation or canceling—those were also lacking from the technology preview example.

The actual DSM implementation started once the framework became available in the first public release (version 1.0), in which about 70% of the framework APIs were the same as the version used to make the prototype DSM solution. The public release included several sample applications and those were taken as reference implementations, guiding the language and generator definition. Implementation of the DSM solution described in this chapter took about 10 man-days and was done by one person not involved earlier with S60 or Python. The implementation started from the metamodel of the prototype language and was done directly in a customizable metamodel-based tool. The review and testing of the DSM were supported by S60 and Python experts. Two experts were available to review the language definition and one expert tested the language within the created DSM tool. Testing with the tool involved using the language to model sample applications and running the generator to execute the applications. The availability of experts helped most in the beginning to get the basic structure of the DSM language defined.

Two review and testing phases were carried out during the process. The first testing was performed after having defined over half of the language, so its basic structure was clear. The second testing was done when the language was already defined, something you may consider a beta version. Most of the feedback and comments were related to code generation: finding a consensus among the four programmers involved on a good structure for the Python code to be generated. Finding good structures for the code was not just for the sake of DSM as that was needed anyway. Most testing work was done by the language definer by modeling about 10 small to midsize mobile applications using various constructs of the language. The language for Python for S60 was relatively easy to test as the domain could be examined by running the applications within the mobile phone, either in the real target device or in a PC-based phone emulator.

8.3 LANGUAGE FOR APPLICATION MODELING

The objective of the language was to make application development as easy and natural as possible. The approach to achieve this was by using modeling concepts directly based on the phone's services and UI widgets and by following the already used “cartoon-style” of UI application specifications. Thus, domain concepts such as “Sending text message,” “Note,” “Form,” “Pop-up,” “Browsing web” became candidate concepts for the language.

The application logic follows the flow of navigation, mainly using UI layout and connections between UI widgets. This cartoon-style UI specification, proceeding from one dialog to another, was also used in many companies developing mobile applications. In it, an application is specified using screenshots of relevant user interfaces along with a textual description of the stage and its relation to other stages.

In some cases, mock-up applications and “phones” made from cardboard are used, with application logic shown by swapping in different screens drawn on paper. Simply put, with DSM we aimed to replace these paper prototypes with a design that can be executed.

8.3.1 Modeling Concepts

The modeling language consisted of three main kinds of elements: dialogs, main UI controls, and phone services. We started with dialogs as they were the most stable and had clear characteristics. They were already working well in the first concept demo. After the rest of the dialogs were defined, the user navigation and application flow could be addressed with the language. The language concepts were named by the customer. Most of the names for language concepts came directly from the native S60 platform and from the Python for S60 concepts.

Basic Dialog Concepts The modeling concepts for dialogs included the following:

- Note: Displays a note dialog with text and a notification icon that indicates information, error, or confirmation. Later, Note was extended to allow writing custom code in the model element. In addition to writing plain code, reusing functions written in Python was also supported. For this purpose, the Note concept had an optional Function property type that referred to a Function in the Function library. The available functions could thus be imported as code files and referred to in models.
- Query: Displays a query dialog with a single field. In the language, this concept had a property Prompt to enter the query label and Type to choose what kind of value the query expects. This latter property was implemented as a list of fixed values to be chosen from: “text,” “code,” “number,” “date,” “time,” or “boolean”. The query dialog can include an optional initial value and the value entered is stored as an optional, but unique, return variable. If a return variable is not given, the value is treated as a local variable. This structure was also used in other concepts, allowing values to be defined that are used globally inside the program. All queries return None if the user cancels the dialog. Further details on navigation will be discussed later.
- Multiquery: A two-field text input dialog. Both queries show a prompt label in the same dialog and have separate return values.
- List: A dialog that allows the user to select a list item and returns the index of the chosen item or None if the selection is canceled by the user. List items are Unicode strings.
- Pop-up menu: A dialog list representing the menu contents with a label and a list of Unicode strings. Similar to the list, it returns the index of the chosen item and an optional return variable was included for the pop-up menu too. On cancellation, None is returned.

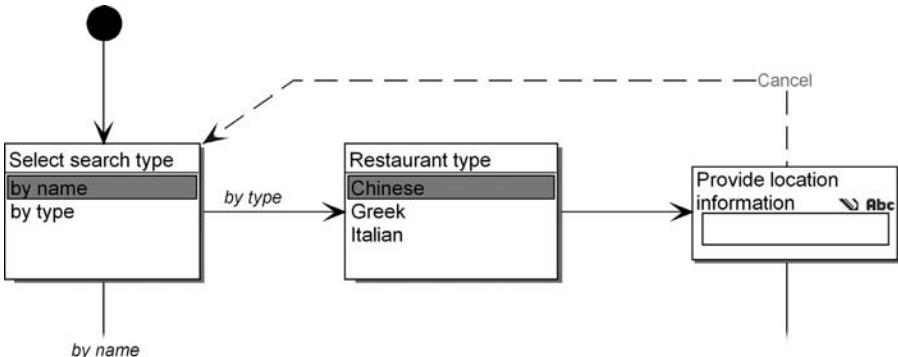


FIGURE 8.3 Sample model illustrating navigation between dialogs (two pop-up menus and a query)

Specifying Navigation with Relationships To describe application logic and navigation flow, dialogs—and later main controls and accessing phone services—could be connected with directed relationships. Figure 8.3 illustrates a sample model where dialogs are connected via directed relationships. Hence, there is no need to explicitly specify separate soft key buttons (two context sensitive buttons, see Fig. 8.2) as modeling properties for each dialog type. Having an extra place to define soft keys would just add extra modeling work, as navigation flow needs to be defined anyway. The directed relationship concept in the language could have a property type to describe the name of the soft key button, but as they are predefined in S60 and follow standardized UI conventions, their naming was not needed. Naming depends on the UI element, but typically, soft key buttons are named OK and Cancel. Symbian C++ development (Babin, 2005) sets different requirements for localization from the Python framework discussed here. In native C++ development, a typical approach is to add a localization entry as an identifier or generate it, if possible, for those concepts that are mapped to localization resources. During code generation, resource files can then be generated and localization data for different languages can be added.

Describing Choices with the Language Dialogs that provided multiple choices needed a Choice construct to enter the selection made for the specified connection. In Fig. 8.3, this situation is illustrated with the selection of the search type. Choosing to search by name opens a pop-up menu to choose the restaurant type. To be precise, the Choice property was added to the start role of the relationship to be close to the dialog element where the choice is made.

In the metamodel, the list items property and choice property used the same property type so that existing list values could be selected as choices, instead of writing them manually into the design models again. If no value was given for a flow, then that flow was the default for all list items without their own flow. This allowed

minimizing the modeling work. For example, in Fig. 8.3, it is enough to draw only one flow from the restaurant type selection to the query, instead of modeling all three possible choices from the pop-up menu separately.

Cancelling (pressing the right soft key) was also supported with a separate directed relationship type. Its use, however, was not defined as mandatory as in most design situations it is not necessary to specify. Normal cancelling behavior could be derived by following the application flow backward and only abnormal cases needed to be specified. This “convention not configuration” approach worked well here and greatly minimized the modeling work—in most cases, only forward navigation needed to be specified.

Relating Manual Code to Models Early on the customer raised the option of manually entering Python code in designs. Perhaps this was a way to allow a back door for manual programming, but the sample applications in the Python framework did not indicate any need for it. However, to satisfy the customer and show that the old approach can be applied as well, both default navigation and cancelling were extended with the possibility to add custom code. In the metamodel, a code property was added. A more sophisticated approach was to refer to functions available in a library. This was defined in the metamodel as a Function property similar to the Note object. The Function included a whole Function object with its own properties, such as name, parameters, code, and documentation properties. Python code written in relationships was used during code generation, and editing it in generated code was also made possible with protected regions. See later in this section for details.

Modeling Concepts to Access Phone Services Mobile applications usually need to access phone services. For this purpose, the modeling language was extended with additional concepts, including the following:

- **SendSMS** concept: A given string starting with a keyword, followed by message elements, is sent to a recipient number. To include variable values in the string a Message elements property was defined in the metamodel to include any of the Return values saved from dialogs or from the main controls. Here, the language integrates the use of a Return value as a Message element. If the name of a return variable changes in the application design, there is no need to manually propagate the change elsewhere in the models. A message element could also be a user-defined delimiter. Common delimiters, such as comma and dash, are provided for selection in a predefined list property. If a message element is taken from a complex data type (e.g., from a Form including several fields), the chosen value is referred to by its index.
- **Open**: For accessing files and browsing the web, an Open concept is used. This concept has a property, a target address, or a file name. In later versions of the Python framework, a stand-alone opening option was added to enable opening in a separate processes. For example, a browser could be started in a new

process outside the current Python application process. Because this stand-alone method of opening did not have the possibility of canceling the process inside Python, it was defined as a new language element instead of simply adding a Boolean property to differentiate the two access methods. This made it easy and clear for the metamodel to forbid adding a cancel flow to a stand-alone open object.

- The Start exe concept was defined to access native prebuilt phone applications, such as Calculator, Calendar, Camera, and so on, from the Python application. This modeling concept, Start exe, had a list property of available S60 phone applications. The first release of the Python framework did not provide access to a phonebook or making calls, but support for them could later be implemented by simply extending the current concept or defining subtypes for it (e.g., for making a phone call with the possibility of entering the phone number directly or accessing the phonebook first to choose the number). Entering the number to be called could already be supported with the Query concept, and then passing it to the phone call module could send the text message.

Specifying Domain Concepts with Internal Behavior In addition to dialogs and services, a Python for S60 application also usually contains some more complex user interface controls. The main UI controls are different from previously defined dialogs as they fill the whole display (see Fig. 8.2) and have a richer internal structure with their own state behavior. Their structure in the language definition is hence richer too. It is usually best to specify such complex internal behavior with a new language, as in the case of the watch in Chapter 9. Here, UI controls had rather limited internal structure that could be handled with menu connections and property types that describe behavior. Therefore, an additional language for the UI controls was not needed, although it could easily be added if needed, with subdiagrams for each UI control.

On the menu side, all UI controls could change the menus provided via the soft key buttons. The soft key buttons and menus for UI controls are managed by the underlying S60 platform, but the menu items available with the left soft key could be extended depending on the definition given for the UI control. Defining menus with the language was made possible via a specific flow, in which the flow end (a role called Menu item) allowed specifying a menu label and a target element. This target was usually a UI element or a phone service. The use of multiple relationships for each menu item did not allow specifying the order of the menu items. Ordering was not considered important as the Python framework itself did not provide full control of menu item ordering but only allowed adding new items. Item ordering could have been supported by adding order numbers for the flows or by using separate menu item elements, in other words, using a structure similar to the one for List.

Other internal behavior was dependent on each UI control and thus was captured as part of the respective type. The main controls include the following:

- Form: A dynamically configurable, editable multifield dialog. Form is characterized by fields, which can be either a simple input entry field or a Combo field. In the language definition, Field was defined as a collection of Form fields and Combo fields. A Form field element is defined by a label, a data type, and an initial value, and a Combo field is defined by a label and a collection of values. Because Form can be used either for editing the field values or just for viewing them, a property to choose between Edit and View modes was added to the Form element. This single choice allowed automating the Options menu content and the behavior of the Form without any additional definitions in the model: If the list is editable, the Options menu includes the save menu item and closing the form after edit asks whether the changes made should be saved. Boolean properties are also defined for Form to disable the automatic support allowing the application user to edit the labels in the form fields, and to specify a double-spaced layout (one field takes two lines as the label, and the value field is on different lines). To save the form, a return variable, similar to the dialogs, was added as a property of the Form concept.
- An editor UI control: A Text Editor modeling element with a title property, an initial text property to show predefined text when starting the editor, and a property for entering editing code. Similar to the other UI controls, a return variable is used to save the given text for later use.
- Listbox: Listbox in a mobile application shows a list of strings or a list of tuples of strings. The Listbox modeling concept has a property to enter a Listbox name that is visible on the screen and a content creation function and its return variable. The content creation function could be entered directly as text into the model or chosen from a list of available library functions. More of these options will be discussed next as well as in the sections on generator definition.

As mentioned, the main UI controls raised the need to specify some Python code related to models. Three options were provided:

- Code could be added directly to selected model elements. This was considered best for cases where just a few lines of code may be needed, like an optional save validation function, which checks the form content before it is saved.
- Code could be reused from the library. Here, a model refers to code available as a function, and the model only includes the parameters to be passed to the function. The modeling language was made to allow both black-box and white-box component libraries so that the modeler could also write function code. To support code reuse from a library, the modeling language was extended with a library concept added to the UI controls, navigation flows, and Note dialog. If this property has a value, it is considered to refer to a function. During code generation, the functions used are included in the generated code.
- Code could be added to the generated code. For this purpose, protected regions were defined in the generated code. Basically, all previously defined properties

were also marked as protected regions so that they could be changed after generation. This also required that empty blocks be generated in case the developer wanted to add code only after generation.

Other Notable Language Constructs Later, the language specification was extended by customer request to allow an optional naming policy. This was suggested to allow modelers to give human-readable function names that are visible in the generated Python code. The function name property was added to the modeling concepts and its use was made optional and left to the modeler's discretion; if not given in the model, unique function names are generated, minimizing the modeling work.

The specification of navigation flow required ways to specify application start and end. For this purpose, Start and Stop concepts similar to state machines were added to the language. As not all applications are necessarily closed during user navigation, the Stop element was extended to choose a policy for closing the application. Application exit was set as the default choice but the Stop element allowed choosing a wait policy. The wait policy is typical for applications that have several views and ending the navigation in one view should not close the whole application but allow for choices in other views or external processes that have been called, for example, via the Start exe concept.

All possible Python framework services, such as retrieving phone location information from the network, accessing drives of the phone, or copying files, were not included in the language as their own concepts. Their use could be supported with the previously described function calling capability: they could be defined as ready library services that are used in the designs with the Library concept.

Finally, a separate comment element was added to the language to allow attaching free form textual descriptions to models. This concept was extended with model checking capabilities: its representation in the design showed the results of a selected model checking report. Figure 8.4 shows a metamodel for the modeling language showing the legal connections between different object types of the language.

8.3.2 Modeling Rules

The basic metamodel included language constructs and their connections, but these did not cover the domain rules and the mobile UI programming model. Extending the language with rules would prevent errors and enable design consistency at modeling time. These were considered important as most application developers, similar to those using plain Python for S60, don't have much experience of mobile application development.

Rules Checked During Modeling The behavioral part of the application was largely modeled using directed relationships for specifying user navigation, view

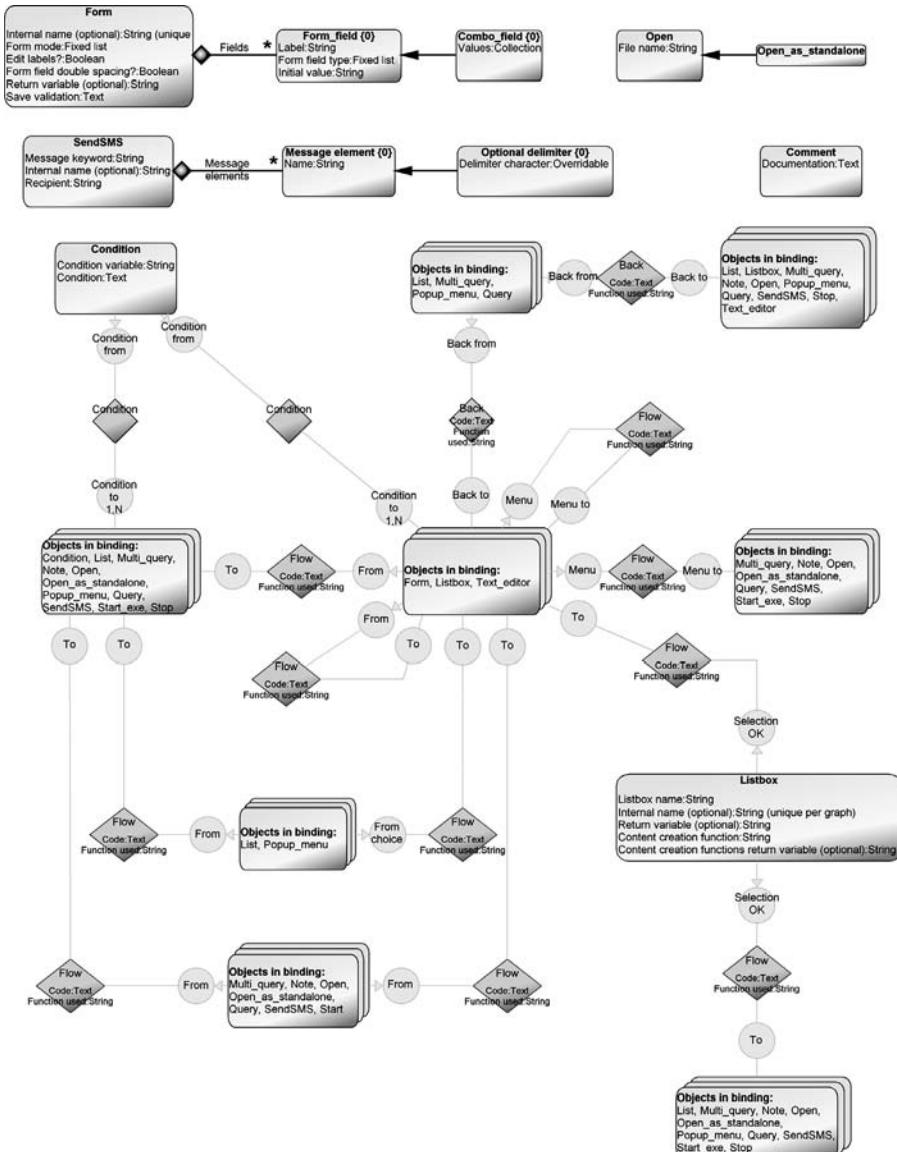


FIGURE 8.4 Metamodel of the language

structure, and access to the phone services. Here, general rules from state machines were reused and extended:

- Only one event can be triggered from the start state and canceling the start is not allowed. If canceling is defined in the application design (i.e., Cancel is pressed

in a dialog after a start stage) the generator produces an application exit code (see Section 8.5.2 for details).

- There can exist only one start object per application design, but if the application consists of multiple views, each view can have its own start object.
- The optional function name was declared unique to enable unique Python function naming during code generation.
- The return variable used in most modeling elements was made unique to support variable naming and to allow reuse of the entered values within the application.

Rules that dealt with the possible navigation actions, such as accept, opening a menu, and cancel, were defined on the connections between the modeling concepts. Some of the rules are as follows:

- For some language elements, normal navigation flow was defined to allow only one target element. For example, in S60, after showing a notifier for a couple of seconds, only one UI element or phone service can be triggered. Accordingly, the metamodel included a rule that allows only one flow from the Note element.
- The exception to the above navigation rule was allowing multiple targets for List and Pop-up dialogs and for accessing menus from UI controls.
- A particularly good choice for keeping models simple and minimizing the modeling work was to base the language on convention rather than configuration. For example, navigation by pressing the cancel button did not usually need to be specified at all.
- Specifying optional cancel navigation flow was restricted to a few UI elements only. List, Multi_query, Pop-up, and Query were defined to be possible sources for cancel (with the Back relationship, see metamodel in Fig. 8.4). Only one cancel per model instance was allowed.
- As the main UI controls (Form, Listbox, TextEditor) did not have a cancel option, their navigation was based on closing the control or choosing from its menu. The latter, closing the application from its menu, was specified in the language by allowing menu items to also include a Stop object.

Rules that dealt with accessing phone services include the following:

- Every phone service allowed triggering one UI element or other phone service.
- Creating a new thread with Open as stand-alone was modeled similarly to the others, but canceling it was not possible from the program calling the new thread. This was also the case with starting external applications outside Python—their dialogs and other widgets could not be handled or synchronized with the modeled Python for S60 application.

Guidance Rules Some of the domain rules were defined to be checked only by user choice. For this purpose, model checking reports were made, to be run when needed or alternatively to be made available during modeling by showing a comment element in the model. In other words, the comment element shows the result of running a model-checking generator that reports on possible errors and gives guidance if it detects that the model is incomplete. The latter was considered useful to guide developers in the beginning as it showed some of the tasks needed to complete parts of the design. The reports checked the following:

- If navigation flow was interrupted so that the application would stop before exiting.
- If choices entered in the flow from Condition, List, or Pop-up had more than one empty choice value or several identical values.
- If menu items started navigation flows that started more than one action. Longer chains were considered to lead to navigation paths that are too difficult to understand by application users. Placing this rule in the guidance report still allows one to make such designs (and thus applications also) so the rule was more of a suggestion than an absolute rule.

8.3.3 Modeling Notation

The notation was created in parallel with the metamodel. As the modeling concepts were largely taken directly from the UI elements, it was also natural to take the symbols from the actual phone UI. This created a powerful semantic mapping from the model to the running application. Figure 8.5 illustrates symbol definitions for the Form, Query, and SMS sending concepts. Form and Query are identical to their appearance in an executable application. Form includes its own compartment for the Options menu, from which users could draw connections to add menu items for the connected application elements. Query shows a prompt and an optional initial value as

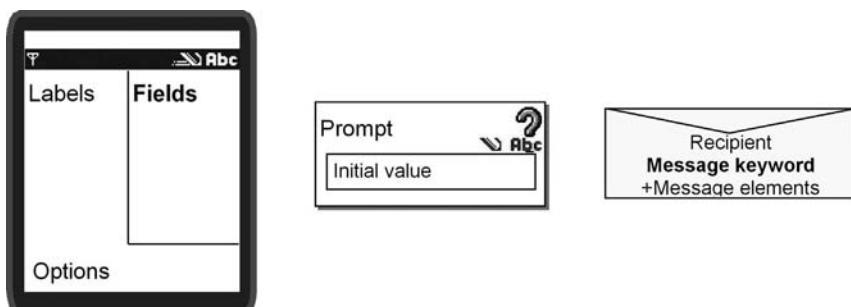


FIGURE 8.5 Defining the notational elements for language concepts Form, Query, and SendSMS

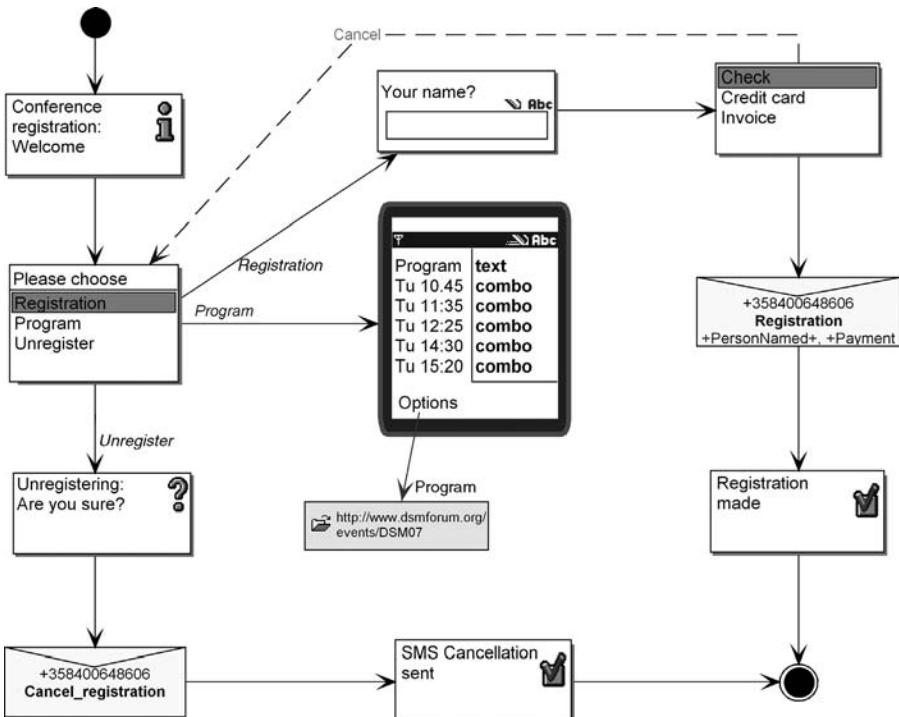


FIGURE 8.6 Design of a conference registration application using the DSM language for Symbian/S60 mobile phones

well as an icon for text entry, number entry, or Boolean entry, as specified by the Query type. Text message sending was shown by an envelope symbol showing the complete message content and recipient number.

Other modeling concepts were defined in a similar way and their appearance is illustrated in the example model (Fig. 8.6). Navigation was illustrated using directed arrows: normal navigation with a solid line with an arrow and canceling with a dotted line and “Cancel” text in the relationship. Menu items were also specified via relationships, but those were connected to the bottom left port compartment in the main UI control. For example, Fig. 8.5 shows the Options menu port compartment that allowed specifying new menu items.

8.4 MODELING PHONE APPLICATIONS

The modeling language was tested immediately after adding to or changing the metamodel. This ensured first-hand experience of using the language.

8.4.1 Example Models

The DSM language is illustrated in Fig. 8.6 with a sample application design. If you are familiar with phone applications, like a phone book or calendar, you will probably understand what this application does. A user can register for a conference using text messages, choose a payment method, view the program and speaker data, browse the conference program on the web, or cancel the registration. As can be seen from the model, all the implementation concepts are hidden (and need not even be known by the modeler).

8.4.2 Use Scenarios

The basic use scenario for building an application approaches the ideal of drawing UI elements and phone services on a canvas, connecting them and then running the generator to produce the application and run it in the target device or an emulator. Figure 8.7 shows a sample application design.

As the application size was considerably smaller than in native C++ development, localization, reuse, and support for product lines were not considered relevant. The application size was usually just one diagram, or possibly many small ones for a multiview application. Applications were usually less than 300 lines of Python code. The application modeled in Fig. 8.6 is 145 lines. Reuse is not considered in the language other than allowing sharing the same function library. At the model level, the

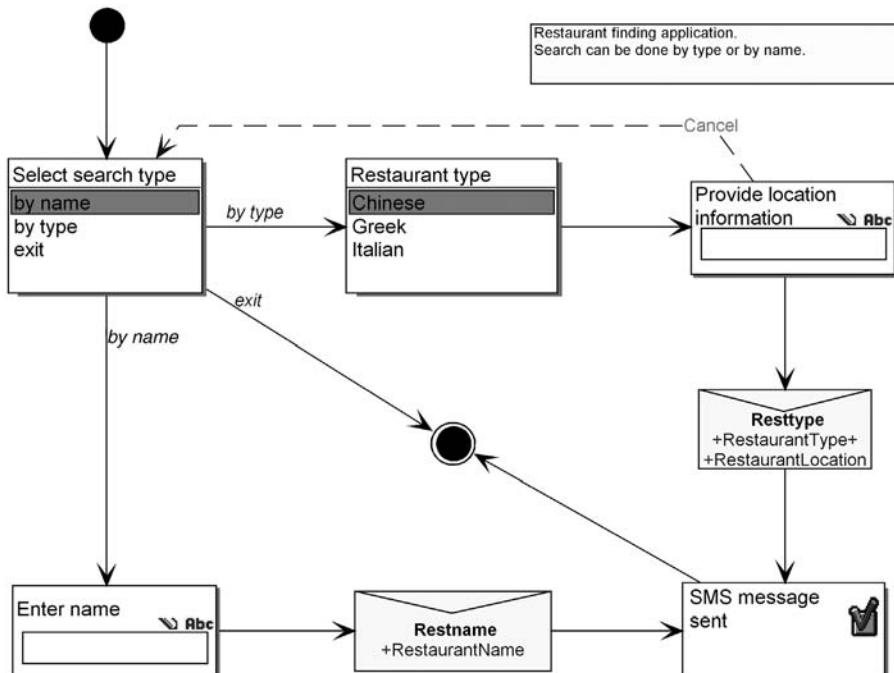


FIGURE 8.7 Sample application design

modeling tool used does however allow one to copy or share the same design elements between different applications, though. Typically, one developer makes the whole application and there is no need to support multiple views of the design or support multiple simultaneous modelers.

8.5 GENERATOR FOR PYTHON

After some example designs, code generation was targeted. The prototype DSM language, built as a proof-of-concept, applied a simple tree structure where the generator navigated from the first dialog to the last one and produced code for them that called the Python framework. Although this approach allowed generating applications similar to the example applications provided with the Python framework, it was very limited. It expected designs and application navigation structure to form a tree and did not allow for variable handling or differentiate local and global variables. As the manually programmed sample applications provided with the Python for S60 package used function definitions, calling them via a dispatcher, the customer thought this would be powerful enough for code produced by a code generator too. The other alternative, use of state machine based approaches (as in Chapter 9), was considered over kill.

From the designs expressed with DSM, the generator produces code that can be executed both in an emulator for testing purposes and in the target device. Next, we describe the structure of the code generator and then show samples of the generated code.

8.5.1 Generator Structure

Following the chosen function-based approach, each UI control, dialog, and phone service was implemented as a corresponding Python function definition. The main UI controls, like Form, also had internal functions. This approach also influenced the generator structure: each modeling concept mapped to one or more generator modules; one generator module takes care of List, another SMS, and so on. The general structure of the generator is illustrated in Fig. 8.8. An autobuild generator produces the Python script and starts an emulator to execute the application.

The generator first produces the imports necessary for the application (like appuifw for UI elements or messaging for sending text messages) and defines the functions used from the library. Then the main part of the generated code is taken from dialogs, UI controls, and phone services described in the design models. These generator modules, identifiable by their language concept name, are collected together in a gray rectangle in Fig. 8.8.

As many of these modeling constructs have some shared behavior, such as naming and navigation, common parts were implemented in shared generator modules. Generator modules “_Internal name” and “_Return variable name” are widely used during Autobuild and thus their use in the call order is not shown in Fig. 8.8 to make the figure more readable. When it comes to the common code, one generator module (Python script) created a part common to all Python for S60 applications: the

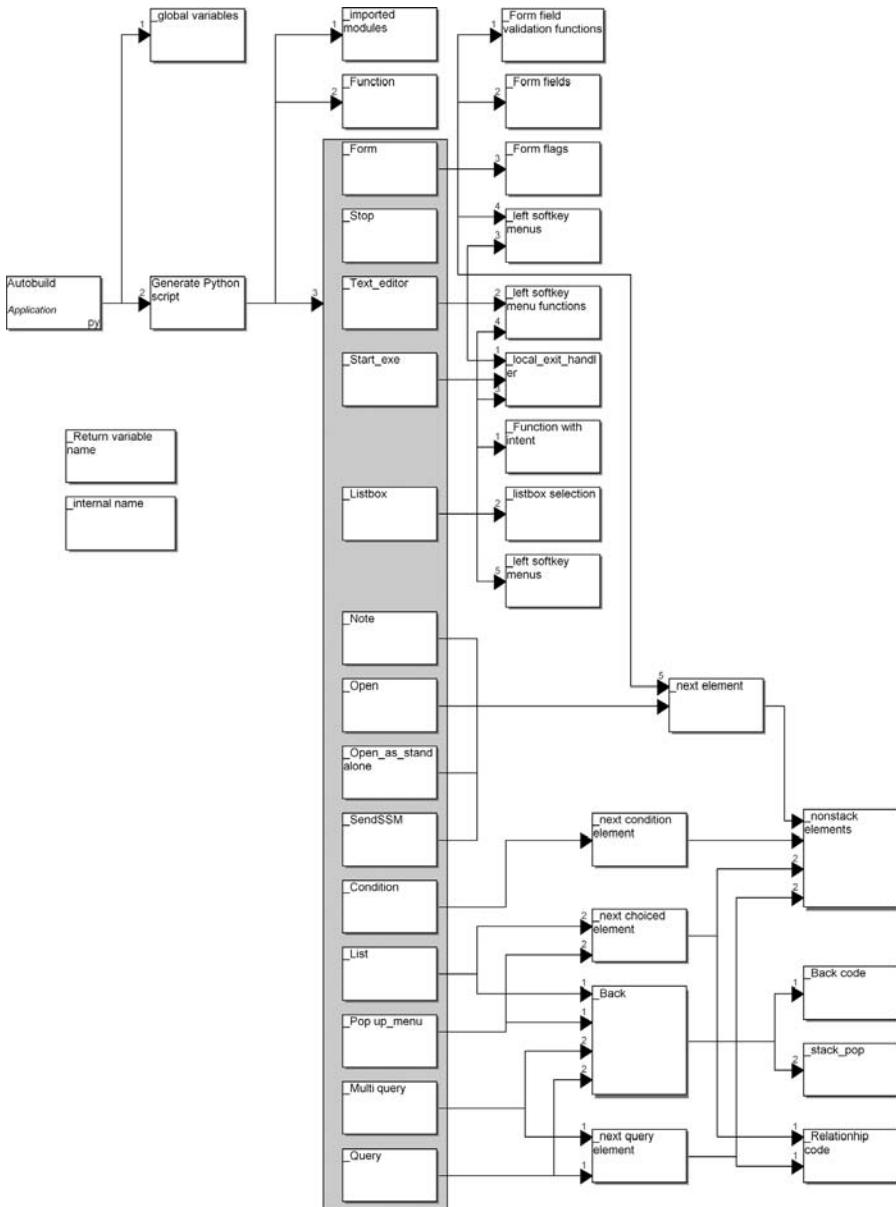


FIGURE 8.8 Structure of the Python for S60 generator

navigation stack handling and the main dispatcher to start the application. This code was nearly identical for every application and thus that generator module did not use much data from the design models. The results of that module are shown in the sample application code in the next section: see lines 128–145 in Listing 8.5.

A Basic Generator Module for One Dialog Concept Let's look in detail at the generator definition for one of the dialog types: pop-up. It opens a dialog with a prompt label and list of choices. Based on the choice made, the next function is called. The generator definition is described in Listing 8.1. It is defined in the generator definition language used by the MetaEdit+ tool. Underlining in the listing refers to the language concepts: It shows which parts of the code are taken from models and thus what can vary between the applications.

Lines 1 and 17 are simply the structure for a generator and line 2 is a comment for the generator module. Line 3 creates the function definition signature and line 4 is a comment for the Python code. Function naming in line 3 calls another generator module that creates a unique function name if the developer has not given a name for the pop-up menu in the application design model.

Listing 8.1 Code generator for pop-up menu.

```

01 Report '_Popup_menu'
02 /* Produces popup menu-style dialog code */
03 'def ' subreport '_Internal name' run '():' newline
04 '# Popup menu ' :Label newline
05 if :Return variable (optional); then
06   ' global ' subreport '_Return variable name' run newline
07 endif
08 ' choices' oid ' = ['
09 dowhile :List { 'u"' id '", ' }
10 '"' newline
11 ' index = appuifw.popup_menu(choices' oid ', u"' :Label '')'
12 newline
13 ' subreport '_Return variable name' run '= choices' oid
14 '[index]' newline
15 if index == None:' newline
16 subreport '_Back' run
17 subreport '_next choiced element' run
18 endreport

```

Lines 5–7 declare the return variable, if this pop-up has one. Lines 8–10 create a temporary variable that includes the choices given for the list in a Unicode string. Line 11 produces the actual pop-up call using the service from the Python framework. It gives as parameters the choice elements defined for the list and label of the pop-up. If no label is given, an empty Unicode string is generated. Lines 12–13 save the selection made in the pop-up into a global variable (done in a subreport “_Return variable name”). If no return variable name is given, then the choice value given by the application user will not be used other than for choosing the navigation path.

Reusable Generator Modules: Navigation As several concepts require similar code to be generated, such as navigation flow and function naming, parts of the generator definitions are made into modules called by other generator modules. For example, the generator module “_Back” is used by several dialogs to generate

cancelling code. The generator module “_next choiced element” in line 16 is also used by List to generate transition code based on the choices made in the list dialog.

Part of this generator module is illustrated in Listing 8.2 below. To allow drawing models where choices don’t need to be explicitly specified but can be left undefined, the models are read twice: first, for cases where all list values are specified (like in the pop-up for choosing an action in Fig. 8.6) and then for cases when a choice does not guide navigation but is used for getting user input for later use (like the list for selecting a payment method in Fig. 8.6).

Listing 8.2 Code generator for _next choiced element.

```

01 Report '_next choiced element'
02 /* Reports next choiced element for lists and popups */
03 do ~From choice {
04     /* If there is specified value in Choice property */
05     if :Choice <>'' then
06         '    elif choices' oid;1 '[index] == u"' :Choice '"':
07             newline
08 ...
09     /* If Choice property is not specified */
10     if :Choice   ='' then
11         do >Flow~To.() {
12             '    else:' newline
13             '        return (' subreport '_Internal name' run',
14             subreport '_non-stack elements' run')' newline
15         }
16     endif
17 }
18 endreport

```

Line 3 follows the navigation flow from Pop-up (or List) based on the choice roles leaving it. If a Choice value is specified on the navigation arrow, the generator produces code for the selected choice (lines 5 and 6). Line 23 analyzes only cases where no choice is specified on the arrow: it creates a single choice with the Python else construct. It should be noted that multiple undefined choices or several identical choices are already checked from models: an application developer is informed about Pop-up and List elements, that have several choices with empty or identical values. Although such checking could be performed at code generation time, it was considered much better to inform the developer as early as possible about errors and conflicts.

Generator Module for Accessing Phone Services On the phone service side, Listing 8.3 describes a code generator definition for producing code for sending text messages. After creating a function definition and a comment (lines 3–4), message elements sent (which can also include delimiters) are declared as globals, so they can be referred to from inside the text message sending function (lines 5–9). As the metamodel is made so that message elements are actually return variables given in

design models, this allows access to data given elsewhere in the application. Lines 10–19 create the message content as a string that is then used as a parameter when calling the Python for S60 API in line 21. Finally, line 22 calls a commonly used generator module to create a navigation call to the next function.

Listing 8.3 Code generator for sending text messages.

```

01 Report '_SendSMS'
02 /* Produce SMS sending code */
03 'def ' subreport '_Internal name' run '()' newline
04 '# Sending SMS ' :Message keyword ; newline
05 do :Message elements {
06     if type ='Message element' then
07         '    global ' id newline
08     endif
09 }
10 '    string = u''' :Message keyword ; ''''
11 dowhile :Message elements {
12     if type ='Optional delimiter' then
13         '+' id ' '
14     endif
15     if type ='Message element' then
16         '\' newline '           +unicode(str(' id'))'
17     endif
18 }
19 }
20 newline
21 '    messaging.sms_send("' :Recipient '", string)' newline
22 subreport '_next element' run
23 endreport

```

As can be seen for the generator definition, Python also sets layout requirements for the code generation definition: indentation is used to express nesting. This Python feature made generator definition a little more tedious but also required creating near-duplicate generator modules. For example, the only generator difference between a normal function definition and the Listbox content creation function is that the latter is indented to be part of the Listbox function.

Different Generator Usages This basic generator produced plain code to be immediately viewed. For alternative code generation tasks, three additional generators were defined. They all called the basic generator for the actual code generation and just guided the use of the generation result for alternative needs: One generator saved the application into a file using the application name as the file name. A second generator uploaded the application into the emulator to be executed via the Python interpreter. The third generator added the application to the main grid of the phone (like a desktop), similar to other main phone applications. This generator also included information about the application number and its place in the grid in the generated code. It would also be possible to have a generator that

creates a stand-alone application installer for sharing the applications, by calling the build tool for application packaging.

8.5.2 Generator in Action

Application designs were executed in an emulator and in a real target device by running the generator. This was part of testing the DSM solution. In addition to executing the generated applications, model checking reports were also tested to show missing or incomplete parts in the design. Listings 8.4 and 8.5 illustrate code generated from the model shown in Fig. 8.6. As specified by the generator, code is structured into functions, one for each dialog and service. In the listings, underlining is used to mark those parts of the code that are taken from models. To make code more readable, the listing does not show protected regions. After all, they were not needed in this sample application.

Listing 8.4 Python code generated from Fig. 8.6.

```

01 import appuifw
02 import messaging
03
04 # This application provides conference registration by SMS.
05 #Globals
06 Setdata=""
07 Payment=""
08 PersonNamed=""
09
10 exit_flag = False
11 call_stack = []
12 lock = e32.Ao_lock()
...
14 def List3_5396():
15     # List Check Credit card Invoice
16     global Payment
17     choices3_5396 = [u"Check", u"Credit card", u"Invoice"]
18     index = appuifw.selection_list(choices3_5396)
19     if index <> None:
20         Payment = choices3_5396[index]
21     if index == None:
22         return (Choice_list, True)
23     else:
24         return (SendSMS3_677, True)
25
26 def Note3_2543():
27     appuifw.note(u"Conference registration: Welcome", 'info')
28     return (Choice_list, False)
...
65 def Open3_1803():
```

```

66 # Opens the specified file
67 appuifw.app.content_handler = appuifw.Content_handler
68 (lock.signal)
69 appuifw.app.content_handler.open") (u"
70 http://www.dsmforum.org/events/DSM06")
71 lock.wait()
72
73 def Choice_list():
74 # Popup menu Please choose
75 choices3_2520 = [u"Registration", u"Program", u"Unregister"]
76 index = appuifw.popup_menu(choices3_2520, u"Please choose")
77 if index == None:
78     return ((call_stack.pop(), False)
79 elif choices3_2520[index] == u"Program":
80     return (SetForm, True)
81 elif choices3_2520[index] == u"Registration":
82     return (Query3_1481, True)
83 elif choices3_2520[index] == u"Unregister":
84     return (Query3_6300, True)

```

The generator first produces module import commands (lines 1–2) based on the services used, like the messaging module that provides SMS sending services. This is followed by documentation specified in the design. Next, each service and dialog is defined as a function. Lines 41–51 describe the code for the payment method selection that uses a List dialog. After defining the function name and comment, a variable named Payment is declared global to be available for the other functions in the application. Line 44 shows the list values as Unicode in a local variable, and line 45 calls the List dialog provided by the Python for S60 API.

Lines 53–55 create the function code for the welcome note. Lines 65–69 show the code for browsing the web. Lines 71–82 show the pop-up menu code produced by the generator defined in Listing 8.1.

Listing 8.5 shows a text message function, application exit, and common framework code for handling navigation stack and application start. Lines 111–120 are created based on the generator definition shown in Listing 8.3. Lines 113–115 define the global variables so that the text message function can access data entered in the design. Line 119 calls the imported SMS module and its sms_send function. Parameters to the function, recipient number, message keyword, and content, are taken from the model and the generator takes care of forming the right message syntax. After all, it is always the same and is now defined just once in the generator.

The rest of the code is almost the same for all applications. Application stop (lines 122–126) is taken from the Stop state using the unique function name given by the generator as entering separate names in models for the Stop or Start states would just add extra modeling work.

Listing 8.5 Python code generated from Fig. 8.6.

```

111 def SendSMS3_677():
112     # Sending SMS Registration
113     # Use of global variables
114     global PersonNamed
115     global Payment
116     string = u"Registration " \
117             +unicode(str(PersonNamed))+",  "\ \
118             +unicode(str(Payment))
119     messaging.sms_send("+358400648606", string)
120     return (Note3_2227, False)
121
122 def Stop3_983():
123     # Application stops here
124     global exit_flag
125     exit_flag = True
126     return (lambda: None, False)
127
128 def main():
129     old_title = appuifw.app.title
130     appuifw.app.title = (u"Conference registration")
131     call_stack.append(Stop3_983)
132     state = Note3_2543
133     try:
134         while not exit_flag:
135             new_state, add_to_stack=state()
136             if add_to_stack:
137                 call_stack.append(state)
138                 state = new_state
139     finally:
140         lock.signal()
141         appuifw.app.title=old_title
142         appuifw.app.exit_key_handler = None
143
144 if __name__ == "__main__":
145     main()

```

Lines 129–130 save the old title and change the application name. In line 141, the old title is returned when exiting the application. The new application name is taken by the generator from the name of the design model. If alternative model naming should later become necessary, the metamodel could be changed to include a specific property for the application title.

The framework code takes care of the navigation stack. Line 11 in Listing 8.4 defines Call_stack as a list. To ensure that there is at least one item in the list (if the user presses cancel immediately after starting the application) a Stop object is added to the stack in line 131. The first state (Note3_2543 in the listing) is given and the application

is started with the main dispatcher with the “try” line (133). The loop continues until `exit_flag` is True; the `Stop` function sets the flag (line 125). In the loop, each function executed inside Python will always return a tuple with information about the next function and whether this current function needs to be stored on the stack. For example, `Note` (line 55) and `SendSMS` (line 120) have the parameter value `False` and they are not stored on the stack. In most of the other functions, `add_to_stack` is set to `True`, and a new state will be added to `call_stack`.

When `exit_flag` is True, `lock.signal` is included to ensure that the application is “awake” and closing goes smoothly. Finally, the old application title is returned and the application’s `exit_key` handler is reset (there can be customized handlers during application execution). The last two lines show the standard way to start a Python application.

8.6 FRAMEWORK SUPPORT

During DSM creation, the S60 mobile phone platform and the Python for S60 framework were taken as given: The language creator simply made the generator call the services provided by the frameworks. An additional framework to support DSM was made to handle application start, exit, and navigation.

In addition to basic start and exit code, this domain framework included a dispatcher that started the application by calling the first function and looping until the application exit was reached. This code was generated as part of the main function of the application code, as already shown in detail with the application code in Listing 8.5. A bigger part of the framework code was made to handle the navigation stack, which enables getting back to the previously accessed function without starting a new one. Here, basic stack handling operations were designed so that application objects (functions in generated code) that could be targets for cancel navigation are pushed onto the stack after accessing the next element in the flow. Correspondingly, they are popped from the stack when they are the target of a cancel. Not all concepts in the navigation flow were put on the stack as they were not suitable for canceling. For example, a canceling operation that would trigger an already closed stand-alone application outside the Python application scope would not make sense.

The last piece of the framework code was made to take care of concurrency control in the UI application code. Python for S60 requires that execution paths be blocked when certain phone services are used: the concept of an active object in the underlying Symbian OS needs to be recognized. Otherwise, the execution control inside the Python for S60 application could not be reached when control is given to the underlying phone service. The Python adaptation to Symbian automatically handled part of this thread management, but depending on the application logic, the Python programmer also needed to take care of concurrency by using the API commands available from the library. In the DSM solution, setting wait locks and releasing them in the correct manner was handled with the framework code. The code calling the concurrency library was put into the generator so the modeler did not need to think about it.

The connection between the framework code and the generated code could be established in many different ways. The framework code could be implemented in a

library and called during generation. Further, the library could be part of the function library available in the modeling tool or saved to external files. To make generation safe and easy, the framework code was put into the generator. This way, the user would not be responsible for finding the correct function or need to take care that the external files containing the framework code were available for the generator.

8.7 MAIN RESULTS

The DSM language and generator were created to make development fundamentally easier and faster. Developers were not expected to master the details of the S60 architecture and coding: that unnecessary complexity was hidden with a language that fit better to the development tasks. We believe that the above few examples illustrate how the objective was achieved. Perhaps an even better indication of this is that, using the language, you too could now make mobile applications! See the book web site for instructions on using DSM solution.

Making application development easier was valued highly among mobile developers, especially those who had been using traditional manual coding approaches. The complexity of the underlying platform and mobile programming model served to further underline the benefits. A good indication of this is a statement made by a mobile application developer, Simo Salminen from Flander Ltd: “When you’re used to Symbian C++, it’s quite a shock to notice how easy UI application building can be.” This statement reveals the clear difference between programming on top of a platform and using DSM on top of a platform.

This increase in the level of abstraction away from programming constructs, and the use of code generators also lead to improved productivity. Although this improvement was never explicitly measured for large applications, it was evident. Markus Hänninen from Enpocket saw that the DSM solution “greatly speeds up the development process and communication with end users.” This communication is usually found to even work better later when models expressed with higher level constructs become useful to people outside the development team too.

8.8 EXTENDING THE SOLUTION TO NATIVE S60 C++

As we mentioned earlier, S60 applications are normally made in C++, as are the native platform libraries that applications call. The C++ Software Development Kit (SDK) is widely regarded as being something of a nightmare to use, with baroque libraries and a number of nonstandard conventions. On the earlier Psion EPOC platform, these conventions were able to offer more efficient memory, power, and even CPU usage, but as the platform has grown, the resulting phones seem to have been unable to maintain these advantages. The result is a platform that demands a lot from the programmer: there are so many hoops to jump through and choices to make that writing concise, elegant code is all but impossible. In S60’s defense, most of today’s mobile phones do not allow third parties to write native applications at all. The growth of the platform can also be explained by the explosion in the kinds of functionality it

must support. The original EPOC allowed simple personal productivity and network applications, but now there are also all the phone and camera functions with their underlying protocol stacks and drivers.

8.8.1 Extending the Domain

The Python support only exposes a small fraction of the native libraries, focused on the dialog-based UI. While this is sufficient for an interesting subset of applications, it leaves out the richer tabbed views and menus that are used in many even slightly larger applications. The Python DSM solution also lacks the ability to support localization and build entirely new first-class applications that are run independently of other applications (like the Python interpreter). When extending the DSM solution to generate C++, we decided to extend the domain it covered to include these concepts of view, menu, and application. The original diagrams would describe what happened in a single view, and a new top-level diagram would specify the application and its views and menus.

Figure 8.9 shows an example of a top-level diagram based on an S60 SDK sample application for demonstrating menus. The top row contains a simple Application object, ConfApp, that specifies the name and unique ID of the application. The second row specifies the three tabbed views of the application, with the initial default view starred. Each view can explode to a diagram specifying the dialogs used in that view, e.g., here, the first view from the sample has been changed to point to the conference registration diagram defined earlier. The third row contains the menus used in the views. In S60, the menus are specified as if they were in a menu bar containing top-level items like System, App, View, and Context. Current S60 phones, however, only display one simple menu, formed by concatenating the menus for these top-level items. A menu item may point to a custom command (not yet implemented in this

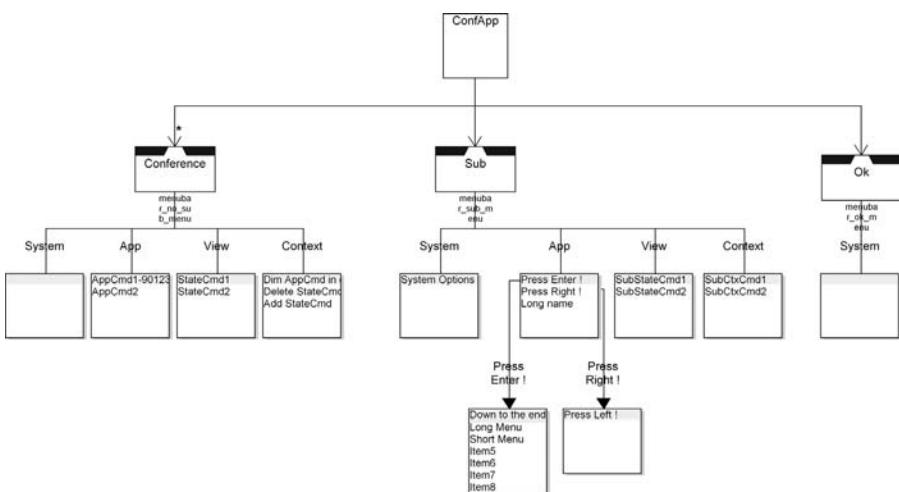


FIGURE 8.9 S60 application, views, and menus

DSM solution), a system command like Copy or Quit, or a submenu. Two example submenus are shown in the fourth row.

8.8.2 C++ Generators

Even a simple S60 application like Fig. 8.9 requires the developer to create a large number of files—27 in this case. The architecture of these files seems to tend toward high coupling and low cohesion, effectively preventing reuse of any of the files in a subsequent application. The sheer number of files makes building a generator somewhat laborious, but only in the same way as building a single application by hand would be. The verbosity and repetition between the files do, however, mean that a small amount of modeling effort will result in a prodigious amount of output in terms of code.

The structure of the C++ generators is shown in Fig. 8.10. The generator begins on the left by creating the five directories into which the files must be created, and an Autobuild batch file that will kick-start the S60 build process and emulator. We will look at the main body of the generation from the top of the figure down, following the grouping into numbered gray areas. Group 1 handles the generation of the static user interface description from the top-level diagram: the views, menus, and reusable strings. Its `_RSS` generator produces the single largest file, `AknConfApp.rss`, weighing in at an impressive 16 KB. Separate `_RSS_AIF` and `_RSS_caption` generators add some smaller RSS files for the application as a whole. Group 2 contains some small generators that descend to the subdiagrams of the views, adding to the main RSS file the strings used for choices in Pop-up Menus and Queries.

Group 3 creates the HRH file, which contains an enum of all the localizable strings used in the application, and the LOC file, which defines the English language version of each string. In the models, each string used is represented as both a unique name and the English language string, the latter being displayed in the diagrams for ease of reading. The MMP, PKG, and INF files and corresponding generators produce the application descriptor and make files.

Bizarrely, the same variable or function name must be formatted in different ways in different places: lowercase in RSS files and uppercase in some places in C++ files. There are also conventions for prefixes for names in various places. These requirements are handled in the short utility generators in group 4, which are called by many of the other generators.

So far, none of these files has actually been real C++ code: some had their own S60-specific syntaxes, some were simple C++ definitions. The C++ code makes up two thirds of the total size and number of files and is created by groups 5, 6, and 7. In S60, an application is represented by three C++ files: an application, UI, and document file, and these are created by group 5. Each view has its own file and an ancillary container file, and these are created by group 6. The dialog contents of each view are created as functions in its file by group 7, which has a subgenerator for each dialog type.

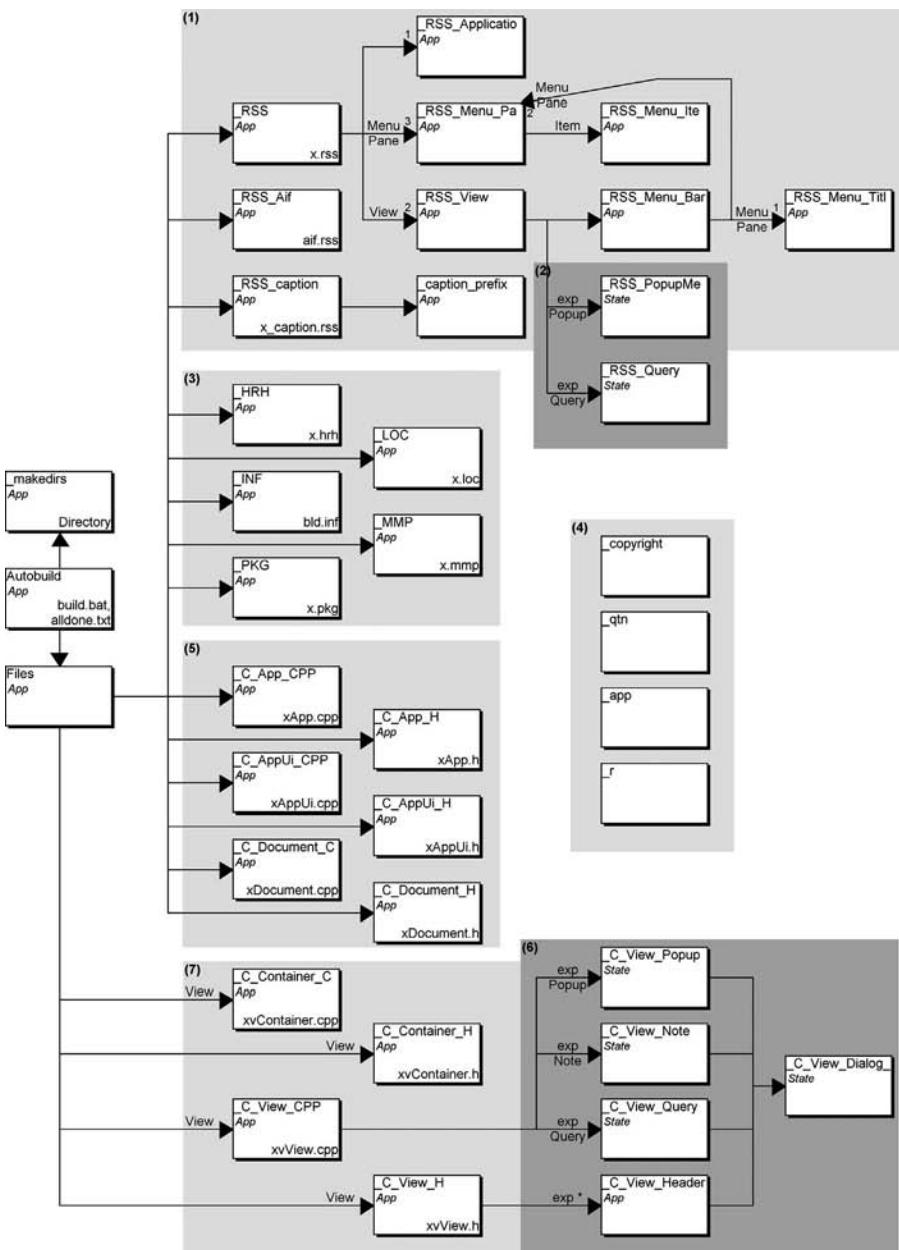


FIGURE 8.10 Structure of S60 C++ generators

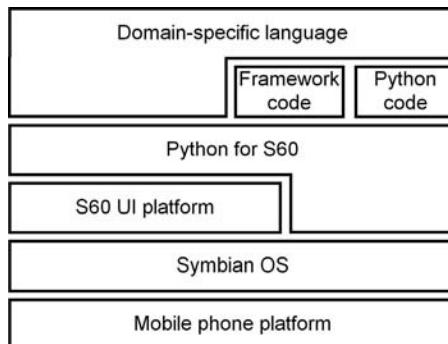


FIGURE 8.11 Domain-specific language on top of platforms

8.9 SUMMARY

The DSM case described here shows how the level of abstraction can be raised on top of an existing platform—or rather, on top of several layers, as illustrated in Fig. 8.11. First, there is a mobile platform providing some hardware-related software. On top of this, an operating systems provides a set of libraries and basic mobile services and applications. Next, to make application development faster and easier, the level of abstraction is raised by providing a suite of libraries and standard applications with a UI application platform. In our case, the UI platform has been S60 (Babin, 2005). On top of this UI platform, the most typical application domain, enterprise, and administrative applications are selected, and a framework is provided based on Python. The DSM solution is made to further hide application development complexity to make it easier and faster.

We have focused here mostly on creating a DSM solution for Python for S60. To make applications, the solution provides a DSM language backed by framework code to support the generated code. The modeling language is based on UI concepts and follows the mobile phone programming model with the use of dialogs, UI controls, and phone services. Each of these concepts is used during code generation to apply the services from the Python for S60 framework. A few language concepts were also used to add manual code that does not relate directly to the Python framework but allows Python programming. The need for manual programming was reduced to a few places where higher level constructs than Python were not found.

The case also shows how DSM can evolve when the underlying platform evolves. During the case, Python for S60 had three releases, yet no changes were needed to the models describing the applications. The changes made by Nokia to the Python for S60 framework—mostly new services—led to changes in only the language and generator. For example, when the text message sending module was restructured and its API changed, only the code generator needed to be changed. Old applications using the text messaging could be updated to the new version simply by generating the code again. This is in contrast to manually changing all hand-coded applications that used text messaging.

The DSM solution can also evolve in the future. After having made enough similar kinds of applications, perhaps a review of manually written Python code included in models will detect similarities that can be abstracted to the modeling language. Also, if Python applications are built in other areas where S60 UI styles can't be used, like games, then the possibility of adding more Python code can be provided within the DSM solution. Most likely it would be better to provide libraries for graphics and extend the DSM solution to cover them too. Then those newer application areas could have the same benefits that the current DSM solution provides.

CHAPTER 9

DIGITAL WRISTWATCH

This example differs from the others in that the intention was not to build a commercially viable family of products. Instead, the initial purpose was to provide a fully worked example of a Domain-Specific Modeling (DSM) language, generator, and models to demonstrate the principles of DSM, as part of our MetaEdit+ evaluation package. The actual process of creating the modeling language was, however, made as natural as possible, following the practices that had become established in real projects. The artificial background of the example is perhaps most clearly seen in its limited scale. Conversely, this small size is its strength as a pedagogical tool: small enough to be understood in a relatively short time but large enough to provide realistic insights into every aspect of DSM.

9.1 INTRODUCTION AND OBJECTIVES

The background for the digital wristwatch language is thus a fictitious manufacturer of digital watches, circa 1985—we shall call them Secio. Secio has noticed that producing the software for each watch by hand is becoming a significant bottleneck, as consumers demand functionality beyond simple setting and display of the time. It has also been realized that different consumers want different functionality and have different requirements for ease of use versus extensive functionality, physical compactness versus amount of information displayed, and so on.

Therefore, Secio has decided to build its range of watches for next year as a product family. There will be different watches for different consumer types and price points, but the watches will be able to share common parts of the software. Such common parts will include basic framework such as the ability to show numbers on an LCD display, as well as individual applications such as a stopwatch or a countdown timer. The basic framework components will be present in all watches, and either already exist or will be coded by hand. All the individual applications will not be present in all watches, yet it is hoped that later addition of an existing application to an existing watch could be a simple operation.

For a variety of reasons—some technical, some political—Secio has decided to try to create a DSM language for building the watch applications. The main objectives are to reduce the development time and complexity of building watch applications. In particular, the current watch software is one large piece of code with little separation of concerns. Being able to separate out different parts of the code would improve reuse possibilities, and also allow the developer to concentrate on one area at a time, thereby reducing errors. The separation of behavior into Model (operations on time values), View (display of the time and icons), and Controller (user input on the watch buttons) will form one weapon to divide and conquer the mass of code. The main weapons though will be the higher level of abstraction and the close fit of the modeling language with the problem domain. Developers will be able to concentrate on the application behavior as perceived by the end user, rather than on the lower-level details of how to implement that behavior.

Behind the stage dressing of Secio's objectives, the main objective was to provide a fully worked example of DSM. An important component in convincing people that DSM works is making the code generated from example models actually run. This presents several challenges not normally found in a real-world case of DSM. First, the most significant challenge is that the users must have a compiler installed for the language of the generated code. These days, compilers tend to come as part of a full IDE, requiring a large investment of time, disk space, and possibly money to set up. Second, the user must have a runtime environment compatible with the format produced by the compiler. For both compiler and environment, there may also be various settings such as paths that are specific to each PC, and these settings will normally need to be synchronized with the generated code. Finally, the users will have a variety of different backgrounds, and varying experience with different languages, IDEs, and programming styles.

The need to make an application that would compile and run on many different platforms pointed away from C, the natural language choice for an embedded system. While core watch behavior in C would have been platform independent, the GUI widgets and graphics library in C tend to be platform specific. Also, the majority of commonly used C compilers and IDEs that users were likely to have are commercial products. The installation and use of the freely available compilers are generally sufficiently difficult to deter someone whose only motive would be to see an example application.

While other languages with freely available compilers and runtime environments existed, the mainstream choice—and hence most likely to already exist on users' computers—was Java. The resulting Java applications would also be small enough to be easily passed to other users, and availability of at least a Java runtime environment

on a PC is virtually guaranteed. While Java was not the current language of choice for embedded development, it was originally developed for such devices and would be familiar to a large number of users.

9.2 DEVELOPMENT PROCESS

The development of the watch modeling language was carried out by Steven Kelly and Risto Pohjonen of MetaCase over a couple of weeks. As this was an example rather than a real-world case, there was no outside customer: as both developers had owned digital watches in the 1980s, they felt qualified to play the role of domain expert. This distinguishes this from the other example cases, where the authors did not have sufficient domain knowledge at the start of the project to create the language on their own. This case thus brings the authors' positions closer to those of readers thinking about their own application domains, and gives us a good opportunity to examine the thought processes of a domain expert. As will be seen, creating a DSM language is largely a question of determining what facts need to be recorded about each application in that domain, and where in the modeling language to store these facts.

The total time spent was approximately 10 man-days, including the modeling language, a full set of models, the generator, and the domain framework. Since this was the first Java project for either developer, the time also included learning the language, its libraries, and development environment. Over the subsequent years, a few more days have been spent on upgrading the framework and generator to work on other desktop platforms, cope with later Java versions, produce watches that run on mobile phones, and add support for model-level tracing of a running watch application. None of these changes has required changing the models, and the result has always been fully running applications, identical in behavior but with environment-related differences in appearance and sometimes in code.

9.3 MODELING LANGUAGE

The initial idea was to have a modeling language for building digital watches. In this section, we will follow the analysis of the domain and the development of the language in chronological order.

9.3.1 Reusing Watch Applications in a Family of Watches

It was soon evident that it would be good to break a watch down into its component applications: time display, a stopwatch, an alarm clock, and so on. Beyond providing a sensible modularization of the whole watch, this would also allow a watch application to be reused in different watches. Since these watches would have different physical designs—displays, buttons, and so on—there was a need for some decoupling of references to these physical elements in the models. For instance, if a model of a watch application wanted to specify that a certain operation was caused by a certain button,

we had to answer questions about whether that button was available in the given physical watch, whether it would be named in the same way or have the same semantics, and what to do if no such button existed.

Thinking about this issue prompted the idea of explicitly modeling a whole group of watches as a family. This would be a separate level of modeling, probably with its own modeling language. Often in DSM such a level exists, but it is not always explicitly modeled: it is enough to simply have several products—watches in this case—each built from its own set of models. However, making reuse of models explicit generally makes it easier to maintain them and concretely shows the dependencies of the various components. For instance, a change in the Alarm application to require an extra button would affect which physical designs of watch the Alarm could be used in. If Alarm had simply been reused, this effect might not have been obvious. If, however, there were a top-level model showing each member of the watch product family, which physical watch body it used, and which watch applications it contained, the effects of that change would be easier to see.

If there was a need for a mapping between the buttons mentioned in a watch application model and those present in a physical watch, there was also the question of how to model this mapping. Would the mapping be included as part of the top-level family model, or would it be a separate kind of model between the top-level family model and the actual watch applications? Further, who would be responsible for building these mappings: the watch application designer, the family designer, the designer of a particular watch model, or somebody else? Similarly, would a separate mapping be required for each pair of a watch application model, for example a stopwatch, with a physical watch body, or could one mapping be reused over many watch applications or physical watch bodies?

The question of the mapping was thus difficult in both technical and organizational terms, and also hard to decide at this early stage. While we did not even have modeling languages for watch applications and watch families, it seemed unrealistic to expect to pick a good solution to a problem that would probably only become apparent once several families had been built. We thus decided to go with the simplest thing that could possibly work: there would be a limited number of named buttons, initially just Up, Down, Mode, and Set. Each physical watch body could contain any combination of these buttons, and similarly each watch application could refer to them directly. While less flexible than a different mapping for each watch, this had a good chance of working well for both watch modelers and users. Both groups would prefer a consistent semantics for the buttons: if in one watch application Up was used to increase a time value, and in another the same function was achieved using Set, learning to use the watch would be rather difficult!

Now we had a fair idea of the contents and division of labor of the two modeling languages. The family model would contain a number of watches, and each watch would be composed there from a number of watch applications and a physical watch body. A physical watch body would specify a number of buttons. These buttons would also play a key role in the definition of the watch applications: different buttons would cause different actions in different applications.

9.3.2 Watch Application Behavior

While the family model would have been easier to work on, we decided to tackle the watch applications next. If DSM were to mean anything, it would have to be able to specify the differing behaviors of the various applications sufficiently exactly that we could generate full code directly from the models. One possible tactic at this point would have been to hand code a couple of watch applications. This would have given us an idea of what kind of behavior they would actually have on a low level, as well as insights into what elements of that behavior might repeat over several different watch applications. However, as neither developer had programmed in Java at this point, it was thought that trying to go this way would be a bad idea. DSM is generally about abstracting from the best practice, and clearly there was no way our first attempts would fit into that category.

Instead, we decided to concentrate on the actual user-visible behavior of the watch applications, and trust that our general development experience and instincts would tell us when we had a sufficiently exact specification to enable full code generation. The first question was thus about the buttons, the only way the end user can interact with the watch: When a user presses a button in a given application, does that button always have the same effect? In other words, could we program a watch as a composition of applications and an application as a composition of button behaviors? In some simple cases, this appeared to be true. In more complicated cases, for example setting the time, it might be true if understood sufficiently broadly. For instance, pressing Set in the Alarm application would start setting the alarm, and pressing Set again would stop setting the alarm. While starting and stopping the set process were two different things, they could perhaps be understood as a toggling of the set process. Going further, though, within the set process it would be normal to press the Mode button to toggle between setting the hours and setting the minutes. However, outside of the set process the Mode button would be expected to exit the Alarm application and move us to the next application.

Although further stretching of semantics might have made it possible to model applications as simply one behavior per button, that behavior would have been more and more conditional on what had occurred before. It thus seemed best to admit that the action taken when a button was pressed depended on what state the application was in. A watch application could thus be modeled as some kind of state machine. Pressing buttons would cause transitions between states, and possibly also other actions.

At this point, it would normally have been a good idea to move from models that were either imagined or drawn simply on paper, to actually building a prototype modeling language. This would generally give a better basis for further decisions, while also making more concrete those things that seemed already decided—and perhaps revealing problems in them. Perhaps because both developers had a deep knowledge of DSM, MetaEdit+, and state machines, we actually continued on paper for a little while longer, looking at the possible kinds of actions.

The clearest action with a user-visible result was the control of the little LCD icons for applications: a Stopwatch icon for when the stopwatch was running, an Alarm icon for when the alarm was set to ring, and so on. We decided to make turning the icons on or off an explicit action, taken when following a transition between two states.

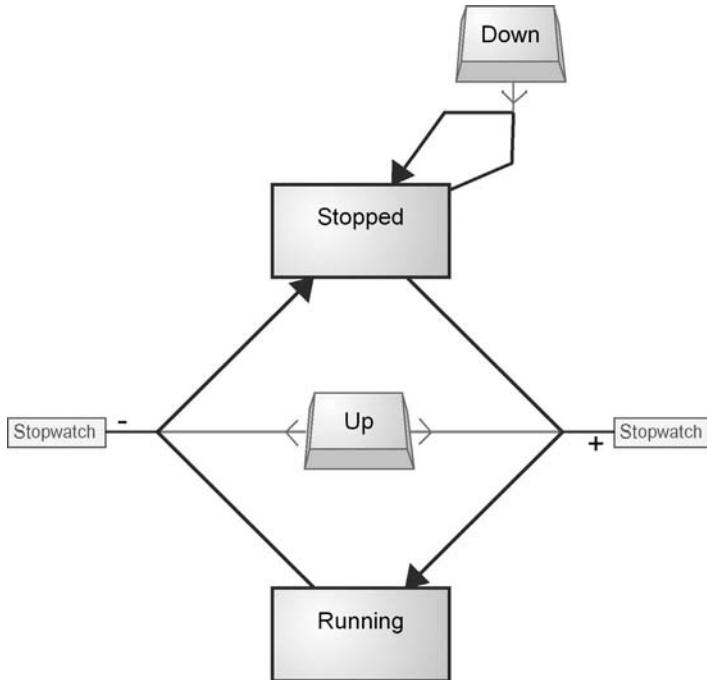


FIGURE 9.1 States, buttons, and icons (40 minutes)

Another possible approach would have been to make each state contain the information about the status of each icon, but we decided against this approach for two reasons. First, the on/off state of the icons was persistent when their application was exited and the next application started, and it was then unclear whether the original application was still in some sense live and in a particular state. Second, and more importantly, storing the icon status in each state would have meant either each state having to store the status of each of the icons, leading to maintainability problems if the number of icons increased, or each application only being able to control one icon, which while initially appearing to be true, might not necessarily be so in more complicated cases. It was also thought that the number of actions needed to make an icon behave as desired would probably be less than the total number of states in that application, making modeling easier if actions were used. We thus decided to represent the icons as objects in the modeling language, and to allow modelers to change their state by drawing a relationship to the icon to turn it on or off. The result so far, after 40 minutes of metamodeling, is seen in Fig. 9.1.

9.3.3 Watch Applications Manipulate Time

After the work so far, we could model a stopwatch that would know which state it was in (say, **Stopped** or **Running**), move between those states when the user pressed a button (say, **Up**), and that would also turn on or off the **Stopwatch** icon when moving

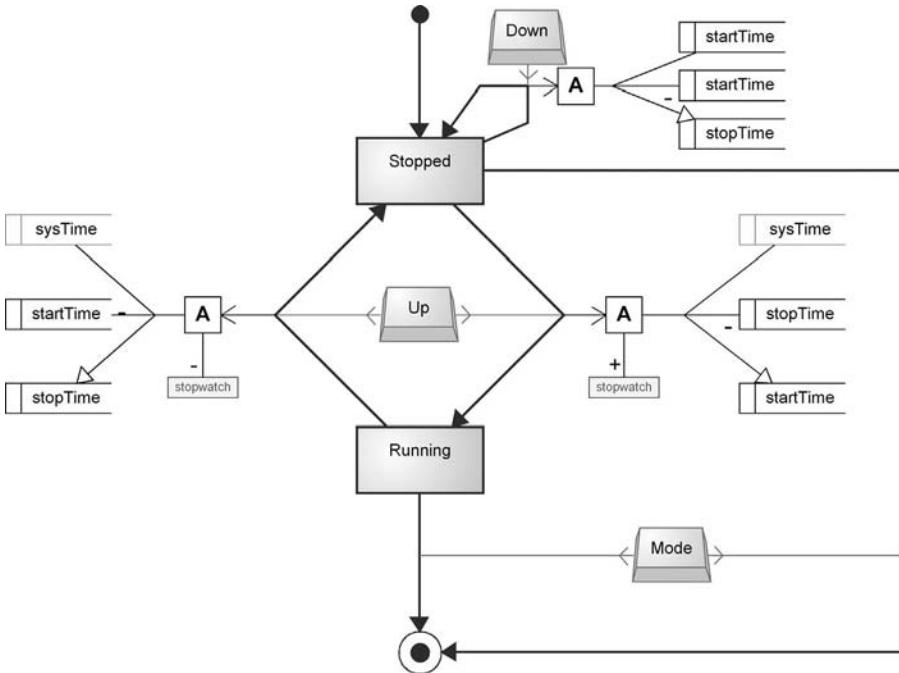
between the states. All very well, but it might be useful if the stopwatch actually recorded something about the elapsed time, too!

If Buttons represented a Controller in the MVC triad that Secio envisaged, and Icons represented a View, then being able to record elapsed time was clearly part of the Model. Storing, recalling, and performing basic arithmetic on time variables were therefore important parts of watch application behavior that we needed to represent in some way in the modeling language. We thought it was fair to assume that any digital watch company worth its salt would already have existing functions to return the current time, along with functions to store and recall and perform basic arithmetic on time variables. This meant we could consider the time operations on a high level, rather than the implementation details of the bits and bytes of data structures to store time.

The first question to be answered was one about the boundaries of the domain: did we need to handle dates as well as time? Thinking through the various possible watch applications, it seemed that most had no connection to dates: an alarm could not be set for a given date, for instance. In fact, only the basic time display would show dates. The handling of dates would most likely be identical from watch to watch: a source of commonality, not of variability. In addition, the behavior of dates is complex, with weekdays, day numbers, months with different numbers of days, leap years, different orderings of date elements for different countries, and so on. All in all, it looked like building a modeling language for handling dates would not result in any productivity improvements for Secio. Rather than introduce complexity for no gain, we decided to leave dates out at this point and concentrate on time.

Thinking about the stopwatch application, it seemed there could be two ways to think about time variables and the underlying platform. In the first way, the platform would just respond with the current time when asked, and we could store that and then later ask again for the new time and calculate the difference. In the second way, the platform could offer a service to create a new timer, which it would then continuously update. While the second way would make modeling the stopwatch easier, it seemed unlikely in the resource constrained embedded environment of a digital wristwatch: updating multiple timers simultaneously would place unnecessary demands on processing power and battery life. We would, therefore, need a way of representing subtraction of time variables, and most likely addition too.

Thinking about other watch applications such as the alarm clock, it appeared there might be a need for another time operation. When editing the alarm time, pressing Up would increment (say) the minutes, which could be represented as adding one minute. However, when the minutes reached 59, the next press of Up would take them to 00, without incrementing the hours. In other words, this was not a true increment of adding one minute, but rather an increment to just one unit of the whole time, which would roll around at either end of the unit's range. This could have been represented in the models with a comparison operator and an subtraction, for example: "if minutes equals 60 then subtract 60 from minutes." However, this would be needed for every edit operation, for every unit of the time, and once for each end of the range. It would also require a modeling concept for comparisons, which we did not seem to need otherwise. We therefore decided to make a "Roll" operation a first-class concept in the

**FIGURE 9.2** Time manipulation (70 minutes)

modeling language, incrementing or decrementing a particular unit in the whole time variable.

Now we knew the concepts we wanted for time variables and operations, how best to represent them in the modeling language? Programmers would be used to a textual representation, and we could indeed have had a textual DSL providing a natural syntax, restricted to just these operations. However, because there were so few operations needed, and each calculation would need only one or two operations, we decided to represent time variables as actual objects in the modeling language, and operations as relationships connecting to them. This would fit well with the earlier decision to have icons as objects, and actions on them represented as relationships drawn to them from state transitions. The result so far, after 70 minutes of metamodeling, is seen in Fig. 9.2.

9.3.4 Time Display

At this point, we had the majority of the concepts of the modeling language and had covered the Model and Controller parts of the MVC triad fairly thoroughly. We also had addressed some of the View part in the form of the Icons, but the most important View element, the display of time, was still largely untouched.

The display code of a real-time application is notorious for being difficult to get right. Had Secio been a real organization, its experienced developers would have been throwing up their hands at this point and saying “All you’ve done so far are the easy

bits, and we never had problems with those anyway. Getting the display to update smoothly in real time is hard work, and you've not even touched on it. Besides, that's the area where our developers make most of their mistakes: they just can't seem to get the thread synchronization and semaphores right, no matter how many times we explain it."

What then can we do for Secio? First, we can take a long step away from the implementation details toward the problem domain and see what actually needs to happen on the watch display from the end user's point of view. Let's take the most used application, Time. What is it displaying? The current time, of course! Now, the value of the current time is changing all the time, so are the updates for that something we need our models to specify? In a sense, maybe not: what is being displayed, on a high level of abstraction, is simply the current time. What then about the Stopwatch, what is it displaying? That seems to vary a little: initially, it just displays all zeroes; when it is running, it is displaying the elapsed time, which is updating constantly but always equal to the current time minus the original saved starting time. If we look at World Time, that displays the current system time adjusted by some fixed offset. A Countdown Timer displays the time the alarm will sound minus the current system time.

This is interesting: we seem to be able to express quite simply the basic idea of what each application should be displaying. This is actually not all that surprising: if the time value displayed required some immensely complicated algorithm to calculate, few people would be able to interpret its values. Of course, actually making the value display and update smoothly will be tricky—real-time software always is—but perhaps the models themselves can remain quite simple. Could we abstract out the complicated parts of real-time display into the domain framework? This would allow the model to simply specify *what* to display, while an expert developer's handwritten code, written once but reused for all displays in all watch applications, would handle *how* to display it.

In our blessedly deep ignorance of the intricacies (and bugs!) of Java's thread handling and synchronization, we decided to take this path. A watch application would specify a calculation to obtain the time to display, either just one such Display Function for the whole application (e.g., Time, Alarm) or perhaps different Display Functions for different states (e.g., Stopwatch states for being stopped and running, or Countdown Timer states for setting the timer and counting down). The modeler's burden would end there, and not a thread synchronization or semaphore in sight. The expert developer would write a display update function that would run in a separate thread once every few milliseconds, ask the application to perform its Display Function calculation, and update the display with the result. Our example model at this stage, after 105 minutes of metamodeling, is shown in Fig. 9.3.

9.3.5 Odds and Ends

When building a modeling language, there are always things that get missed on the first pass. Thinking through a concrete example application and its model helps keep things on track, but any given example will rarely contain an instance of everything

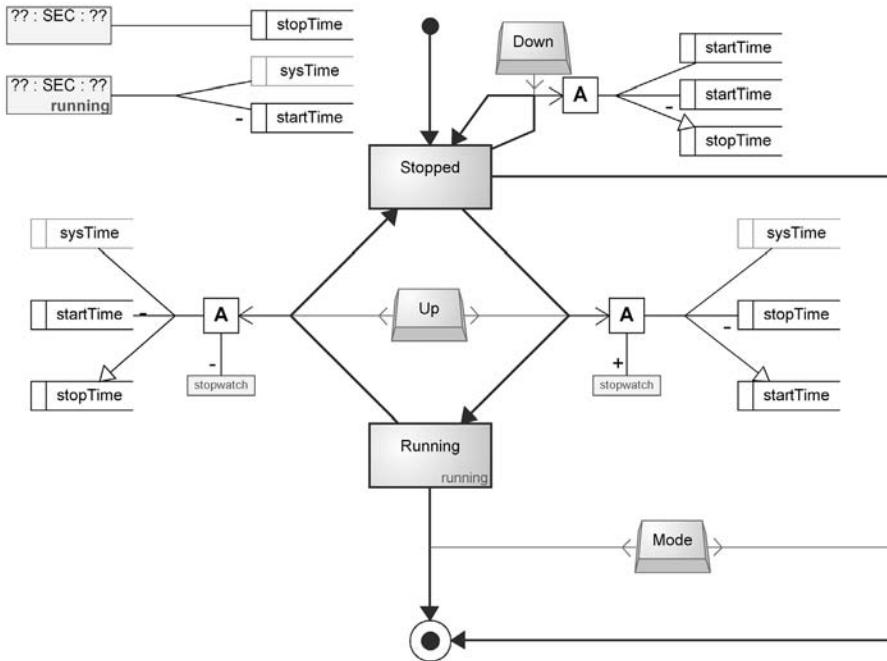


FIGURE 9.3 Display Functions (105 minutes)

needed in the modeling language. So too in this case: Stopwatch had been a good example for many areas, but it was missing features that were used in several other watch applications. We also missed one important part of Display Functions, which only became apparent in the Stopwatch when it was used in several different watch models.

The most obvious omission from the modeling language was the complete lack of the concept of an Alarm. It had been mentioned early on but never focused on for long enough to actually figure out how it should be modeled. Setting the alarm seemed easy enough: we could edit some time value for it, either the time at which the alarm would ring, or the amount of time until it would ring. An alarm ringing would be similar to a button press in that it could cause a change of state. More interestingly, this change of state could happen at any time, even if the watch was running a different application at that point. Clearly, we could not draw transitions from every state in the entire model in case the alarm rang there. Instead, we settled on having an alarm symbol, and a single special “alarm” transition from that to a state. Whenever the alarm rang, its application would take control and jump to that state. On exiting the alarm application, the watch would return to the application that was current when the alarm rang.

In deciding whether to set alarms at a time of day or as an amount of time until the alarm rang, we hit a problem. What would happen if the user changed the watch time after setting the alarm? If we stored alarms as a time of day, a standard Alarm Clock application would be fine, but a Countdown Timer would suddenly be wrong.

For example, if a Countdown Timer was set at 5:00 p.m. for 10 minutes, it would have stored its alarm as 5:10 p.m.; if the user moved the clock back an hour at 5:05 p.m. to 4:05 p.m. the countdown alarm would not ring for another 65 minutes. Conversely, if we stored alarms as an amount of time until ringing, the Countdown Timer would work fine, but a standard Alarm Clock would ring at the wrong time. We decided to solve this by allowing each Alarm object to specify whether it responded to changes in local time or not.

Thinking about setting alarms also reminded us that we needed some way to make a pair of digits on the time display blink, so the watch user would know which time unit he was editing. While we could have avoided adding a concept by saying that a time unit blinking was just a special case of an Icon, this would have meant that going from an “Edit Hours” state to an “Edit Minutes” state would have had to turn off “blink hours” and turn on “blink minutes.” We would also have to make sure that every path to “Edit Hours” would turn on the “blink hours” pseudo-Icon. A better approach seemed to be to recognize from the names of the states that blinking of a time unit was actually a feature of the state itself. We thus added a property “Blinking” to State, making it a list where the user could choose from Hours, Minutes, or Seconds. This also fitted well with choosing a Display Function in each state: the actual display thread would need to know both the function and any blinking time unit to be able to keep the display updated. Having the blinking property in the Display Function itself would not have helped, as it would mean having to create duplicate Display Functions differing only in which time unit was blinking.

Display Functions were also the source of the third addition to the modeling language. We realized that the mapping of the different time units of a time value to actual digit pairs on the watch display could be different in different physical watches, and also different between two watch applications in the same physical watch. For instance, in a lady’s watch there are often only two digit pairs. In the normal time display these are used for hours and minutes, but in a Stopwatch and possibly in a Countdown Timer, they should show minutes and seconds. However, editing a Countdown Timer generally only allows choosing hour and minute values, so while editing those should clearly be shown. Specifying a hard mapping in a watch application from time units to digit pairs would not work though, since it would mean the application would not work so well in a different watch model with a different number of digit pairs.

We tried several different schemes for specifying mappings, including specifying which time unit would be shown in the leftmost or rightmost digit pair. Neither of these gave satisfactory results when we wrote out on paper what would be displayed in various states, applications, and physical watches. In the end, we hit on the idea of a Display Function specifying a “Central time unit,” where “Central” was a mixture of “most important” and “should be displayed in the center digit pair.” A little heuristic—for what counts as “center” when there are an even number of digit pairs—rounded off the scheme, and we added an appropriate property to Display Function. Now we had a language that could specify all we could think of about watch applications.

9.3.6 Putting it all Together

Having a language that could specify watch applications brought us back to our starting point: Secio wanted to be able to compose a product family of Watches out of Watch Applications. One part of each Watch would thus clearly be a list of the applications it contained, for example, Time, Stopwatch, and Alarm. A Watch would not just be a jumble of applications though: the user would cycle through the applications in a certain order. At first, we thought this would be represented as a collection property, to which the modeler could add the desired Watch Applications. The idea of the cycle, however, gave us the idea of representing this graphically. Since we had already considered the possibility of having layered state machines for the Watch Application—a State in an application could decompose to a subapplication—we hit on the idea of using the same Watch Application modeling language to specify application composition.

This approach brought several benefits. First, it reduced the number of concepts the modeler would have to learn. Second, it gave the modeler more freedom in deciding how many levels the models should be divided up into: if a Watch had only one Application, for example a Stopwatch, the Watch could point directly to the Stopwatch, rather than having to specify an intermediate level with a single Stopwatch Application. Similarly, if a given application became too large for a single graph, or if parts of an application could be reused in other applications, they could be separated out into their own subapplications—on as many levels as seemed appropriate. Third, it allowed the possibility of more freedom at the application composition level: rather than restricting all Watches to always move between their applications in a fixed cycle, the choice of which application to go to next could be as complicated as the modeler desired. An example model is shown in Fig. 9.4.

While one part of a member of the product family of Watches would thus specify the behavior of the Watch, a second part was needed to specify the physical details of the Watch. Mostly these would be the domain of an industrial designer, but the behavioral part would also need some of that information. In particular, if we wanted to generate a Java applet as a Watch emulator, we would need to know three things: the buttons on the watch, the icons it had, and the number of digit pairs it could display. The first two could in theory have been found by trawling through the set of applications and including all referenced buttons and icons, but this was not how we wanted things to work. We wanted a Stopwatch application, say, that used Up, Down, and Mode, to be usable in a watch without a Mode button, for example one containing only the Stopwatch application. Similarly, rather than have an error if an application tried to turn on an icon that was not present in the physical watch, we would simply do nothing at that point. This made the watch applications more reusable, and allowed the application modeler to concentrate on the application itself, without having to know beforehand the exact details of the physical watches in which it would be used.

We decided to call a member of the product family a Watch Model—“model” being used in the sense of “a Corvette is a car model,” rather than “a graphical model of a car.” Each Watch Model had a Logical Watch, which pointed to the graph showing

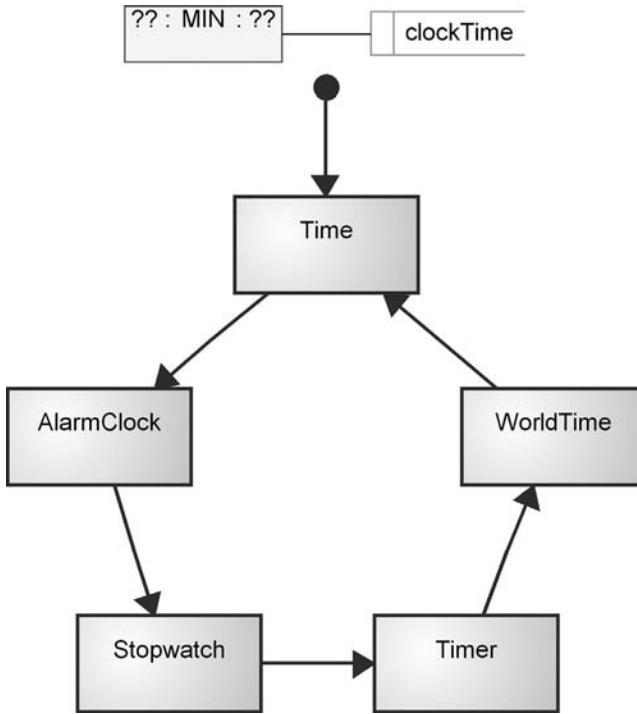


FIGURE 9.4 Logical watch as a cycle of applications

the cycle through the watch's applications, and a Display, corresponding to the relevant parts of the physical watch body: icons, digit pair zones, and buttons. The Logical Watches and Displays would thus form components, and each might be used in more than one full Watch Model.

As these concepts were different from those used in the Watch Application modeling language, it was clear that we actually had a new modeling language here. We decided to make it graphically rather verbose, for pedagogical rather than practical reasons. Each Watch Model (rounded rectangles with a light green fill) showed the Display and Logical Watch it used, and the sets of possible Logical Watches and Displays were also shown in the same diagram (Fig. 9.5).

Another possibility here would have been to use a matrix representation rather than a graphical diagram. In models shown as matrices, the axes contain objects and each cell contains the relationship (if any) between the corresponding objects. The Logical Watches could thus be listed down the left side of the matrix and the Displays across the top. The cell at the intersection between a given Logical Watch and Display would then represent a possible Watch Model composed of those parts. The relationship's name would be the name of the Watch Model and would be shown in the cell. Fig. 9.6 shows how this would provide a useful visual

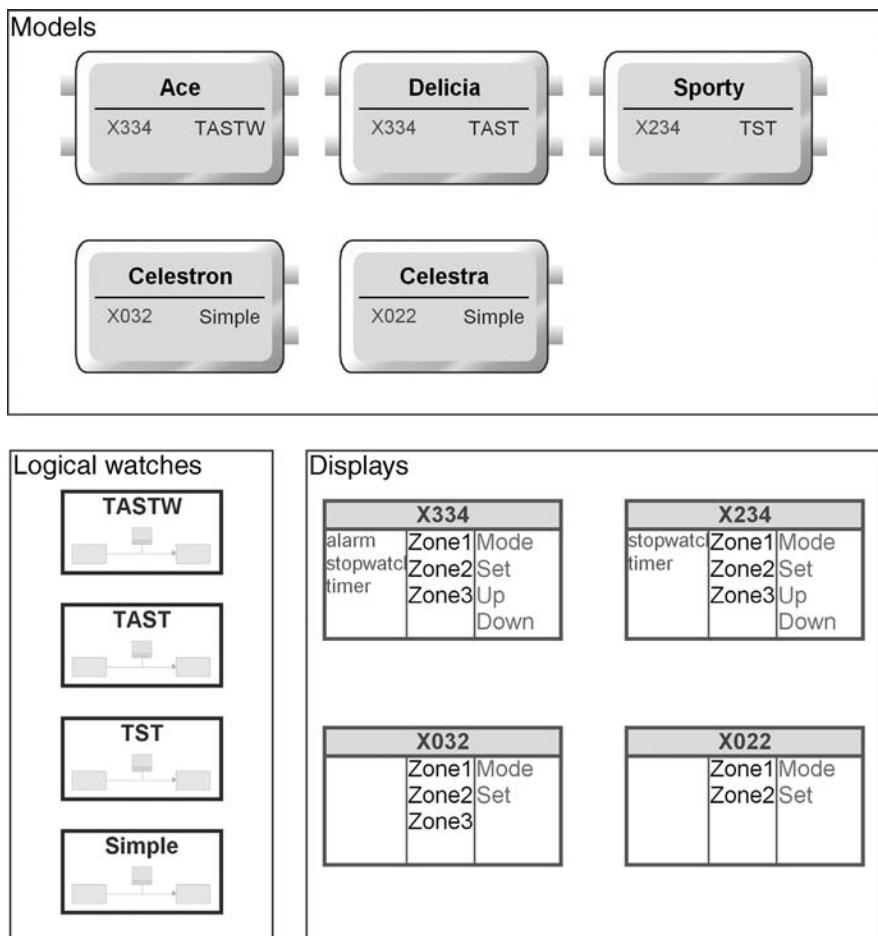


FIGURE 9.5 Watch Family diagram

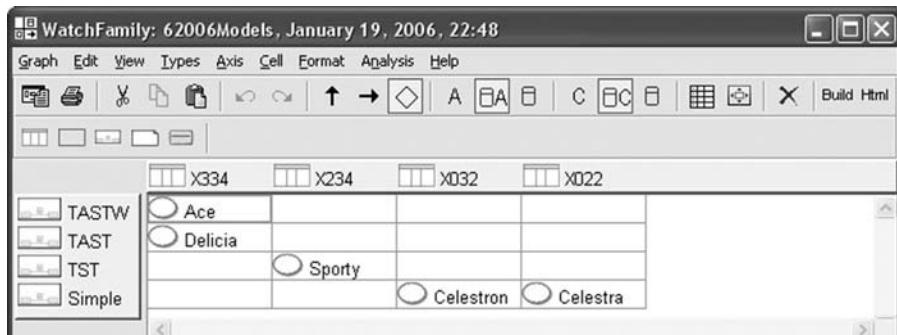


FIGURE 9.6 Watch Family as a matrix

overview of which potential combinations of Logical Watches and Displays had been used, how often each Logical Watch and each Display had been used, and which potential combinations remained as possible openings for new products. The matrix format would also collect the Watch Models, Logical Watches, and Displays together into their own areas, without the rather clumsy group symbol of the graphical diagram.

9.3.7 Rules of Watch Applications

The main need for rules and constraints in the Watch applications, as in most DSM languages, was in the relationships. The main relationship was the Transition between States, triggered by a Button and causing various kinds of actions. This was a clear case of an n-ary relationship: one relationship connecting several objects.

One question was how much we wanted to allow the aggregation of several transitions into one composite transition. For instance, it would have been possible to allow the same transition to come from many States: for example, a transition to the Stop state, triggered by the Mode button, would often be appropriate for several States. Similarly, we could have allowed the same transition to be triggered by multiple buttons (“Press any button to stop alarm ringing”), and a transition could clearly trigger multiple actions (e.g., setting an alarm and turning on the alarm icon).

However, as the transition would generally link four objects anyway—quite a large number—it was thought better to allow one From State, an optional Event from a Button, an optional Action, and one To State. A small Action object would serve as a placeholder for multiple actions, which would be connected to it by relationships. The Event role from the Button object would be optional: some state transitions could happen without any button press.

The transition from the Start State to the first State would be required not to have a Button: an application is never actually in the Start State, which simply marks the entrance point. Additionally, there was a constraint that a Start State could have only one transition leaving it, and similarly there should only be one Start State in each Watch Application. Interestingly, the Stop State behaves differently: there can be many Stop States, each with possibly many transitions arriving at it. While an application is never actually in a Stop State, there is no ambiguity resulting from this situation, in contrast with the case of Start States.

The other main kinds of relationship whose rules required thought were the time operations. We decided to implement these as Plus, Minus, and Set roles. The contents of the variables pointed to by the Plus and Minus roles are added and subtracted appropriately, and the result of the operation is stored in the variable pointed to by the Set role. There could thus be one Set role, and zero to many Plus or Minus roles. As this left the unsatisfactory possibility of a relationship with one Set role and no other roles, we decided that there should also be a Get role, which would point to the first variable term in the operation. Thus, for something like

“stopTime = sysTime – startTime” stopTime would be in a Set role, sysTime in a Get role, and startTime in a Minus role.

As it turned out, this was a good decision for code generation: the first term in a calculation is not generally preceded by a plus or minus sign. It was, however, a slightly poor decision in terms of modeling all possible calculations: for example, “stopTime = – startTime.” Interestingly, there has not yet been a need for such a calculation. Still, perhaps a better solution would be to replace the Get, Plus, and Minus roles with a single Get role with a property specifying whether it was Minus or Plus.

9.3.8 Notation for Watch Applications

To model an application in sufficient detail to allow full code generation generally requires several different kinds of information. In general-purpose modeling languages each kind of information would be separated into its own language. A clear problem with this approach is that it is hard to get a complete view of the application, as it is split over several diagrams. Such an approach also requires a fair amount of duplication, as objects (or references to them) must be reused in several different diagrams, in order to provide the full information about the object from all viewpoints. This leads to another problem: the difficulty of keeping all the diagrams synchronized.

To avoid these problems, a domain-specific modeling language often includes several different viewpoints within one language. Because the language constructs needed for each viewpoint are only a domain-specific subset of those needed for that viewpoint in a generic modeling language, the resulting language remains of a manageable size. It does however contain a variety of different kinds of information, and one way to use notation is to help separate the different kinds of information visually.

The Watch Application modeling language contains three main types of information, corresponding to the three parts of MVC that Secio had decided to use. The Model part represents the underlying data on which the application acts, in this case time variables. These parts of the model are shown in black or gray, with gray being used for read-only pseudovariables, such as sysTime, by analogy with graying out a field in a user interface to show it is read-only. The View part represents the output of the application to the user, that is, the visible parts of the application that change as it runs. These parts of the model are shown in green: the Icons and Display Functions. The Controller part represents the input to the application by the user, in this case the Buttons. The Buttons, and the lines from them to the Transitions, are shown in red in the models. Since the behavior of the buttons depends on the State the application is in, we decided to make the States and Transitions a dark red: part Controller, part Model. As each State showed the name of the Display Function it used, which was part of the View information, that name was shown in green.

The symbols themselves were drawn as vector graphics, aiming more for simplicity than beauty. As a State has no clear visual presence in a physical watch, its symbol was just a rectangle containing its name. If its Display Function was not the default for that graph, that was shown in green in a smaller font at the bottom right. If one of the time units was blinking in that State, the abbreviation of the time unit was

shown with four short rays pointing outward. For the Start and Stop States, we chose the familiar representations of a filled circle and a filled circle within a hollow circle. Buttons clearly had a physical counterpart, but also a fairly standard pictogram of the edges of a stylized three-dimensional button. As the physical counterparts on actual watches tend to be very small, we used the pictogram. For the Alarm we drew a yellow bell shape, using a smaller red version of it as a symbol for the special Alarm Transition relationship. Time variables lacked a clear visible counterpart, but were a familiar concept from several generic modeling languages; in the end, we chose the Data Store symbol from Data Flow Diagrams.

An important part of any modeling language is the flow of control or information, which is normally indicated by arrowheads on the role lines. We thus placed arrows on the lines from a Button to the Transition it triggered, from the Transition to the Action it executed, and from the Transition to the State it ended up in. As the last of these was the most important element of flow in the whole modeling language, it was shown most visibly, with a dark red filled arrowhead. The final piece of flow was in the time operations, where the value of the calculation is assigned to a time Variable: this was shown with a hollow black arrowhead.

As always, these decisions were a balancing act, trying to make the meaning of the various parts of the model clear without straying into a graphical melee where every part tries to scream its own importance. If the direction of information flow along a role line was obvious from other context, no arrowhead was used. This was the case for Actions turning Icons on or off, the Get, Plus, and Minus roles in time calculations, and the role connecting a Display Function to its calculation. The direction for the Button and Action roles of the Transition relationship could also be implied, but small open arrows were added to make clear the part played by each of the roles in this n-ary relationship.

9.4 MODELS

The three models we have looked at earlier in the chapter reflect the three levels found in most of the watch applications. The top level is a single diagram in the Watch Family language (e.g., 2006 Models shown in Fig. 9.5). Each Logical Watch there is decomposed into a simple Watch Application diagram (e.g., TASTW shown in Fig. 9.4) at the middle level showing how it is composed from various applications, represented as a cycle of states. Each state there is decomposed into its own Watch Application diagram (e.g., Stopwatch shown in Fig. 9.3) containing the behavior of that application.

The AlarmClock application in Fig. 9.7 shows some aspects not present in the Stopwatch. When the application starts, we move straight to the Show state, where (as indeed in all states in this application) the Display Function simply shows the alarmTime variable. Pressing the Set button takes us to the EditHours state, where the hours are flashing on the display. Pressing the Up or Down buttons there rolls the hours value of alarmTime up or down. Note that the thick dark red role from EditHours

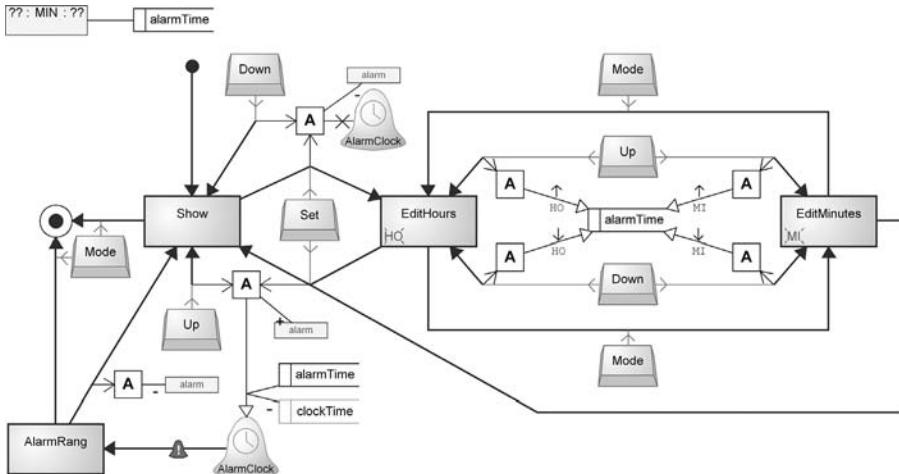


FIGURE 9.7 AlarmClock application

returns back to EditHours along the same path, ending in an arrowhead, making the role leaving EditHours invisible. Another way to draw this would be as a circular relationship, but as these are so common, it was thought that the modeler would prefer this shorthand: all four roles of the transition are present; the From role is just not particularly visible. Pressing Mode in EditHours takes us to the similar EditMinutes, and vice versa.

Pressing Set in either Edit state takes us back to the Show state, carrying out the Action immediately below and to the right of Show on the way. This turns on the alarm icon and sets the AlarmClock alarm to ring after a while, calculated as alarmTime minus clockTime. The alarm is set to be sensitive to changes in local time (indicated by the clock face in its symbol). In the Show state we can press Down to unset the alarm and turn off the icon. Assuming we leave the alarm to ring, when it rings it follows the transition with the small red bell symbol from the Alarm symbol to the AlarmRang state. From there, without any buttons being pressed, we move directly to the Show state, turning off the alarm icon on the way. From the Show state, pressing Mode will exit the application.

9.4.1 Use Scenarios

An important feature of the Watch modeling language is its support for reuse of models and model elements. This is present at all levels, and across different watch families, logical watches, and applications. Reuse here means that two or more parts of a model refer to the same model element. Since this is a reference rather than a copy, updates to the reused element will instantly and automatically be in effect in all parts referring to that element. Reusing rather than copying thus reduces the amount of work needed for maintenance, reduces the risk of corrections only being made to one

copy, and makes the amount of data in the models smaller and thus easier to understand.

At the Family level, the same Logical Watch can be reused in several Watch Models: for example, the Simple application for viewing and setting the time is used in both Celestron and Celestra watches. Similarly, the same Display can be reused in several Watch Models: for example, X334 is used in both Ace and Delicia. The Buttons and Icons used in Displays are each defined only once, and reused in each Display that has them.

At the middle level, each reference to a lower-level Watch Application is clearly a case of reuse. Since the name of a State in the middle-level diagram is the same as the name of the Watch Application it decomposes to, a State for a given Watch Application would be essentially identical between different Logical Watches. We thus reuse the whole Stopwatch State from the TASTW Logical Watch in the TAST and TST Logical Watches. Any improvements to the Stopwatch will thus instantly be a part of all Watch Models that use these Logical Watches.

At the bottom level, the Buttons and Icons are reused from their definitions at the Family level. The global pseudovariables such as sysTime are also defined only once and reused in many Watch Applications. This is in contrast with the true Variables and the Alarms, which are defined local to a given Watch Application. They may be reused within that one application, for instance first in a calculation that sets the value, and later in a calculation that reads the value.

Looking at the Watch Applications, both Timer and Stopwatch contain a variable called stopTime, but these refer to different things: a Timer's stopTime is the time of day when its alarm will ring, whereas a Stopwatch's stopTime is the amount of time that had elapsed when the stopwatch was stopped. Two Variables may thus share a name, but because of the scoping they are two different objects in the models, and two different variables in the running watch applications. The scoping is implicit in the modeling language, and made explicit in the code generation.

The same Display Function will normally be referred to by many States. In the simplest case, all States in a Watch Application will use its single Display Function. To make this as easy as possible for the modeler, such a Display Function can be given a blank name, and then each State that does not define a Display Function will use this default Display Function.

In the Watch Application diagrams we also take advantage of another kind of reuse: reuse by relationship. The clearest case is the reuse of Actions: an Action can be reused by more than one Transition, simply by having each Transition link to it with its Action role line. Another case, so obvious that it might be overlooked, is the reuse of States, which are reused within a single application in the sense that they may be reached by more than one path.

Since in some cases trying to link disparate objects together can lead to a visual spaghetti of crisscrossing lines, it is also possible to make representational duplicates of objects in a diagram. For instance, in the Time application in Fig. 9.8, it is possible to return from the EditMinutes, EditHours, and EditSeconds states to the basic Show state, via an ExitEdit state that makes the edited time persistent. Rather

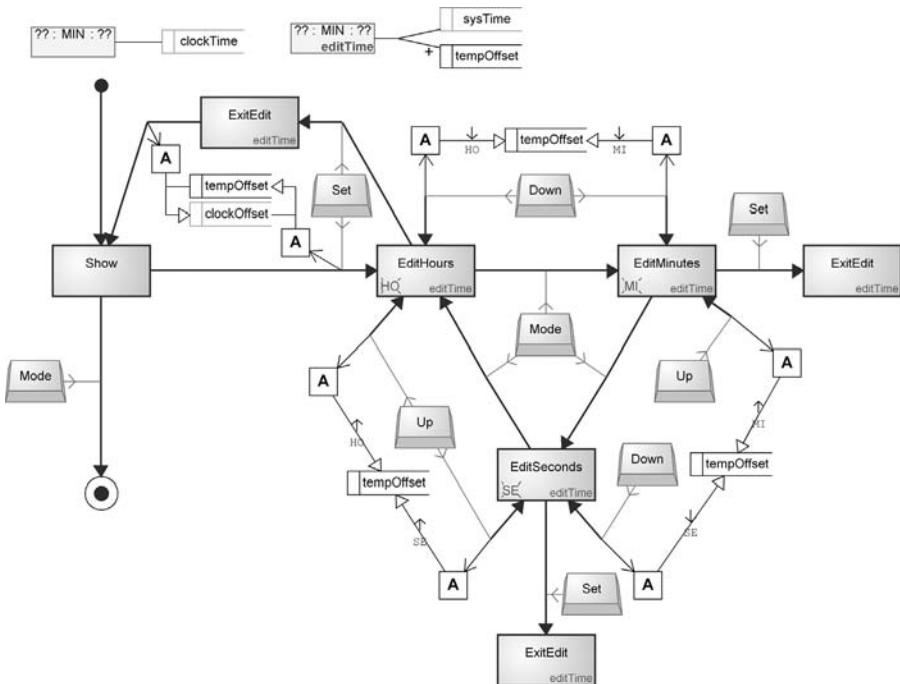


FIGURE 9.8 Reuse in the Time application

than try to draw three relationships all arcing back to the start, a duplicate of the ExitEdit state is placed conveniently close to each of the Edit states, allowing a single short transition for each. No transitions leave these duplicates, but the model knows they are all the same as the main ExitEdit state, so the transitions that leave there are applicable in the duplicates too. Conceptually we can think that we jump straight from a duplicate to the main ExitEdit state, but in fact there is no difference, and no one representation of the ExitEdit state is pre-eminent. The ExitEdit state simply has the sum of all of the transitions that enter or leave any of its representations in this graph. Since having real reusable objects like this may be too radical for more conservative users, it would also be possible to have a new object type “State Reference,” and use that for the other representations of ExitEdit. Of course, if the users are that conservative, you should just let them code the whole thing in assembly language and forbid the use of subroutines—or simply give them a nice analog watch as a retirement gift... .

The Time application is also an example of a possible type of reuse that has not been implemented in this modeling language. There is clearly something similar in the States, Buttons, Transitions, and Actions for EditHours, EditMinutes, and EditSeconds. Each Edit State selects one time unit of the same time Variable, and the Up and Down Buttons roll this time unit up and down. A similar pattern is seen in

other applications where a time variable is being set by the user, that is, Alarm and Timer. There, however, only the hours and minutes are being set, and there are differences between the three cases as to the Display Function used. Because of these differences, it was not possible to build a subapplication that each of these three applications could use.

We did, however, consider a new modeling language concept, TimeEditor, which would have stood for a set of related states like this. A TimeEditor instance would have specified the variable to be edited, the first and last time units to be edited, and the Display Function to be used. This information would have been sufficient to cover the variability between the three cases, and we could have built a domain framework function to handle the behavior, and made the generator create a simple call to that function.

With only three cases, however, and none of these being impossibly complicated with the existing modeling language, we decided to leave the extension until later. Rather than creating a new concept for this particular kind of repetition in the models, it might prove to be better to identify several kinds of repeated patterns and come up with a more general way to specify parameters to a subapplication. In the long run, this approach would probably have allowed more expressive power with a smaller set of concepts.

9.4.2 Watch Application Metamodel

For completeness, Fig. 9.9 shows the full metamodel for Watch Applications.

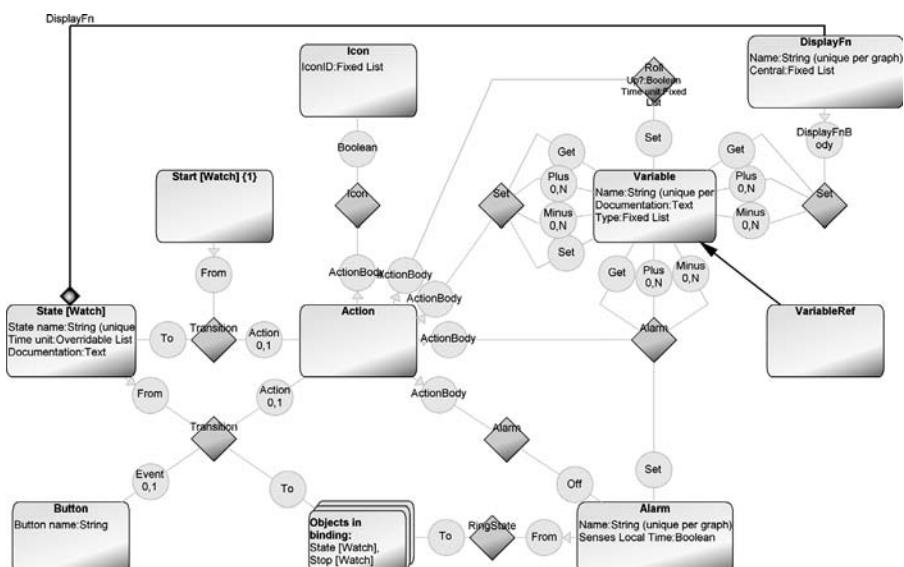


FIGURE 9.9 Watch application metamodel

9.5 CODE GENERATION FOR WATCH MODELS

Generators were built for watch models to both Java and C. We will look at each in turn, focusing first on Java.

9.5.1 Java Generator for Watch Models

In the watch example, the generator goes beyond simply producing Java code corresponding to the models: It also takes care of creating and running build scripts, and ensuring that the domain framework is available. Generation proceeds with the following steps:

- (1) The batch scripts for compiling and executing the code are generated. As the mechanisms of compilation and execution vary between platforms, the number and content of the generated script files depend on the target platform.
- (2) The code for the domain framework components is generated. This is an unusual step, since normally any domain framework code files would already be present on the developer's local disk or referenced from a network share. For the purposes of the Watch example as part of an evaluation package, however, we had to provide the framework code files and make sure they were installed in an appropriate place. The easiest way to do this was simply to include them as textual elements along with the models. This way we ensured that all required components were available at the time of compilation, and we also had control over the inclusion of platform-specific components.
- (3) The code for logical watches and watch applications is generated. The state machines are implemented by creating a data structure that defines each state, transition, and display function, along with their contents. For each action, the code generator creates a set of commands that are executed when the action is invoked.
- (4) The generated code is compiled and executed as a test environment for the target platform. Basically, this step requires only the execution of the scripts created during the first step.

How was the code generator implemented then? Each generator in MetaEdit+ is associated with a certain modeling language and can thus operate on models made according to that specific language. To avoid having one huge generator, generators can be broken down into subgenerators and call each other, forming a hierarchy or network. The top level of the Watch DSM solution is the Watch Family modeling language, and this also forms the top level of the generator. The generators at this top level are shown in Fig. 9.10 (a “*” in the name of a subgenerator denotes an individual version for each target platform). Arrows denote calls to subgenerators, and the order of execution is depth first and left to right.

At the top is the main generator, “Autobuild.” The role of “Autobuild” here is similar to that of the “main” function in many programming languages: it initiates the

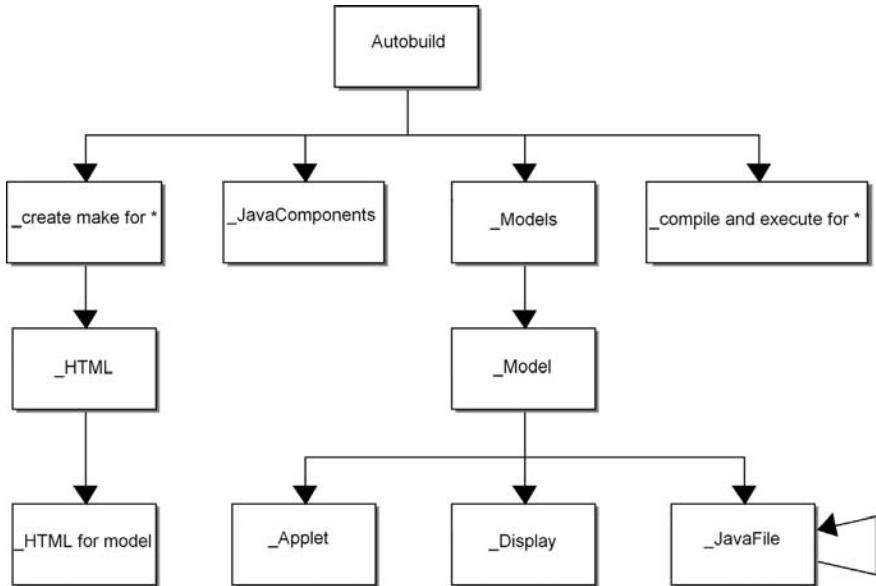


FIGURE 9.10 The watch code generator architecture, part 1

whole generation process but does not itself contain anything other than calls to the subgenerators on the next level down. The subgenerators on the next level relate closely to the steps of the Autobuild process presented earlier in this section. As “_JavaComponents” just outputs the predefined Java code for the framework components, and “_compile and execute *” just executes scripts produced during the earlier steps of the generation process, we can concentrate on the more complex subgenerators, “_create make for *” and “_Models.”

The basic task of the “_create make for *” subgenerators is to create the executable scripts that will take care of the compilation and execution of the generated code. As this procedure varies between platforms, there is an individual version of this subgenerator for each supported target platform. If there are any specific platform-related generation requirements, for example creating the HTML files for the browser-based test environment in Fig. 9.10, they can be integrated into the “_create make for *” subgenerator.

The responsibility of the “_Models” and “_Model” subgenerators is to handle the generation of code for the Watch Models, Logical Watches, and Watch Applications. For each Watch Model, three pieces of code are generated: an applet as the physical implementation of the user interface, a display definition to create the specified user interface components, and the definition of the logical watch.

An example of the code generated for an applet (produced by the “_Applet” subgenerator) is shown in Listing 9.1. The generated code defines the applet as an extension of the AbstractWatchApplet framework class and adds the initialization for the new class. Values from the model are shown in bold.

Listing 9.1 Generated code for an applet.

```
public class Ace extends AbstractWatchApplet {
    public Ace() {
        master=new Master();
        master.init(this,
            new DisplayX334(),
            new TASTW(master));
    }
}
```

The simple generator that produced this is shown in Listing 9.2, with the generator code in bold.

Listing 9.2 Code generator for the applet (generator code in bold).

```
'public class ' :Model name; ' extends AbstractWatchApplet {
    public ' :Model name; '() {
        master=new Master();
        master.init(this,
            new Display' :Display:Display name; '(),
            new ' :LogicalWatch:Application name; '(master));
    }
}'
```

The display definition can be generated in the same vein by the subgenerator “_Display,” as shown in Listing 9.3. Again, a new concrete display class is created, inheriting from AbstractDisplay, and the required user interface components are defined within the class constructor method.

Listing 9.3 Generated code for a display.

```
public class DisplayX334 extends
AbstractDisplay
{
    public DisplayX334()
    {
        icons.addElement(new Icon("alarm"));
        icons.addElement(new Icon("stopwatch"));
        icons.addElement(new Icon("timer"));

        times.addElement(new Zone("Zone1"));
        times.addElement(new Zone("Zone2"));
        times.addElement(new Zone("Zone3"));

        buttons.addElement("Mode");
        buttons.addElement("Set");
        buttons.addElement("Up");
        buttons.addElement("Down");
    }
}
```

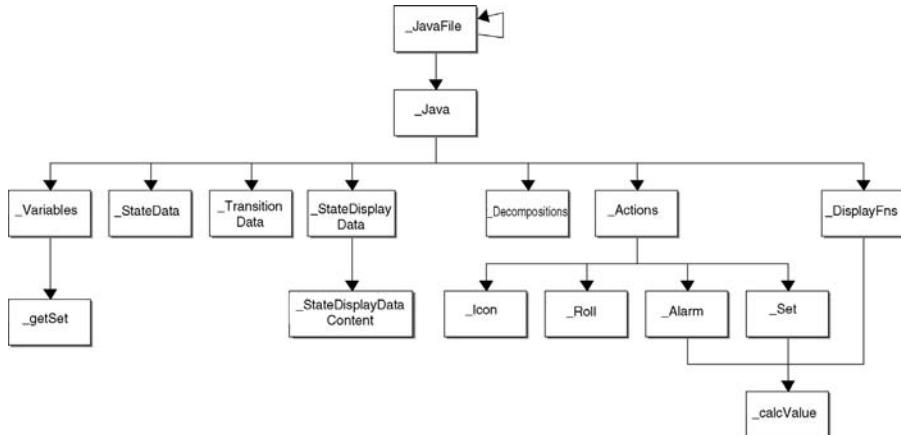


FIGURE 9.11 The watch code generator architecture, part 2.

However, to understand how the code for a logical watch and a watch application is generated, we need to explore the code generator architecture further. The lower level of the architecture (i.e., the subgenerators associated with the Watch Application modeling language) is shown in Fig. 9.11.

The subgenerators “_JavaFile” (which is the same as in Fig. 9.10) and “_Java” take care of the most critical part of the generation process: the generation of the state machine implementations. To support the possibility to invoke a state machine from within another state machine in a hierarchical fashion, a recursive structure was implemented in the “_JavaFile” subgenerator. During the generation, when a reference to a lower-level state machine is encountered, the “_JavaFile” subgenerator will dive to that level and call itself from there.

The task of the “_Java” subgenerator is to generate the final Java implementation for the Logical Watch and Watch Application state machines. An example of such code can be found in Listing 9.4, which shows the implementation of the Stopwatch application.

For a comprehensive understanding of the “_Java” subgenerator output, let us study the generated code line by line. As before, a new concrete Watch Application class is derived from the AbstractWatch Application class (line 1). From here on, the responsibility for the generated code is distributed among the “_Variables,” “_StateData,” “_TransitionData,” “_StateDisplayData,” “_Actions,” and “_DisplayFns” subgenerators.

The “_Variables” and “_getSet” subgenerators are responsible for declaring the identifiers for actions and display functions to be used later within the switch-case structure (lines 3–7). They also define the variables used (lines 9–10) and the implementations of their accessing methods (lines 12–30). A short return back to the “_Java” subgenerator produces lines 32–34, followed by the state (lines 35–38) and state transition definitions (lines 40–45) generated by the “_StateData” and “_TransitionData” subgenerators. The “_StateDisplayData” and “_StateDisplayDataContent” subgenerators then provide the display function

Listing 9.4 The generated code for Stopwatch application.

```
1 public class Stopwatch extends AbstractWatchApplication
2 {
3     static final int a22_3324 = 1;
4     static final int a22_3621 = 2;
5     static final int a22_4857 = 3;
6     static final int d22_4302 = 4;
7     static final int d22_5403 = 5;
8
9     public METime startTime = new METime();
10    public METime stopTime = new METime();
11
12    public METime getstartTime()
13    {
14        return startTime;
15    }
16
17    public void setstartTime(METime t1)
18    {
19        startTime = t1;
20    }
21
22    public METime getstopTime()
23    {
24        return stopTime;
25    }
26
27    public void setstopTime(METime t1)
28    {
29        stopTime = t1;
30    }
31
32    public Stopwatch(Master master)
33    {
34        super(master, "22_1039");
35        addStateOop("Start [Watch]", "22_4743");
36        addStateOop("Running", "22_2650");
37        addStateOop("Stopped", "22_5338");
38        addStateOop("Stop [Watch]", "22_4800");
39
40        addTransition ("Stopped", "Down", a22_3324, "Stopped");
41        addTransition ("Running", "Up", a22_4857, "Stopped");
42        addTransition ("Stopped", "Up", a22_3621, "Running");
43        addTransition ("Stopped", "Mode", 0, "Stop [Watch]");
44        addTransition ("Running", "Mode", 0, "Stop [Watch]");
45        addTransition ("Start [Watch]", "", 0, "Stopped");
46
47        addStateDisplay("Running", -1, METime.SECOND, d22_5403);
48        addStateDisplay("Stopped", -1, METime.SECOND, d22_4302);
49    }
```

```

50
51     public Object perform(int methodId)
52     {
53         switch (methodId)
54         {
55             case a22_3324:
56                 setstopTime(getstartTime() .meMinus(getstartTime()));
57                 return null;
58             case a22_3621:
59                 setstartTime(getsysTime() .meMinus(getstopTime()));
60                 iconOn("stopwatch");
61                 return null;
62             case a22_4857:
63                 setstopTime(getsysTime() .meMinus(getstartTime()));
64                 iconOff("stopwatch");
65                 return null;
66             case d22_4302:
67                 return getstopTime();
68             case d22_5403:
69                 return getsysTime() .meMinus(getstartTime());
70         }
71         return null;
72     }
73 }
```

definitions (lines 47 and 48), while the basic method definition and opening of the switch statement in lines 51–54 again come from the “_Java” subgenerator.

The generation of the code for the actions triggered during the state transitions (lines 55–65) is a good example of how to be creative in integrating the code generator and the modeling language. On the modeling language level, each action is modeled with a relationship type of its own. When the code for them is generated, the “_Actions” subgenerator first provides the master structure for each action definition: a case statement using the unique ID of the Action. It then follows the relationships for each part of the action and executes the subgenerator bearing the same name as the current action relationship (either “_Icon,” “_Roll,” “_Alarm,” or “_Set”). The subgenerator produces an appropriate line of code for this part of the action, for example turning an icon on in line 60. This implementation not only reduces the generator complexity but also provides a flexible way to extend the watch modeling language later if new kinds of actions are needed.

Finally, the “_DisplayFns” and “_calcValue” subgenerators produce the calculations required by the display functions (lines 66–69). The “_calcValue” subgenerator—which is also used by the “_Alarm” and “_Set” subgenerators—provides the basic template for all arithmetic operations within the code generator.

The generation of the simple middle-level Watch Applications such as TASTW proceeds in the same way. As the middle-level models are simpler than the applications they contain—for example, they have no actions—the resulting code is also simpler. In order to support the references to the lower-level state machines (i.e.,

to the Watch Applications of which the middle-level model is composed), the definitions of these decomposition structures must be generated. This is taken care of by the “_Decompositions” subgenerator.

9.5.2 C Generator for Watch Models

Since the choice of Java and the highly specific state machine framework of the Java generator were not a familiar approach to most embedded developers, we later built a C generator. The generator, shown in Listing 9.5, is significantly shorter than that for Java and follows a completely different approach. The application’s States and Buttons are turned into enums, and the state machine is generated as two-level nested switch-case statements. The outer level has a case for each state the application is in, and the inner level for that state specifies the actions and transition for each button that may be pressed in that state.

Listing 9.5 The C generator for Watch Applications.

```

subreport '_C.Enums' run
'int state = Start;
int button = None; /* pseudo-button for buttonless transitions */
'
subreport '_C_RunWatch' run;
'void handleEvent()
{
    int oldState = state;
    switch (state)
    {
        '
foreach .(State | Start)
{   '       case ' id ':' newline
    '           switch (button)' newline
    '               :' newline
        do ~From>Transition;
    {   '                   case '
        if ~Event; then
            do ~Event.Button { id }
        else
            'None'
        endif
        ':' newline
        do ~Action.Action
        {   do ~ActionBody>()
            {   '                               subreport '_C_' type run
            }
        }
        do ~To.(State | Stop)
        {   if not oid = oid;2 then
            '                           state = ' id ';' newline
            endif
        }
    }
}

```

```

        '           break;'  newline
    }
'
        default:'  newline
        '           break;'  newline
    '}  newline
}
'
default:
    break;
}
button = None; /* follow buttonless transitions */
if (oldState != state) handleEvent();
}
'
```

This generator produces the familiar nested switch-cases used in much embedded software. The results for Stopwatch are shown in Listing 9.6.

Listing 9.6 The generated C code for the Stopwatch application.

```

typedef enum { Start, Running, Stopped, Stop } States;
typedef enum { None, Down, Mode, Up } Buttons;
int state = Start;
int button = None; /* pseudo-button for buttonless transitions */
void runWatch()
{
    while (state != Stop)
    {
        handleEvent();
        button = getButton(); /* waits for and returns next
button press */
    }
}
void handleEvent()
{
    int oldState = state;
    switch (state)
    {
        case Start:
            switch (button)
            {
                case None:
                    state = Stopped;
                    break;
                default:
                    break;
            }
        case Running:
            switch (button)
```

```

    {

        case Up:
            stopTime = sysTime - startTime;
            icon (0, stopwatch);
            state = Stopped;
            break;
        case Mode:
            state = Stop;
            break;
        default:
            break;
    }

    case Stopped:
        switch (button)
        {
            case Mode:
                state = Stop;
                break;
            case Down:
                stopTime = startTime - startTime;
                break;
            case Up:
                startTime = sysTime - stopTime;
                icon (1, stopwatch);
                state = Running;
                break;
            default:
                break;
        }
        default:
            break;
    }

    button = None; /* follow transitions that do not require
buttons */
    if (oldState != state) handleEvent();
}

```

9.6 THE DOMAIN FRAMEWORK

From the point of view of the DSM environment, the domain framework consists of everything below the code generator: the hardware, operating system, programming languages and software tools, libraries, and any additional components or code on top of these. However, in order to understand the set of requirements for the framework to meet the needs of a complete DSM environment, we have to separate the domain-specific parts of the framework from the general platform-related parts.

In many cases, the demarcation between the platform and the domain-specific part of the framework remains unclear. For example, the version of Java in which the watch example was originally written did not contain any useful service to handle timed events such as alarms. Thus, we implemented such a service ourselves as part of our domain framework. The more recent versions of Java, however, do provide a similar mechanism, meaning that it could be part of the platform if the watch implementation only needed to support more recent versions of Java.

Without any deep theoretical discussion about what is the border between framework and platform, we shall use the following definitions here: The platform is considered to include the hardware, operating system (Windows or Linux here), Java programming language with AWT classes, and environment to test our generated code (e.g., a web browser with Java runtime). The domain framework consists of any additional components or code that is required to support code generation on top of this platform. The architecture of the watch domain framework—as defined in this way—is shown in Fig. 9.12 (solid line arrows indicate a specialization relationship, while dotted line arrows indicate inclusion relationships between the elements).

The domain architecture of the watch example consists of three levels. On the lowest level, we have those Java classes that are needed to interface with the target platform. The middle level is the core of the framework, providing the basic building blocks for watch models in the form of abstract superclass “templates.” The top level provides the interface of the framework with the models by defining the expected code generation output, which complies with the code and the templates provided by the other levels.

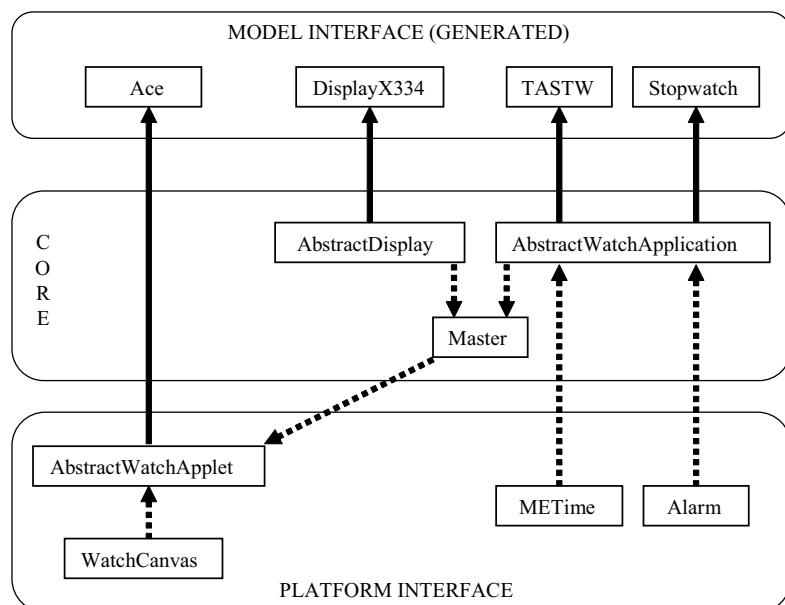


FIGURE 9.12 The watch domain framework

There are two kinds of classes on the lowest level of our framework. METime and Alarm were implemented to raise the level of abstraction on the code level by hiding platform complexity. For example, the implementation of alarm services utilizes a fairly complex thread-based Java implementation. To hide this complexity, class Alarm implements a simple service interface for setting and stopping alarms, and all references from the code generator to alarms are defined using this interface. Similarly, METime makes up for the shortcomings of the implementation of date and time functions in the Java version used. During the code generation, when we need to set an alarm or apply an arithmetic operation on a time unit, the code generator produces a simple dispatch call to the services provided by these two classes.

The other classes on the lowest level, AbstractWatchApplet and WatchCanvas, provide us with an important mechanism that insulates the watch architecture from platform-dependent user interface issues. For each supported target platform, there is an individual version of both of these classes, and it is their responsibility to ensure that there is only one kind of target template the code generator needs to interface with. Initially, there was only one target platform: Java applets in a web browser.

On top of the platform interface and utilizing its services is the core of the framework. The core is responsible for implementing the counterparts for the logical structures presented by the models. The abstract definitions of watch applications, logical watches, and displays can be found here (the classes AbstractWatchApplication and AbstractDisplay). When the code generator encounters one of these elements in the models, it creates a concrete subclass of the corresponding abstract class.

Unlike the platform interface or the core levels, the model interface level no longer includes any predefined classes. Instead, it is more like a set of rules or an API of what kind of generator output is expected when concrete versions of AbstractWatchApplet, AbstractWatchApplication, or AbstractDisplay are created.

9.7 MAIN RESULTS

Comparing the Watch models to the tangle of code that is normally found when similar embedded systems are hand coded, it is clear that the DSM solution helped build systems better here than is normal in the industry. However, looking at the generated applications, it is equally clear that the code produced is shorter and in some ways simpler than would normally be written by hand. How much then is the improvement due to DSM actually only due to the greater attention spent on developing a good framework?

To gain some insight into this, we devised an experiment. Subjects extended the Stopwatch application to add lap time functionality, first by DSM with code generation, and again manually by editing the original Stopwatch Java code in the same architecture. While the sample size was too small to be statistically significant, the results in Table 9.1 may still be interesting.

The necessary changes to add a lap time function were roughly eight lines of Java code or eight objects in the model. For a senior developer, the productivity for

TABLE 9.1 Productivity Improvements of DSM Modeling versus Manual Coding

	Modeling		Coding		Modeling:Coding Productivity Ratio
	Time (s)	Functions/hr	Time (s)	Functions/hr	
Senior developer	38	94.7	200	18.0	5.2 : 1
Junior developer	160	22.5	639	5.6	4.0 : 1
Total		117.2		23.6	5.0 : 1

modeling was over five times that for coding. In the case of a junior developer, the difference was only four times, but for the combination of both developers the productivity for modeling was five times that for coding.

As is often found, the senior developer was three to four times more productive than the junior developer, and this difference was evident whether modeling or coding. However, the productivity of a junior developer modeling was greater than that of a senior developer coding—a tantalizing prospect for project managers. Imagine all of your developers being 25% more productive than your top developer is currently!

To return to our original question, it is now clear that the main benefit of DSM is in the difference between modeling and coding. While building a good framework to support DSM is useful, hand coding on top of even a good framework is still four to five times slower than DSM.

9.7.1 Extending the DSM Solution to New Platforms

Initially, there was only one target platform for the Watch applications: Java applets in a web browser. With the advent of Java applications on mobile phones another possible platform appeared. While it is still Java, the new platform, MIDP, differed radically in its user interface possibilities. Text fields to show time digit pairs were replaced by a bitmap display, and mouse and keyboard input was replaced by simple soft keys and cursor keys. In addition, the main Applet class was replaced by a Midlet class, and MIDP Java did not support reflection, which we had used in the code generated for Actions and Display Functions. Finally, the build process for compiling and packaging a MIDP application required a number of extra steps and several new files.

In spite of these differences, extending the Watch DSM solution to support this new platform was possible mainly by simply building versions of the AbstractWatchApplet and WatchCanvas classes for the phone MIDP framework. Only minor changes were necessary to the code generation, to work in the more restricted MIDP environment. No changes were necessary to the modeling language or models. Altogether, the changes took four man-days: 2 for the MIDP framework code, 0.5 for the MIDP build script, 1 for refactoring existing framework code, and 0.5 for adapting the code generation.

Later, the generators and framework were further extended to take advantage of the MetaEdit+ API to provide visual tracing of model execution. As the watch application ran, it would make WebServices calls back to MetaEdit+ when control

passed to a new state, and MetaEdit+ would highlight that state. Again, these changes only affected the framework and generators.

The DSM solution thus insulated Secio from changes in the implementation environment, offering good support for a family of products across a family of platforms. The current Watch models are capable of generating code for each of the three platforms and on a variety of operating systems. Without DSM, there would probably be no hope of maintaining one code base for all platforms.

9.8 SUMMARY

Designing and implementing the first working version of the Watch DSM language with one complete watch model took eight man-days for a team of two developers. Neither developer had prior experience of Java programming or of building watch software, and there were, of course, no pre-existing in-house watch components. It took 5 days to develop the Java framework, 2 days for the modeling language, and 1 day for the code generator. Another day or two was then spent adding support for multiple Watch Families and creating the full set of example models. These times include design, implementation, testing, and basic documentation.

Calculating the return on investment for these times is not possible without more information about Secio: what their current code framework was like, how many watch models they wanted to produce, and so on. An estimate for the latter can be found from the fact that for a real digital watch manufacturer, Casio, their current portfolio for the United Kingdom alone contains over 220 different watches (not including variants differing only in color or materials).

PART IV

CREATING DSM SOLUTIONS

In this part, we teach you how to create the various parts of a DSM solution (Chapters 10–12), discuss the processes and tools for creating and using the DSM solution (Chapters 13–15), and wrap up with a summary and conclusions in Chapter 16. The examples from Part III are often used in explaining the principles discussed here, so you would be advised to at least skim those examples first.

Creating the various parts of a DSM solution may be the responsibility of one person, or then the task may be split over two or more people. Chapters 11 and 12 on the generator and domain framework, and to some extent Chapters 14 and 15 on tools and usage of DSM, will make most sense to experienced programmers. Chapters 10 and 13 on DSM language creation and processes may interest those in more of an architect or project management role.

CHAPTER 10

DSM LANGUAGE DEFINITION

Defining a language for development work is usually thought to be a difficult task. This may certainly be true when building a language for everyone and for every system. How could you create a modeling language if you did not know its intended purpose, users and applications to be modeled?

The language definition task becomes considerably easier when the language need only work for one problem domain in one company. You can focus on a restricted domain, the area of your interest. This is often the same area where you have already been working. There are thus most likely some fundamental concepts already available and in use, even if they are only used in spoken language, product presentation slides, high-level product designs, or requirements. Actually, most people already have a domain-specific vocabulary in use and that vocabulary exists with good reason: it is relevant when discussing such systems. Usually, there is no need to introduce a whole new domain-specific language as it is already in use, albeit implicitly and partially. The person who specifies a Domain-Specific Modeling (DSM) language identifies the modeling concepts in detail and formalizes them by creating a metamodel.

10.1 INTRODUCTION AND OBJECTIVES

We describe in this chapter how to define one or several integrated modeling languages for a specific problem domain: from initial language concept identification to testing and maintenance. Throughout the chapter, we use the examples from Part III to

illustrate language definition in detail. First, in Section 10.2 we look at how to find and define the modeling concepts and their characteristics. This leads us to introduce language specification languages: Section 10.3 describes metamodeling as a technique for language definition and formalization. Having a formalization mechanism, our next concern in Section 10.4 is identifying and specifying various connections and rules among the modeling concepts. We then extend the scope from one language to language integration: having multiple languages that focus on different aspects of the problem domain. In Section 10.6, we give guidelines for defining concrete syntax: the notation of modeling languages. Metamodels provide an extra benefit for language definition as they can be instantiated to prototype and test the language. Section 10.7 gives guidelines for language testing, both before and after the language has been used. Finally, a DSM language almost always evolves along with the domain and our understanding of it. Section 10.8 describes language refinement and space and gives guidelines for maintaining the language already in use.

10.2 IDENTIFYING AND DEFINING MODELING CONCEPTS

Language defines the boundary to our world: it sets what we can describe and also what we can't (Wittgenstein, 1922). For DSM the latter is crucial, as narrowing down the design space makes it possible to have generators that target full code generation. If it is not possible to narrow down the scope, then most likely the modeling language is unusable for generating the required code or other artifacts. In other words, we have created another general-purpose modeling language. The first rule of language definition is, therefore, to start with the better understood parts of the domain and extend the modeling language gradually to cover the full domain—possibly integrating with other languages. It is worth remembering that DSM languages should never be considered to permanently restrict our view of the world because they can be changed when needed. If the domain changes or our understanding of the domain changes, we should be able to change the language too.

The language creator needs to find concepts and abstractions that are relevant for a given development situation. Generally, the main domain concepts map to the main modeling concepts, while others will be captured as properties, connections, submodels, or links to models in other languages. While defining the languages, the balance with the generator, domain framework, and available components also needs to be decided. Next, we describe approaches to identify modeling concepts and map them to some of the models of computation behind every modeling language. Although individual modeling concepts are most often applied with one language only, models are often integrated and reused. The language creator can also identify connections between modeling concepts and inspect language use situations where existing model elements could usefully be reused. Unlike in a general-purpose language, a domain-specific language can enforce reuse within the domain and address model integration where different aspects of the domain are combined into one language or several integrated languages. In the latter case, the parts of the metamodel are related by reusing the same elements or linking elements between different languages.

The identification of the modeling concepts is normally highly iterative and may need to be repeated several times. You definitely can't identify all the concepts immediately and therefore it helps to define languages in stages and try them out by making example models. The number of iterations depends on many factors: the size of the domain, its stability, availability of domain knowledge, and the experience of the language definers.

10.2.1 Where to Find Modeling Concepts

Although it may be tempting to use concepts that originate from the code as a starting point for language definition, higher abstraction, and thus better productivity can be achieved if the modeling concepts are from the nonimplementation concepts. A study analyzing DSM language creation approaches in over 20 industry cases (Tolvanen and Kelly, 2005) shows that concepts coming from the look and feel of the system built, the variability space, and the domain expert's concepts lead to higher abstraction than those originating from generation output. In other words, the language definer should focus more on the problem domain than its code (the solution domain in Jackson, 1995). Naturally, while aspiring toward a higher level of abstraction, we need to keep in mind that ultimately we need to have the ability to generate code from models too. Starting from domain concepts is always better, though: adding coding concepts later on is usually easier than vice versa.

Problem domain concepts also have other characteristics that make them good candidates for the language:

- They are usually already known and used. They can be found while communicating with customers but also among development teams. It is not necessary for the languages to introduce new concepts: they can build on existing ones when possible. Using concepts that are already in use is also relevant for the acceptance of DSM. People are comfortable with the concepts, and they don't need to invest in learning new languages. In addition to concepts, it is often a good approach to base the notation of the language on problem domain symbols already in use (see Section 10.6).
- Problem domain concepts usually have established semantics in place. The language definer can then base the language on existing definitions and, if they are not fully available, can ask domain experts. For example, in the domain of mobile User Interface (UI) applications, developers know the concept of the soft key: two or more keys whose functions change based on the application context. Similarly, in the banking domain, everybody knows what a bank day means. If the semantics of a particular concept cannot be defined, most likely it is not a relevant concept for a language either.
- They establish a natural mapping to the actual problem being addressed by DSM. This makes model creation easier in the first place and enables modeling operations like reusing model elements at the domain level. A close mapping to the domain also makes models easier to read, understand, remember, check, and communicate with.

Not all problem domain concepts are suitable language concepts. Concepts that every feature, application, or product always has should not normally be part of the language. Why would we want to model something that is always the same? Instead they should be provided via components and code libraries or produced by the generator. In modeling languages, only those concepts that allow describing variation should be considered. Perhaps it is easiest to consider variation as a property of the language element. For example, buttons in a digital wristwatch (Chapter 9) need to be differentiated and thus the button concept has a property to specify its name. Variation can also be expressed by connecting language elements with each other. For instance, in the watch case an action does not have any properties of its own, but the action modeling concept is needed to specify how time variables are modified, alarms used, and icons shown and hidden.

Concepts that can be identified from the combination of existing concepts should also usually be excluded. For example, in the mobile phone case (Chapter 8), there is no explicit button concept that the user presses to indicate navigation paths in the application. Unlike in the watch example, where the number of buttons and their labels can vary, in the mobile phone case they are implicitly presented with the different navigation relationships: one for default selection, one for cancelling, and one for accessing menus. As alternative navigation paths need to be specified anyway, a special button concept in the language would be unnecessary. Keeping the language smaller is better, as then it becomes easier to learn and use.

Sources for Modeling Concepts Identification of the relevant concepts in the language is largely dependent on creative insights and experience in the domain. It helps if one has been involved in making similar kinds of applications in the past. Whenever possible, you should consult people more experienced in the domain, such as problem domain experts, architects, and lead developers. Their insights and opinions are often the main source for creating the DSM solution. You can interview them, observe their work routines, and use other mechanisms that reveal problem domain characteristics. Keep in mind, however, that the person specifying the language into a metamodel and implementing the language into a tool does not necessarily need to be experienced in the domain.

The candidate concepts for the modeling language can be found in very different sources. We can identify some of them from the jargon and vocabulary in use. Frequently used concepts exist with good reason: people find them relevant and concise when discussing a product and its features. The vocabulary often provides the best starting point, as it mostly uses natural concepts: people do not think of solutions immediately in coding terms. Starting from the existing vocabulary also means that there is no need to introduce a new, unfamiliar set of terms or map existing concepts and their semantics to those provided by some external languages. What does the Unified Modeling Language (UML) (or SysML, IDEF, BPMN, etc.) know, for instance, about banking applications, pacemakers, or applications you are developing? It is far better to use the concepts of your domain in a language than map them to external concepts and related semantics.

Other typical sources for finding candidate concepts from the domain include the following:

- **Architecture:** A description of the architecture is often a good source since the architecture usually operates on domain concepts. This is especially true for embedded systems. Usually, the architecture is also the most stable part and reveals core abstractions about the application elements and their behavior.
- **Existing products,** applications, features, and related manuals: These capture the structure, behavior, and semantics, but unfortunately their complete analysis is not always practical as it may be too time consuming to do an exhaustive analysis of all the products. It is more convenient to select a representative set of applications for more detailed inspection. Naturally, those kinds of applications should be selected that have functionality closest to the newly planned ones. It can also be that the applications and their features have not yet been developed. Then, you may look into similar kinds of applications in the same domain or inspect earlier generations or versions.
- **Available specifications:** By analyzing existing descriptions of the features, applications, or products, you can understand the structure of the domain and translate it to a conceptual schema. Requirement documents are especially good to raising the level of abstraction as they usually focus on the problem domain rather than the implementation. Requirements also map the customer terminology, making wider DSM use possible: customers can then better read and check the models. The specifications don't need to be particularly formal or be based on some known specification language. Actually, models made without any restrictions lead modelers to capture the domain in a manner they see as most effective. Inspecting how they want to approach the problem and using alternative views immediately reveal the approach they see as most "natural." In this sense, you should never underestimate presentation slides, drawing tools, or whiteboards. They are great for making specifications that map closer to the problem domain—but often poor for any automation like reuse, checking, sharing, analysis, generation, and so on.
- **Patterns:** If a company has an established collection of patterns, they may describe domain concepts or reveal common structures within the problem domain. Domain-specific patterns may also be available elsewhere that are representative of the problem domain under examination. Here, pattern matching can be applied to check the ingredients of a pattern and pattern recognition to detect underlying patterns.
- **Target environment** and its interfaces: Existing libraries, component frameworks, and interfaces were often made to raise the level of abstraction—but with limited guidance and automation for their use. Inspecting them shows how the applications and features are to be built and which services are already available.
- **Code:** Abstractions can be identified with a bottom-up approach: examine existing applications and features and generalize from them. This is not limited to finding abstractions for the language, but also helps identify the structures of

generated code (see Chapter 11 for details). In particular, the advice of experienced programmers and coding guidelines need to be acknowledged: other developers follow these as they relate closely to established manual practices. It is not recommended, however, to follow only a bottom-up approach as you also need to predict future possibilities. This is important since after automating the development tasks with DSM you can develop features or whole applications that you could not develop earlier because of the higher costs or longer development times.

Sometimes, the above-mentioned sources may not be available: the domain is new, there are no past experiences available, or the specifications were not made available in the first place. The domain knowledge is then scattered in the organization and found only in individuals' memory. Then, you should create concrete examples of alternative ways to specify the problem—not just one but multiple examples with prototypes. The sample applications are not only useful for identifying abstractions but can also be used later to test the DSM solution.

10.2.2 Useful Categories of Modeling Concept Sources

Domain concepts often resemble each other since many of them originate from the same source. Finding one good candidate often leads to finding other similar ones. The sources we have found useful for finding language concepts can be categorized as follows:

- Physical product structure
- Look and feel of the system
- Variability space
- Domain (expert) concepts
- Generation output

In practice, you often need to look at more than one category, but each offers a clear strategy with which start. We will discuss each category in more detail and provide some examples.

Physical Structure Physical structures usually provide a good starting point for the language definition, as they are relatively easy to identify and clearly restricted. For example, a language for developing automation systems for a paper factory or power plant could be based on problem domain concepts like valves, motors, sensors, and controls. A valve will have attributes like size and direction and rules on how it may relate to motors and sensors. By analyzing the physical structure of the product, other candidate concepts could be identified. Although a language may specify only software, many aspects of the software are closely connected to hardware. For example, the language concept valve needs to represent not the actual device or hardware but its controller or interface.

Typical problem domain areas where languages are partly based on physical structure can be found, for example, from communication systems, network-related

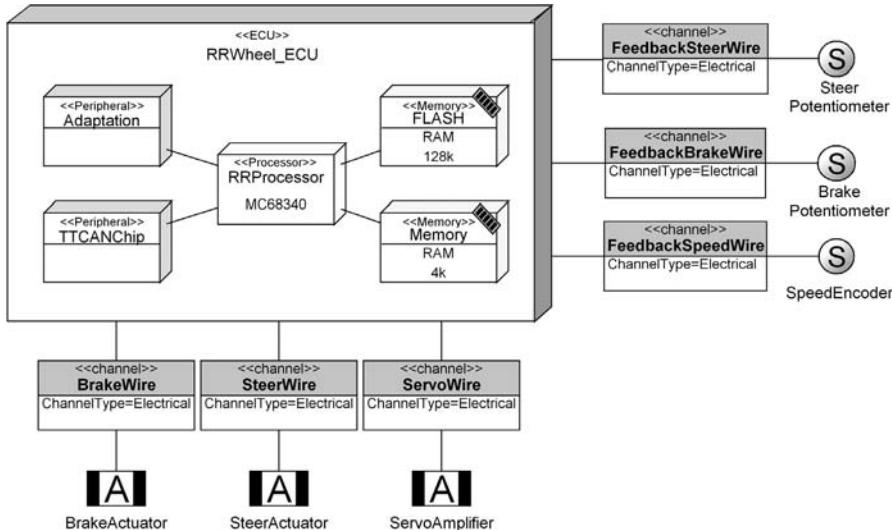


FIGURE 10.1 Physical structure-based language for modeling the hardware architecture in automobiles (based on EAST-ADL)

software, industrial automation, railway control, power and electricity control, home automation, logistic systems, and hardware architecture. Also, the design of distributed systems usually needs a language that is at least partially based on the concepts found from physical structures.

Figure 10.1 illustrates a DSM language that uses physical entities as modeling concepts. In EAST-ADL, one of its five modeling languages focuses on describing the hardware architecture in automobiles. Shown here is the architecture for electronic control units (ECU) with processors and memories connected through a CAN bus. The language provides several alternatives for bus types and constraints on the application of buses in the described hardware architecture. These modeling concepts can be identified by analyzing all the possible hardware components a car may have.

Languages based on physical structures usually focus on static declarative modes but may also include behavioral elements. Designs in such a language usually provide configuration data for the rest of the generation process and are usually linked to other models in order to achieve more comprehensive code generation.

Look and Feel of the System Products whose design can be understood by seeing, touching, or hearing (HMI/MMI systems) often lead to languages that apply end-user product concepts as modeling concepts. A language for defining voice menus can include concepts like “menu,” “prompt,” and “voice entry” as well as guidelines on how these may be linked to achieve user navigation. This type of language is quite easy to define and test, as it has “visible” counterparts in the actual product.

The language developer can thus look for language concepts by analyzing how the user of the product uses it. The product manuals and user guides that explain features, give guidelines for applying functions, or help in navigating the system all provide

good sources for candidate modeling concepts. These kinds of languages are typical when targeting application development for Personal Digital Assistants (PDAs), mobile phones, diving instruments, wrist computers, and other consumer electronics as well as automotive infotainment and navigation systems. Perhaps one of the biggest areas in software development for these languages is the GUI navigation of typical administrative and enterprise applications.

Figure 10.2 shows an application design in a language that is clearly based on look and feel. The language targets the human-machine interface of an automotive infotainment system, specifying displays and their content together with the behavioral logic of the applications. If you are familiar with some infotainment applications, like navigation or setting preferences for traffic announcements, then you will most likely understand what the application does just by looking at the model.

Look and feel are represented directly in the language by using as modeling concepts the actual displays, user interface widgets, and user controls provided by the system. Having identified modeling concepts for one kind of display, widget type, or navigation path, the other modeling concepts can be defined similarly. It is relevant to note that, although the underlying framework and way to map these modeling concepts to individual code may differ among the concepts, it does not matter in the language creation phase. Basing a language purely on look and feel is not normally enough: You also need to identify and map non-GUI concepts to other concepts in the language or find a mapping to other languages.

Variability Space One efficient approach to start defining a language is to focus on variability: you define the language so that variability options are captured by the modeling concepts, and the modeler's role is to concentrate on the areas that differ

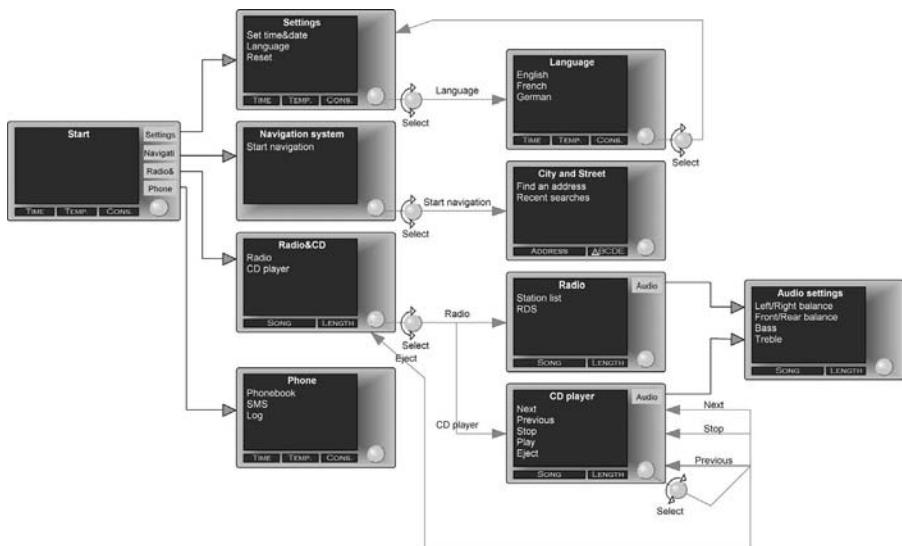


FIGURE 10.2 Language based on the look and feel of a car infotainment system

between different products or features. Long term success in defining a language here depends largely on your capability to predict what kind of variation space is needed in future products. Note that although this kind of language can most often be found in product-line development, it does not necessitate having multiple products: Variation also exists among features of a single product.

Language definition based on variability comes down to conducting a thorough domain analysis (Weiss and Lai, 1999): Identifying which abstractions are the same for all applications and which are different. Static variation is usually easy to cope with—developers have been making parameter tables and wizards to choose among alternatives for decades. Things get more complicated when the parameter choices depend on other parameter choices and here feature modeling (Kang et al., 1990) is useful and often applied along with configuration tools. Note that variation at the code level is not so relevant here since a generator may produce the required code to one or more places in various files to implement the variability. Parameter and feature choice approaches usually break down if we also want to tackle variability that is of a dynamic nature, or if we want to create new features and functionality inside the current variation space. This is a common situation since companies have hardly ever made all the product features to be selected.

Figure 10.3 illustrates the spectrum of variability. Wizards and feature-based configuration focus on making choices among known decisions and features. Domain-specific languages do not set choices explicitly but give a practically infinite space to set variation. You do not know all variants, as they can be numerous. There are as many variants as there are ways to instantiate the metamodel.

In the simplest case of modeling language design, the variability can be represented solely as properties of modeling objects—something similar to parameter

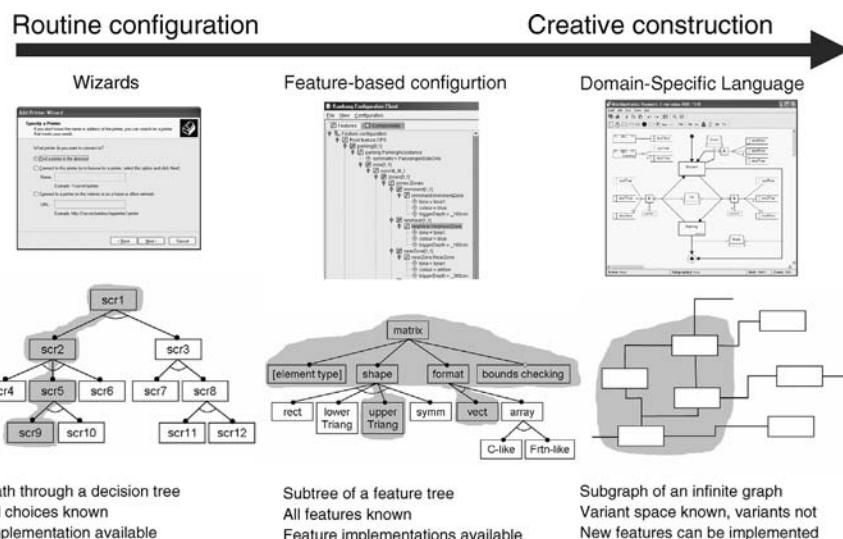


FIGURE 10.3 Spectrum of variability (modified from Czarnecki, 2004)

lists. The possible values a given property may have can be defined as a list containing the predefined legal values of that variability point. In a little more complex cases, and similar to feature models, variability can be expressed via connections between modeling elements, their linkages to submodels, and so on. Language concepts may also be rich enough to allow creating totally new implementations in the variation space set by the language. In this way, all the variants do not need to be made yet, or even thought about, since the creation of new variants can be done by creating totally new specifications. Among the examples in Part III, the representative case for this is the digital wristwatch language (see Chapter 9).

Domain Experts' Concepts Domain experts can also be test engineers, commissioning, configuration, packaging and deployment engineers, or service creators. Because they are usually not programmers, a modeling language for them needs to raise the level of abstraction beyond programming concepts. Languages that are based on domain experts' concepts are relatively easy to define because for an expert to exist, the domain must already have established semantics. You can derive many of the modeling concepts directly from the domain model. The same holds true for some of the constraints.

Figure 10.4 shows a language based on domain experts' concepts (see Chapter 6 for details). For this particular language, the modeling concepts are related to financial and insurance products. Concepts like "Risk," "Bonus," and "Damage" capture the relevant facts about insurances. Using this language an insurance expert, and thus a nonprogrammer, draws models to define different insurance products. Generators take care of transforming these designs into code for a web application for analyzing and comparing insurance products. In this way, the expert programmer can build the mapping from the language to the code once, and neither he nor the insurance experts need to know the intricacies of the others' area of expertise. The higher abstraction in models using domain experts' concepts also means that the generated output can be easily changed to some other implementation language.

Generated Output The fifth and last category of concept sources is the generation target of the language: the concepts and structures we see in the code to be generated are mapped directly into the modeling language. One of the most typical cases here is defining a metamodel based on the schema of the XML to be generated: Each tag type refers to a concept in a modeling language. While these languages are easy to build, their ability to increase productivity and quality is questionable. There is a danger of creating languages like class diagrams: presenting a class as a rectangle that maps one to one to a line in a file. This kind of language may still be valuable when the generated output is already in a domain-specific language, like a particular XML format. The XML schema will provide you with a wealth of information for identifying the modeling concepts and constraints. To follow the XML metaphor, designs can be considered valid and well-formed right at the modeling stage. Graphical models can also help overcome many of the limitations of XML.

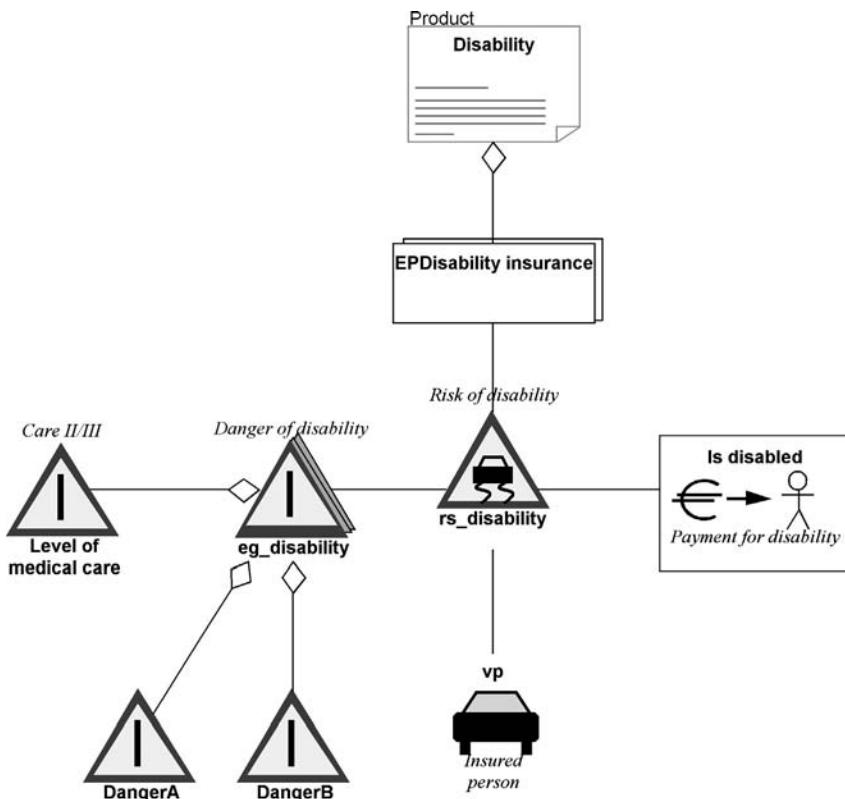


FIGURE 10.4 Modeling financial and insurance products for a J2EE web application with a DSM language based on domain expert concepts

An example of this kind of DSM language is the Call Processing Language (CPL), which is used to describe and control Internet telephony services (see Fig. 10.5 and Chapter 5 for a detailed description). The modeling concepts include “proxy,” “location,” and “signaling actions,” essential for specifying IP telephony servers. These same concepts are already defined as elements in the XML schema, and the property values of the modeling concepts are attributes of the XML elements. Having generators produce the configuration in XML gives significant and obvious productivity and quality improvements. With the modeling language, it is far more difficult to design services that have errors: something that is all too easy in handwritten CPL/XML.

10.2.3 Choosing Computational Models

Domain concepts do not exist independently: they relate to each other. These connections can be illustrated in the models too with supporting modeling concepts.

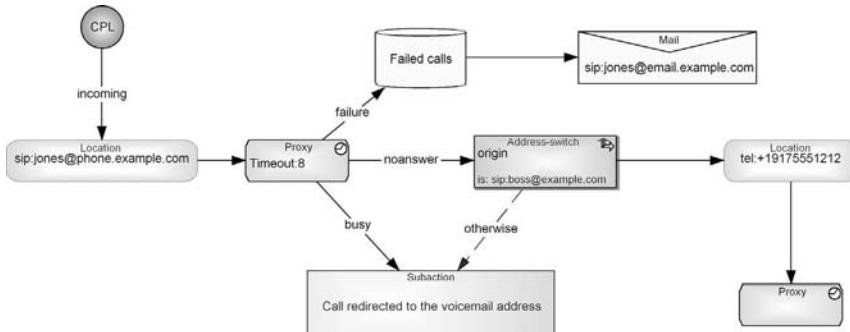


FIGURE 10.5 Modeling call processing in IP telephony, a DSM language based on its XML generation target

The way domain concepts should be connected leads to finding a suitable model of computation (MOC, see Section 4.2.2). Usually, the main approach can be identified relatively easily: should we focus on specifying static structures, behavior, or both. When inspecting details of the domain there are, however, big differences, and it is the details that matter when using models for code generation.

The insurance case described in Chapter 6 is the only example in this book of modeling just static structures. The rest of the examples also cover behavior, and some address only behavior. The watch example in Chapter 9 is based on using two different languages and models of computation. First, the physical structures of individual watch products are specified by defining their elements: displays, icons, buttons, and time units. The main part of the watch products is, however, the functionality of different applications, like how time, alarm, or stopwatch applications work. For this purpose, a state machine is used as a foundation for the second language. It is then extended with domain-specific concepts for specifying state-based and event-driven watch products. Both languages are further integrated by using partly the same concepts so that model data specified in one language can be shared with another. For some domains the static parts, like data elements, can have such minor roles that creating a separate language for just a few aspects is unnecessary. It is best to extend the behavioral language to allow specifying static structures. This was the case with the mobile phone application (Chapter 8).

You should not choose the MOC solely based on the domain. The expected generated code also influences the modeling language choice. For example, if most behavior is provided by the underlying framework, we can produce just structure and interface data. This suggests the use of languages addressing static structures and is often the easiest way to achieve model-based code generation. Although we can generate functional and behavioral code from models describing static structures (like schema creation for a database), the possibilities of specifying behavior, logic, dynamics, and interaction are easier with languages addressing behavior, such as various flow diagrams, state models, and interaction diagrams.

Library of MOCs The easiest way to start defining a modeling language is to base it on the computational model of an existing language—which already has some proven structures. Tools may help here by offering a library of basic metamodels to start with. You can then copy ideas, or sometimes even concrete elements of the metamodel, into your language and extend them with domain-specific concepts. Such concepts can be new modeling objects, their relationships, properties, bindings, constraints, model hierarchies, reuse rules, and so on. You have here numerous possibilities. Consider, for example, a relationship between model elements. It can be specified by its direction, multiplicity, n-ary, parallelism, or cyclic structures. Each may have further specification of details, like the maximum and minimum values for multiplicity; n-ary relationships can be further specified by the number of different, possibly optional, roles they have; cyclic relationships can further be defined to differentiate between direct and indirect cyclic structure; and so on. Communication flows can further be seen to be synchronous or asynchronous and have different policies, such as balking, timeout, or blocking. It is the numerous small extensions to the basic models of computation that make a modeling language domain-specific.

Multiple Languages and MOCs A single language and MOC may not be enough to carry out specification work. The domain can have multiple aspects each requiring separate views, there can be different roles among modelers or different levels of detail on which to focus, or putting everything into one language is simply not possible. We have not found strict rules on when to add new languages rather than continue adding new concepts to current languages. For example, in one telecom case, language developers had defined just two languages for a relatively large domain: one for static UI and another for the rest. The second language showed multiple aspects, such as data access, process, real time, and events, all in the same model.

When deciding on the number and type of languages, it is best to test the ideas early by making test models: instantiating the metamodel. You should note that new languages also mean that models need to be kept correctly integrated. Keeping everything in a few languages makes modeling work and consistency checking simpler when compared to spreading the specifications over multiple different kinds of languages or views. UML is an extreme case: with 13 separate diagrammatic languages, each having partly different modeling concepts, it becomes difficult, if not impossible, to keep specifications in synch. In DSM, the integration of the languages can be built in via the metamodel. For the telecom company, the decision to divide the domain into two isolated the UI changes from the rest. Even so, models made with the different languages had some integration and linkages. We will discuss language integration in more detail in Section 10.5.

Starting Language Definition from Scratch or by Modifying Existing Languages Although you may use a basic computational model as a starting point, it does not mean that you need to define languages by modifying already existing metamodels. Quite often defining the language from scratch—and naturally keeping in mind the basic models of computation—is simpler and faster to do. You don't need to first learn concepts and semantics defined elsewhere, and you end up

with a simpler language definition. Also, you know the language better as you defined it. This becomes important later when modifying the language, and also because you don't need to consider changes made by others.

For example, all the languages in Part III were defined from scratch rather than by using existing metamodels or their parts. The only exception here is the mobile phone case, in which the modeling language used first for the Python framework was used for generating C++ for another case. It was, however, natural to reuse the already defined metamodel since the domain was practically the same. Modifying an available language can still be a good approach if the domain addressed is closely related to an existing language and changes to this language are minor. The Meta Object Facility (MOF) case in Chapter 6 resembles this situation, but using all the UML class diagram concepts and then choosing which of them are relevant during individual modeling situations would be too costly when compared to creating the language from scratch. Also, much of the class diagram metamodel (template classes, associate classes, etc.) was unnecessary and would introduce extra complexity for modelers. Often the limited capabilities of tools for creating new metamodels and modifying existing languages also partly dictate the language selection and modification.

10.2.4 Defining Modeling Concepts

Having identified the abstraction for the specification work along with some suitable models of computation, you should start mapping domain concepts to modeling concepts. While doing so, you balance between having a dedicated modeling concept and leaving decisions to the modeler. So the question you often need to answer is: should this particular concept be an instance value or a type? A string entered by a modeler or recognized by the language?

Next let's consider an example: how should icons in a digital wristwatch (Chapter 9) be handled in the language? Should it be a particular kind of instance value a modeler enters into the model or a concept of a modeling language? The choices could be as follows:

1. Apply a certain naming convention to identify that this model element is an icon. The modeling concept used to specify icons is thus more general since it can also be used to specify elements other than icons.
2. Use some built-in language extension mechanism. In UML, this could be done by giving a stereotype <<icon>> to the model element.
3. Have a dedicated Icon concept in the language. This Icon concept can then have the properties and other constraints relevant just for icons. For example, if the possible set of icons is fixed, the Icon concept can have a list of legal values from which to choose. If there are different kinds of icons they could be specified as subtypes having their own characterizing properties and possibly inheriting common ones from the main Icon concept.

In DSM, a language identifies all the relevant concepts, whereas a general-purpose language leaves most, if not all, for the modeler to decide. The latter choice then

makes model checking, reuse, integration with other models, and code generation difficult. Consider, for example, integrating two models. You may use string matching by naming models or model elements similarly or have special properties or other naming concepts to indicate the integration. The other alternative is to recognize integration already at the language level: the modeling concepts are the same or share the same details. Using two UML diagrams as an example: Class diagrams and state transition diagrams clearly share some information, such as operations in a class and events of a state transition. This integration can be left to the modeller, or the modeling languages (metamodels) can be integrated. In the latter case, the name of the operation in the class diagram is the same modeling concept as the event name in a state machine. Since the metamodel knows that these concepts are the same, it is possible to check them, and when the name needs to be changed in one place, its change can be propagated elsewhere. The metamodel knows how the languages are integrated.

To achieve the benefits of DSM, you should in general always seek the possibility of having language support. This gives first-hand support allowing errors to be found and even preventing them early, guiding during development work, supporting reuse, and so on. Modelers then don't need to learn all the details, and it becomes easier to define a code generator as you can be more sure that the input for code generators, models, is correct. If you leave domain extensions to modelers, everyone needs to know them—yet they still do them differently! There are also multiple ways to specify these extensions, such as naming conventions, annotating and commenting models, or using additional languages such as constraint languages or action languages. These choices depend partly on the tool chosen for language specification and language use (see Chapter 14 for tools).

Mapping Domain Concepts to Different Modeling Concepts Domain concepts can be mapped into different kinds of concepts in modeling languages. The usual starting point is that each domain concept maps one-to-one to a modeling concept. The main concepts often act as objects existing more or less independently from others. When adding more details, the focus generally moves from objects to other kinds of language concepts such as their properties, connections in terms of relationships, roles, or ports the objects may have in different connections, submodels, and even links to other model elements expressed in other languages.

The selected language type (e.g., state machine) and its MOC help to determine what kinds of additions are possible. You enrich the chosen base languages with domain-specific concepts and rules. To illustrate the use of alternative modeling concepts, let's inspect another example from the watch case: Consider supporting a new button pressing policy in the language. Instead of pressing a button only once to start an operation in a watch, we have a new policy that is based on keeping the button pressed for a longer time, like 3 seconds. How should that be expressed in the language? It could have any of the following:

1. Its own modeling object: A long button press object in addition to the existing short button press.

2. A property of the current object: The current button concept in the language can be specified as either a short press or a long press. If the short press is the most typical case, the property could have the corresponding default value. Alternatively, if the value is left unspecified, a code generator could produce the default value.
3. A role (or relationship): Different ways to press a button can be specified as a new role or a relationship, keeping the button concept in the language unchanged.
4. A property of the current role: The pressing policy could be specified by adding a property for the current role that represents a button press.

Other kinds of extensions could be considered, like having the button pressing policy specified in the family diagram in which the button is first introduced. This concept, although hardly relevant for our sample case, would keep the behavioral designs untouched and move the decision to each individual watch product.

View of Variation Creation of a DSM solution should also be inspected from the variability point of view: how differences among applications can be specified. This is not always a topic for language definition since the variation can be handled elsewhere, typically in a generator. This makes management of variation simple for developers, as it totally hides it from models. A developer just chooses among different generators, which actually access exactly the same set of models to produce the code. For example, the mobile phone case uses the same modeling language but different generators to produce either Python or C++ code. Similarly, the digital wristwatch case illustrates the use of the same modeling language but with different generators, such as one for Mobile Information Device Profile (MIDP) Java and another for C. This offers the benefits of having a single source and multiple targets: the cost of creating applications for other target environments is almost nonexistent. Also, bug corrections and other changes need to be made in one place only and newer versions can be generated for all the targets.

Usually in such cases, the underlying target environment is different, as are the supporting components and framework code. In the case of the Symbian smartphone, there are two options: when creating Python applications the generator uses the Python for S60 library and produces corresponding framework code such as dispatcher and stack handling code; to produce C++ code, native for Symbian, the generator instead uses the UI platform of S60. The framework code can be produced by the generator or it can be selected from the library during generation. The same rules apply here that applied to selecting components: Each component that is optional or whose use is based on variability can be represented with variation data. For example, in the watch case there are two different Canvas components for Java implementation: one for running the application in a browser and another for MIDP devices. These components are then characterized with an additional property for specifying the platform, i.e., “applet and awt” or “midlet.” Depending on the target, the generator then reads the platform properties given in models and includes the right

components along with the code generated from the models. The same principle is also normally followed in cases where there is a need for different generators for different purposes, like one for early prototyping, one for producing code with model debugging information, and one for generating production code.

Supporting Variation in Languages Most domain engineering approaches (e.g., Kyo et al., 1990; Arango, 1994; White, 1996; Weiss and Lai, 1999) emphasize language as an important mechanism to handle variation. This means that variation is represented right in the metamodel. In the simplest case, a variation point is defined as a property of a modeling concept. The possible parameters of variation can be further defined as predefined values from which a modeler chooses the right one. In the watch case, for instance, each display element can be specified by the icons it may use. These are defined in the metamodel as a property type having as predefined values the icons that can be used. A modeler then just picks one or more of them depending on his need. The parameter can naturally be more complex than just a single value, like an object having additional properties, a graph, or even a set of graphs. In Fig. 10.6, the display function showing clockTime illustrates such a parameter selection. The design on the left shows time in minutes, seconds, and milliseconds, whereas the application on the right shows time in hours, minutes, and seconds. This choice of the central time unit is made from the list of possible predefined values.

Rather than having separate properties for illustrating variation, the main language concepts can be applied to describe variation. Every model element added to the specification can originate from the variation. If we again use the watch case, since it is the best case of a product line among the examples in Part III, placing an object type like “Alarm” means that the application has alarm functionality. Further, variation can be represented with model connections. By connecting the alarm element, we can describe how the alarm is set and what happens when an alarm rings. In the same way, the transition relationships illustrate the variation in execution order within each application. The same transition relationship is also used to specify execution order among the applications for a particular product. For example, Fig. 10.6 illustrates two different watch models: one with three applications and another with five. The variation is specified here by having different application states as modeling objects and by connecting them in different orders with relationships.

In the above case, each application state is further described in additional submodels: each state has one subdiagram describing how each application, such as time, stopwatch, and timer, works. This illustrates the option of having complete diagrams as sources of variation. If the same stopwatch application cannot be specified for two cases, the language then allows making different subdiagrams for each case. This option means that two different designs are created. The obvious drawback is that the application specifications most likely have some similarities but are still created separately. This not only means that there is duplicate work in the beginning but also that possible bug corrections and extensions that are beneficial to both applications need to be made multiple times: here twice, to cover both models.

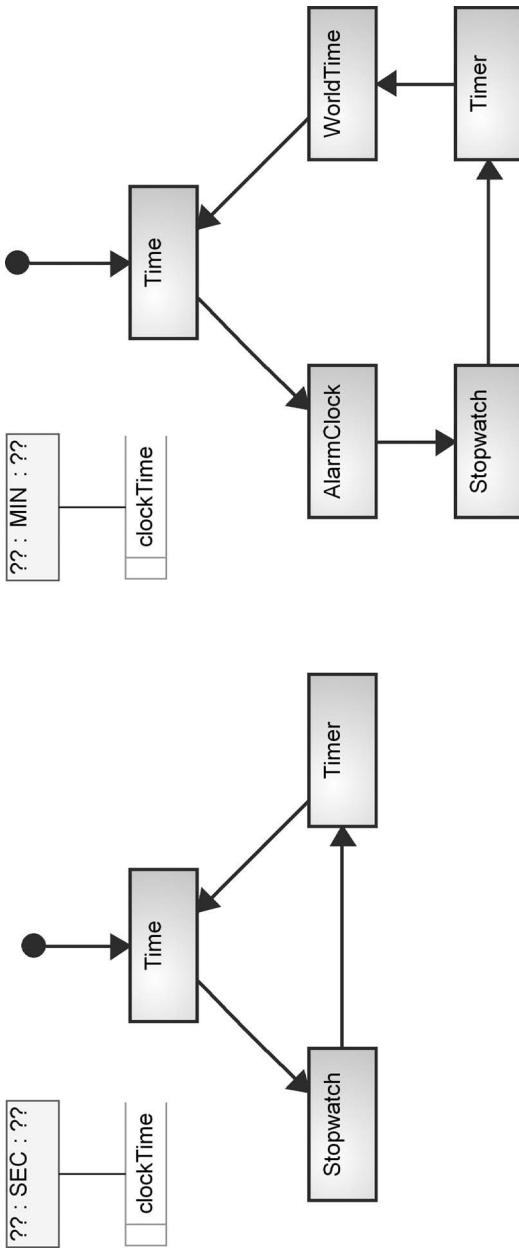


FIGURE 10.6 Two different models describing two variants based on including applications and their execution order

Capturing variability does not need to be limited to specifying just added functionality. Modeling concepts can also be used to remove, or more accurately to ignore, certain model elements during code generation. Following the same language structure as above, one way is to have a property for those elements that can be optionally ignored. The modeler can then just change the property values for those elements that are not included in the generated code. If the excluded elements are connected with other model elements, then they can be excluded too. An alternative mechanism to ignore some model elements is to connect them to a model element that is used just for excluding parts of the design based on the required variability. Such exclusion elements may have their own properties that the generator reads while selecting the exclusion elements and ignoring the related model elements during code generation. Often, though, having multiple relationships just for exclusion may make the model complex since it now has both application data and exclusion data. This becomes even more complex if there are multiple variability objects, one for each variation point. Then, it often becomes better to create a configuration diagram just for describing the variation. This variation diagram then usually works also as a starting point for code generation: it is related to other models and the generator already knows the desired variation when accessing the actual specifications.

Modeling Concepts Should Also Support Editing Language definition should not focus only on the final static model but also cover the use of the modeling language. This means thinking about the model editing process, how models can be kept consistent, and how reuse of model elements can be supported. In our example of button pressing, having different button objects for a long press and a short press would make some model editing actions unnecessarily complex. Simple refactoring actions, such as changing the name of the button, could mean editing the name twice: for both button types when two different policies are used. It would also become more difficult to check the consistency of the models: what if only the short press button name is changed from “Mode” to “Set”? How could we inform the modeler that the “Mode” long press needs to be changed too? Defining the button with information about its usage would limit reuse possibilities and increase modeling work: we could not reuse the button specification so often anymore as it would always come with its usage information. While modeling, we would then need to specify the button multiple times if it is used differently in the same application. A closer mapping to the behavior in the problem domain would lead to reuse and minimize modeling work. Having the button pressing policy specified in a role allows reusing existing button definitions and specifying its usage information separately only when relevant: The same button element could then be used multiple times in models and have different usage situations.

Language Definition Guidelines Typically, the best way to start language definition is to extend the selected MOC with basic domain concepts: first, add the most essential and most used language elements. Then, continue by identifying their connections along with the various rules and constraints. As most of us will be defining our first language for the domain, it is important to test the modeling concepts

early. This means trying out the language early on. Having a working language, albeit limited in expressive power, and gradually testing and extending it makes language definition agile.

You don't need to define all the rules in the beginning as the modeling concepts could change. The same applies for notation and generators. Implementing them too early, when the language is not yet stable, can waste time and effort. Some guidelines for language definition are as follows:

- Follow established naming conventions: While defining the concepts, it is usually best to use exactly the same names and naming policies for the language concepts as are already used. Sometimes during code generation you may need to follow a particular naming policy (uppercase, lowercase, etc.) to map to the selected implementation target. Although it is possible to make the generator do the translations, defining a DSM solution becomes easier if the naming policy is supported by the language. This applies not only to checking values entered in the models but also to naming modeling concepts. For example, in the IP telephony service example (Chapter 5), the naming of modeling concepts is taken directly from the domain and expected output. The naming applied in language types can then be straightforwardly used by the generator to produce names for XML. Having multiple implementation targets, however, often means the generator translates model data to the given target language.
- Keep the language simple and minimal: One of the most typical errors is trying to cope with all possible imaginary scenarios or make the language theoretically "complete." This makes things unnecessarily complex and increases the burden of defining a higher abstraction. It is best to stick with the identified needs and support them first. You can always extend the languages later if needed. If we stay with the watch example, the language for logical behavior now supports basic arithmetic operations for time, but for the sake of covering a larger set of operations, we could also add, or example, multiplication and division. But since these operations are not currently needed, supporting them would just take extra effort with no immediate or long term benefit.
- Try to minimize the modeling work: Don't ask modelers to fill properties you already know or can infer: they can be produced by the generator or provided by the domain framework. Follow the principle of using convention over configuration. For example, in the mobile phone case (Section 8.3.2), the need to specify cancel navigation is made unnecessary in over 90% of the cases, as a default target for canceling can be produced by the generator. Still the modeling language has the concept for specifying the cancel navigation if the default path is not appropriate.
- Have a precise definition for each modeling concept: When defining modeling concepts you need to precisely define what each concept means. This is unlike in most general-purpose languages, in which semantics are vague and left to every individual model creator and reader to define. Code generation expects that you know what you are automating. You should have examples illustrating the alternative cases, the concepts, and how they behave.

- Consider language extension possibilities: If the domain is new or it is unclear whether the defined language provides the needed modeling capabilities, you may add special extension concepts to the language. For example, add to the language a modeling object that can be connected freely with other modeling concepts and that has just one description property. The code generator can then skip these model elements, yet we can inspect what additional modeling needs there may be. Alternatively, we may limit the extension to only a few places, in the modeling concepts that it makes sense to extend. A special case of this is having a “code object” in the language, including plain code or referring to external sources having the code. For example, in the mobile case, the form validation function is made open in this way: a developer can enter any Python script in the validation function and that code is then included in the generated code. Clearly, this must be used sparingly to maintain a high level of abstraction.

Every domain concept (and variation point) does not need to be in the language. Some relevant domain concepts can be “composed” by combining existing concepts. For example, the mobile phone case applies the state machine as MOC where navigation shows that the state changes in the application but the language still lacks traditional transition characteristics like event triggering the transition, a condition that needs to be met and the action performed during transition. Events are already limited to a few possibilities, like pressing predefined softkey buttons, and a condition is left to be specified only in a few cases where a selected value is used as a condition for choosing a specific navigation path.

10.3 FORMALIZING LANGUAGES WITH METAMODELING

As soon as you have identified the relevant parts of the modeling language, you should formalize it. This is best done by defining the metamodel. Formalization into a metamodel is needed because otherwise the language does not guide the modeling work and code generation would not be possible (see Section 4.2.4 for details). We need to emphasize here the difference from the definition of UML, which is also described as a metamodel. The metamodel of UML is a collection of separate, loosely connected class diagrams that are not instantiated and tested during language development. Partly for this reason tool vendors implement support for the languages of UML differently. They may look the same, but inspection of the details shows the difference. You can avoid this by using a metamodeling tool that can also execute the metamodel. The metamodel also serves as a basis for integrating and sharing the models with other tools in the chain.

10.3.1 Metamodeling Process

Metamodels are often best specified first with just pen and paper in some format. This usually means drawing a data model showing the modeling concepts and their connections. Often companies use an entity-relationship diagram or a class diagram in

this initial specification. If a class diagram is used, do not specify details, such as operations of classes and relationships other than associations and inheritance. When you know the structure to some extent, it is time to define the metamodel.

Metamodeling simply means modeling your language: mapping your domain concepts to various language elements such as objects, their properties, and their connections, specified as relationships and the roles that objects play in them. This process is supported by the metamodeling language which, needless to say, should itself be a domain-specific language for specifying modeling languages. The metamodeling language depends on the DSM tool you use, but at a minimum it should allow you to define the concepts of your language, their properties, legal connections between elements of your language, model hierarchy structures, and correctness rules. In all but the smallest cases, support for reuse and different model integration approaches is also essential.

Language for Metamodeling A good metamodeling language guides you during language definition: it allows you to focus on defining modeling concepts and hides the implementation details (how to run it in editors). During metamodeling, you specify some of the language concepts directly and others by combining some domain concepts. In making a decision about which concepts to include, it helps to use your language for modeling and generate artifacts from it. Here, tools can help you, as ideally they should allow you to focus on language definition only and provide various modeling editors for your language instantly and automatically. This makes language creation agile: you can easily test and learn what the language looks like in practice, how it allows you to make and reuse models, and so on. This minimizes the risks of making a bad language, or a good language but for the wrong task, and helps in finding good mappings for code generation. This kind of prototyping is best when language design is something new for you or the domain is not yet completely defined.

Metamodels for the Language Definer, Models for Others The possibility of trying out your language specification immediately after you have defined some of the concepts significantly changes the role of metamodels. First, they are formal: you can run them. Next, they formalize the domain knowledge in such a way that other developers and stakeholders can understand it too. They may not be able to understand the metamodel, but they will understand models created using terms they are familiar with. Concrete example models allow to show the idea to other developers early and make it easier for them to understand the ideas and contribute to the language. This will greatly support language use later, in the introduction phase.

10.3.2 Sample Metamodeling Task

Let's next inspect the metamodeling process with an example. Consider the case mentioned earlier of supporting different button pressing policies in a modeling language specifying watch applications. The task of the language designer is to choose the most suitable structure for the language and define it into a metamodel.

The options presented in Section 10.2.4 were creating a new modeling object (long button press), adding a property to a current button to choose the policy (short or long press), adding a new role to the language (long press event), or adding a property to the current Event role for different pressing policies. Since the first two options do not support reuse, and the same button would be used in every case with the same policy, it is best to modify the role. This allows us to use the same button in different ways: sometimes the button is pressed briefly, and sometimes for a longer period. Rather than making the language larger with its own dedicated role type for long pressing, we decided to add a property to the current Event role.

Figure 10.7 illustrates this change. This metamodel defines that there can be a transition from a state to another state or a stop state. A transition can be activated by an event caused by a button, and it can initiate an action. You may compare this to the metamodel illustrated in Figure 9.9. The only difference is that the Event role now has a Boolean property called “Long press?” Its default value is false since a short press is the more common case.

Once the metamodel is updated, we can next try out the language. Depending on the tool (see the different alternatives in Chapter 14), you may need to generate the metamodel for another tool or just use it immediately for modeling. Again, depending on your tool you may need to define a new model or the tool can update the existing models automatically for the changed language. Figure 10.8 describes a sample model, in which the long press policy is used for the Down button. An ellipse and related text are used in the notation to visually indicate the difference in the model. Using both is perhaps overkill, but something is necessary to help read the model.

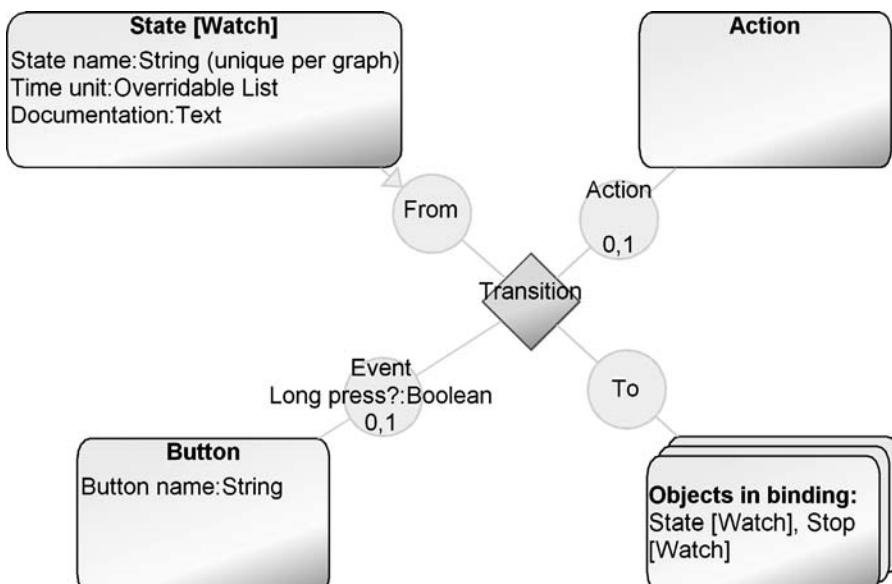


FIGURE 10.7 Metamodel having support for alternative button pressing policies

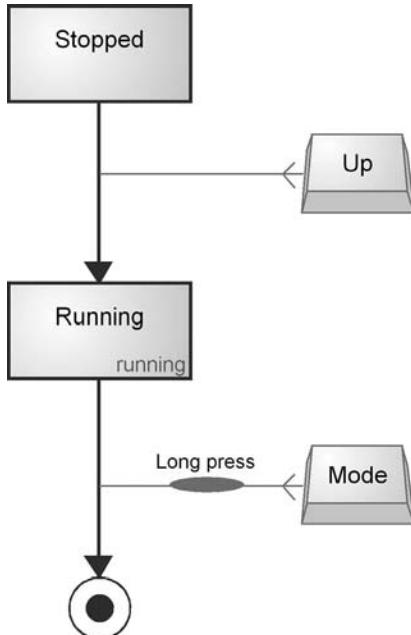


FIGURE 10.8 Sample model that is an instance of the metamodel specified in Fig. 10.7

10.4 DEFINING LANGUAGE RULES

Along with modeling concepts, we also normally detect various domain rules, constraints, and consistency needs which a language should follow. These rules obviously need to be defined too. Having rules in the language provides many of the benefits of the DSM approach: they make a language domain-specific. Typical benefits are as follows:

- Prevent errors early: illegal or unwanted models simply can't be made.
- Guide toward preferable design patterns.
- Check completeness by informing about missing parts. While not necessarily relevant for the modeler, the code generator expects a “complete” specification. It is worth noting that by “complete” we mean that models can be used as input for full code generation.
- Minimize modeling work by conventions and default values.
- Keep specifications consistent: if one element is changed in a model, the change is reflected elsewhere to either update models or report about the inconsistency.

In our experience, the rules are best defined after having decided on the main modeling concepts: then you start to see the patterns and rule types. It is not rare to notice that a certain rule, such as occurrence, naming, or cardinality in a relationship, is

shared among many modeling concepts. We can divide the rules into those that originate from the domain and those that deal with how the modeling language operates.

10.4.1 Domain Rules

Most of the concepts in a modeling language come with rules. These rules should naturally be recognized by the language too. The types of rules you can specify depend on the DSM tool and metamodeling language applied. We discuss the rules here independently of any tool, but the types of rules that occur most often in modeling languages deal with the following:

- Naming conventions, e.g., a value must start with a capital letter or must not include certain characters
- Uniqueness, e.g., there can't be another element with the same property value
- Mandatory properties forcing an element to have a value
- Default values
- Occurrence: a concept can only have a certain number of instances in a model
- Binding rules stating which kinds of elements can be connected together
- Connectivity rules stating how many times an object may have a certain kind of connection
- Reuse rules stating that a modeler can choose a certain value or refer to another model element
- N-ary relationship rules stating how many objects a single relationship can connect
- Integrating models, such as sharing the same value with another element, possibly in another model, and possibly made with another language
- Model structuring rules, such as hierarchies or references to libraries

You may identify these rules from the same sources you identified the domain concepts. However, detailed language rules often can't be detected from existing material, and the fastest way is to check them directly with the domain experts. You may also look at the kinds of rules that the selected model of computations uses.

10.4.2 Modeling Rules

Strict enforcement of domain rules can sometimes conflict with the needs of modeling. While the domain rules as such are obviously correct, they may not always work well as rules of a language. Usually, this means that the usability of a language would suffer if domain rules were used as-is. A classic example is forcing the removal of an element, like a relationship, before a new one can be created. While correct by static domain analysis, it can often be better to temporarily permit the inconsistency, allowing modelers to see the old information while entering its replacement. Past information is often useful when creating new model elements.

It may not always be possible to check domain rules in a language. The number of model elements and models, possibly made by other developers at the same time, could be so large that checking rules after each model manipulation would require too much time, hindering modeling work. Checking some rules, like uniqueness among all model elements, can be time consuming or often impossible. It is, therefore, sometimes better to allow illegal models for a while. For example, checking minimum cardinalities at modeling time does not normally make sense since it would prevent modeling work. Adding a start state to a state transition diagram would create an illegal model since a start state needs to be connected to some other state, making it the initial state. In contrast, maximum connectivity, checking that there is no more than one transition from a start state, is good to put in the metamodel: it ensures correctness without interfering with the normal flow of modeling.

Checking and Informing About the Rules Rules can either be strictly enforced in the metamodel or shown as warnings by a separate model check. Placing rules in the metamodel is usually the better choice since it keeps specifications legal at all times. The worst option is to check the rules just before code generation since then there is no support during the actual modeling.

Depending on the tool, the results of rule checking can be visualized in different ways. Ideally, the best way is to give immediate feedback after modeling creation tasks. This necessitates that a model, and with some rules all models, need to be checked after each model manipulation task. In practice, this only makes sense for certain model manipulation tasks, such as creating a relationship or changing a property value. If immediate feedback is not possible, then model checking can be done as a separate process, typically when the modeler decides to do so or when a code generator is executed. Here, modelers can be informed in different ways about the results. The most natural option is showing the warning close to the model element or highlighting the elements the rule is related to. For example, an element having an error or not yet completely specified could be selected or reported. Each modeling action can also inform about the possible illegal action or even inform the modeler about other options of language use. The last option is to have a separate checking report showing the results in some textual form, like an error dialog. Tools may also help in the visualization, such as by allowing checking results to be traced back to the model or model element having an error.

10.4.3 Rule Definition Process

In general, the rules of a language should work like file security: nothing is permitted unless it is specified. The advantage of this approach is obvious: language definition becomes easier to handle, as new rules can be set and tested incrementally. This is the exact opposite of using profiles in UML: things are already allowed and then we start adding extra rules to override those already defined in the standard metamodel. This not only complicates the definition but also makes the rules cumbersome and numerous. Try it yourself: implement the languages described in Part III using plain

UML as a starting point and adding rules with profiles. You will quickly find why metamodels are simpler and easier to define.

Remember that rules can evolve over time too. Modelers usually require more expressive power and the domain itself changes. For example, if modelers can't specify some functionality, you may relax a rule and move its checking to the generator. Alternatively, if unwanted structures or specifications that lead to poor performance are identified, you may always add rules to the language or check the models before starting the code generation. A DSM tool should then allow updating the models made with the earlier language version along with the metamodel.

10.5 INTEGRATING MULTIPLE LANGUAGES

A good design language is not isolated from other views modeled with other languages but is integrated. This is nothing new: the software we develop today for manipulating data acts in a similar fashion: when a loan application in a bank uses account numbers, these are not saved separately for this application but are shared with others, such as the ones managing the accounts, calculating interest ratios, and so on. Specification data in models is no different: its integration needs to be specified similarly. You may integrate models specifying different views of the application, or integrate models to existing code libraries, to other models describing variability, or to models specifying nonfunctional requirements. As a language developer, you can build this support right into the language. You may also consider doing integration via transformations, but that seldom works, as explained in more detail next.

10.5.1 Integration Approaches

The best way to integrate languages depends on the number of views, how model data needs to be available to developers, and whether there is a need to keep developers in separate modeling spaces or modeling is based on reusing others' work. Figure 10.9 illustrates the most typical approaches.

The first approach is to keep the languages separate and integrate them at generation time. This approach makes things simple for the language developer but more difficult for modelers and generator developers. This is reasonable if you need to

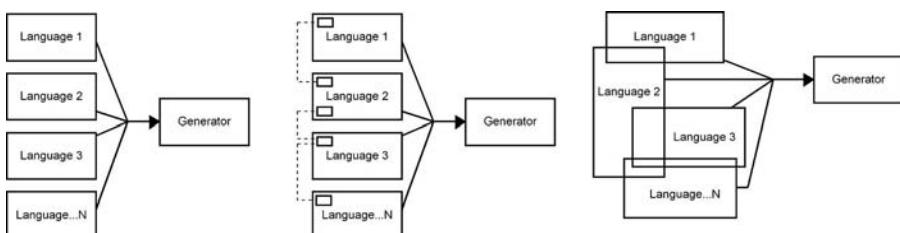


FIGURE 10.9 Language integration approaches

keep the modelers separate or they don't need to interact. For example, while using subcontractors you may decide to keep some models for internal use only and have another language for subcontractors. It can also be that the nature of modeling work differs among developers so that all groups don't need access to others' work. This approach naturally makes generator development more difficult as it can't guarantee via language rules that models are correct. The generator then needs to run the checking as a separate process or, more likely to check the input internally without providing trace links back to models.

Whenever possible, it is better to support integration at modeling time—it enables integration before the generation. The second approach is to define concepts enabling model integration. Stahl and Völter (2006) suggest the use of a few concepts that are common between two or more languages. These are called gateway elements in the metamodel. These few special modeling concepts are then used to link separate languages (and models) together during generation.

In reality, the domain concepts are integrated and related, whereas the integration concepts are just additional reference elements that need to be handled separately. Therefore, it is usually much better to define that languages share and refer to the same modeling concepts. This means integration based on the metamodel. You may share the same modeling concepts completely or just parts of them, like some of their properties. You may also integrate languages based on model decomposition, have another language for specifying details of one particular model element, or define other mappings between modeling concepts. While this is more difficult for the language developer, it makes the life of modelers much easier, and generator developers can better rely on the input for generators being correct: it is checked in the models. For modelers, the integrated metamodel gives a fundamental benefit: They can see other models, and models update automatically based on changes made elsewhere. Data in models can then be edited in an integrated way and there is no need for copying and pasting elements between different models or maintaining references among models. If the DSM tool supports multiple users, the change can be simultaneous for all developers. Also, model partitioning and versioning can be supported by allowing work with models consisting of multiple integrated specifications, rather than versioning small specifications separately and externally keeping track of their dependencies.

10.5.2 Why Integrating Models with Model-to-Model Transformation is a Bad Thing

Experiences with model-to-model transformation, whatever tool is involved and whatever export/transform/import method is used, show this is normally a Bad Thing. You should only consider it when you don't need to change the automatically produced model. Let us explain why.

Normally, the idea is that each piece of data in one model gets transformed to more than one piece of data in the second model (let's say two pieces). This is fine if you never (or rarely) look at the second model and never (or very rarely) edit it. But if you edit it, you are now working with two pieces of data, but clearly they are not totally

independent, since they could be produced from one piece. The idea of DSM is to come up with a minimal sufficient representation of systems, and this is the main reason for its 5–10 times productivity increases: code- or UML-based ways of building systems involve lots of duplication—the same information in several places.

People asking for model-to-model transformation say next that “we want to be able to change the first model still, and have those changes reflected automatically in the generated model.” That means you are working with 1 + 2 pieces of information, and also that you need to come up with some way to propagate the changes down to the second model “correctly.” “Correctly” here means without destroying information manually added there, updating the automatically generated parts, creating newly generated parts, and updating manually added parts to reflect changes in the first model. This last case happens if the manually added part refers to the name of an element originally from the first model and that element’s name has now been changed.

Next people say, “and we want to change the generated model and have the first model update automatically.” This is even harder. The only cases when it can happen are where you don’t really have a first model and a second model but rather two different representations at the same level. Even then, it can only apply to the intersection of the sets of information recorded in the two different models. For example, in UML tools you may have one class in a model mapping to one class in the code. With sufficiently simple changes, a bit of luck, and a following wind, the best UML tools today are capable of maintaining that simple mapping for the names of classes, attributes, and operations. The actual code isn’t kept in synch. Although tool vendors tend to claim that UML models are kept in synch, this generally means only some parts of class diagram elements. Synchronization is partial simply because the details of the code are not in those models that the tool can keep synchronized. The only way the synchronization can be made “better” is by moving the two languages closer: for example, by allowing UML operations to contain the method body as text, or the code to show things like “boundary class” as specially formatted comments. Each move toward better synchronization is thus a move away from having the first language on a higher level than the second language.

The DSM solution is to turn the question on its head and ask, “OK, you showed me the model, possibly based on a high-level modeling language, that doesn’t yet capture enough information to build full systems. And you showed me a second model and a transformation into it. Now tell me what extra information you want to put in the second models.” Note that here we’re asking for information, most likely on a (problem) domain level, not for the actual representation of that information. “Now let’s look at how we can extend or change the high-level modeling language to provide places to capture that information.” (Sometimes this step may require rethinking the modeling language, especially if it’s been based on a language that somebody had already made.) “And finally, let’s show how we now have all the information we need, and we can generate full code directly from the first model, using the information it originally captured and the new information we would otherwise have entered in the second model.” You should remember that there was only a certain amount of information to be added to the second lower-level model, regardless of its

representation or duplication into several places there. Since we would earlier have been able to generate the initial second model from the information in the first high-level model, add extra information, and then generate code, we can now clearly expect to be able to generate full code straight from the first model.

Summa summarum: if you can specify a transformation from a high-level model to a second, lower-level model and then have modelers add more detail and perform another transformation into code, you can look instead at the information that gets added by modelers, extend the high-level language used to specify the first model to capture it, and merge the transformations into one. It makes life easier for modelers since they have just one language and one model. There is no round-trip synchronization hassle or need to manage different models, their sharing, and versioning. It also makes life easier for the creator of the DSM solution: one language, a single one-way nonupdating transformation. Additionally, you normally find a way to record the information into significantly fewer data elements in the higher-level language.

10.5.3 Reuse with Models

Reuse of models and model elements can be recognized as first class citizens by the modeling language. This means that the metamodel knows when and where to find reusable model elements instead of creating them again. Reuse at the model level is naturally desirable as reuse can then happen at a higher level of abstraction rather than at the code level, and it can happen at modeling time. This means that reuse is built in to the metamodel.

The first approach to reuse is simple string matching: referring to a model or model element by typing its name in another model. This approach is possible in every language, but such reuse is not guided or enforced. Modelers need to enter particular values, usually following some naming conventions, and the generator makes the mapping and finds the reusable elements to be included in the generated output. The modeling language does not “know” when it would be better to reuse existing model elements instead of creating new ones, where to search for reusable elements, or whether reuse is based on a white-box approach showing internal details or a black box showing just the public interface.

A more sophisticated way is to share some of the same modeling elements between the models and modeling languages. Reuse of model elements in different parts of the same model is illustrated in the mobile phone case (Chapter 8). The return variables that store the values entered by the user are also used as input for other modeling concepts. For example, the SMS sending object has a variable message element that refers to any specified return variable. This allows the definition of SMS sending to use any variables entered earlier, and if the name of the return variable needs to be changed later its users, like SMS sending, don’t need to be updated manually. The metamodel takes care of that refactoring. Reuse is not limited to single values but also can involve whole models or their parts. For example, in CPL (Chapter 5) the subaction concept refers to other diagrams. This allows reusing any call processing service already defined. The watch example (Chapter 9) illustrated reuse of selected model elements

rather than whole diagrams. The family model specifies the static elements of a watch model, such as buttons and icons. Later, in the behavioral model describing a logical watch, those buttons and icons could be directly reused. Changing the button in the static product component specification then automatically updates all the behavioral specifications that use the button.

Reuse always takes place in some context and direction and these can be specified in the DSM language. For example, the metamodel can make it impossible to create new model elements and enforce reusing instead. More flexible would be first guiding to reuse but allowing new model elements if suitable ones are not available. Finally, the language may be defined so that reuse is not mandatory and can be specified later once the reusable model or model elements become available.

10.6 NOTATION FOR THE LANGUAGE

Notation gives a visual representation for the models. Defining the notation makes sense only after the modeling concepts are already identified. Notation is therefore the wrong place to start defining a DSM language. Although model representation plays a minor role in code generation and other automated tasks, it is highly relevant for acceptance and usability. Especially in the beginning when a modeling language is new, there can well be more comments and feedback on how a language looks than on how it works. The creators of a DSM solution may feel a bit disappointed seeing that developers now work much faster and produce better quality products but only give feedback on notation and symbols. This can be because the other parts of the DSM solution are not as visible to modelers, and notational issues are always easy targets for opinions.

A good practice when defining the language is to ask users to come up with the symbols and other representational elements. This increases user involvement, helps in solving any “not-invented-here” attitude, and makes the language easier to learn and use. By language user, we don’t mean only those creating the models. The more models are used for purposes other than designing for code generation, the more likely we need others’ involvement. Those who read models, for example, to validate the specifications could participate too. Also, if some models are used to make product presentations by sales people or create configuration tools for deployment, it is good to ask opinions from this larger audience. It is much easier to introduce them to something they helped define: it has become their language, not yours.

10.6.1 Selecting Notation and Representational Forms

For defining a notation several guidelines can be given. For choosing notation, Costagliola *et al.* (2002) present a framework of classification and Rozenberg 1997 has edited a handbook on graph grammars and graph computing. In general, we can use anything for notation, from photorealism to an abstract box. Based on studies on cognitive dimensions (Blackwell, 1998), implicit, stylized pictograms

are often better than photorealism, making models easier to read, understand, remember and work with. Photorealism also makes it hard to add text within symbols. This calls for more abstract notational elements that also illustrate detailed information with textual or pictorial properties. The worst approach is to use the same abstract symbol for different concepts. Using the same symbol for all the different concepts is like trying to understand a foreign language where the only letter is A with 20 slight variations of inflection! This is unfortunately often the case when using profiles to modify class diagrams. All the different domain concepts look the same: a rectangle. The only difference is a stereotype label, which is often presented similarly to the rest of the design information: using the same font and location inside the main symbol that is used for the other central data in the model.

Selecting Symbols for the Notation The best practice is to take representations for the notation and its symbols directly from the representations of the domain concepts. The sources for the notation are often the same as for the actual concepts. In cases where the domain is related, for example, to user interfaces, the notation can be defined rather easily, as was done in the mobile application case presented in Chapter 8. Also, if the domain to be modeled includes physical structures or has already established representational forms, the definition of the notation is considerably easier. You may use symbols of the physical product, such as a valve or sensor, to denote the actual software behind these concrete product elements. Usually all notational elements cannot be fully derived from already known symbols, so the rest must be created for languages. However, there is no reason to create new notation just for the sake of having something new. Instead, you should use and borrow good representations already in common use. Use the base language and model of computation you started with to help and guide you. For example, in models describing behavior such as state machines, flow diagrams, activity diagrams, and so on, well-established notation for start and end states already exists: a dot and a circle containing a dot. If your language gives additional semantics (modeling concepts or their properties) for the start and stop states, you may well extend the representation. For example, we may have a stop state that ends the application and another kind of stop that just ends the current function but not the whole application. These alternatives should also be made visible in the model by giving them different notation.

Usually, notation definition starts from the main modeling concepts but it is also reasonable to start with those symbols that come most naturally, giving a style that forms a basis for notation for other concepts too. When inspecting the other language concepts, you may find that they can use similar representations. Using a traffic sign for one concept may lead to considering other traffic signs for other concepts. Since notation can be richer than just showing basic design information, it is often best to define first the main notation and later add extensions that indicate special information, such as having a consistent symbol element to indicate that a model element is not completely defined, has errors, is described in more detail in a submodel, is reused from a library, and so on. Remember that notation can help the

reader and it can change based on the properties it has or even based on the connections it has.

Each concept of the modeling technique should normally have one identifying representation. You should find representations that differentiate the concepts. This principle minimizes the overload of notational constructs and guarantees that all domain concepts can be distinguished in the models. Accordingly, the completeness of representations (Batani et al., 1992; Venable, 1993) or representational fidelity (Weber and Zhang, 1996), that is, availability of exactly one notational construct for each concept, is a well-known criterion for dealing with interpretations between modeling concepts and notations. If you have a lot of similar kinds of concepts that are still different in the modeling language (e.g., have different rules, connections, properties) you may use the same shapes, colors, fonts, and so on to identify their kind. This approach was used, for example, in CPL (Chapter 5). You may also define notational elements so that they visually show the same aspect, subdomain, reuse, origin, or architectural role. For example, in the watch case (Chapter 9) colors are used to show the MVC (Model-View-Controller) architecture.

Selecting a Representational Form The concrete syntax, how we represent the models, is not limited to symbols. We can choose among different representational styles too. The example models in this book illustrate representations of different graphical modeling languages where the symbols and icons represent different modeling concepts. Sometimes a matrix or a table works better than diagrams. A matrix is especially good if connections between model elements are important. A matrix also scales better than a diagram since more information can be shown in the same space. With matrix representation we thus don't often need to partition model elements into submodels or parallel models. Table representation is especially good in showing properties of modeling elements: a parameter table is a classic example. The challenge in a table is showing the dependencies among the elements.

The choice of the representational form can also depend on the model manipulation actions. For example, a matrix gives a basis to automatically identify high cohesion and low coupling between model elements with diagonalization. Tables and matrices also offer other model manipulation options, like sorting based on model information. A classic example is finding priorities based on the properties of model elements. Diagrams are particularly good in finding patterns and organizing model elements into subdesigns. Ideally, we don't need to fix on one representational style. The same model can be shown in different representational forms and model manipulation operations could take place in any of the possible representations.

10.6.2 Symbol Definition Guidelines

Although each notation will look different depending on the domain, some principles can be generalized. We have found the following symbol definitions to be effective.

- Use different kinds of notational elements for different modeling concepts.

- Use square and rectangle symbols when you need to show more text inside a symbol: the space can be used better than the ellipse, cloud, circle, triangle, and so on.
- Use vector graphics when the symbol needs to be scaled.
- Show only relevant data directly in the visual representation. Those days are gone when all the models were created with pen and paper showing all information on the same sheet. Today modeling tools can use filters and show the details in additional property sheets, dialogs, and browsers next to models.
- When the language is new for users, you may offer more guidance as part of the notation, like showing the name of the modeling concept as part of the symbol. Later, when the language has been learnt, you may remove them from the language definition or perhaps the tool allows users to hide them.
- Use colors. We hardly ever develop software that uses just black and white in its user interface, so why should models? Colors help in representing and reading the models and simply make them look better. You can use coloring and shading to illustrate different aspects or views, like MVC architecture (Chapter 9), similar kinds of domain concepts (Chapter 5), and UI (Chapter 8). Traffic light colors can be used to indicate preferences or choices, and different colors can be used to highlight priorities or follow the already established color use in the domain. Printers today can reproduce colored models with good results and modeling tools can also help in translating colors and different color settings into printable forms.
- Special effects such as small icons, shading, and fountain fills make models look better on screen but they don't necessarily look as nice in other exported picture formats or printouts. They also make files used for interchanging models between tools bigger. Some interchange formats make the files for saving representational data unnecessarily large by saving every model element individually although their representational definition could be defined only once in the metamodel.
- In DSM languages targeting GUI or physical products you may apply two different notational versions of the same element: a 1:1 mapping to the real-world representation, or if that is not yet available, because no UI or product has yet been made, a more abstract notation.
- If different levels of detail need to be shown visually, you may apply different versions of the symbol based on user request, for example, one showing just a few properties and one showing complete details.
- If the notation aims to follow the real product closely, such as the GUI functionality described in Chapter 8 for the mobile phone case, different colors, fonts, and so on can be used to distinguish other model data from elements that resemble the real product. For example, the Multiquery object type in Chapter 8 uses gray for variable names to indicate that they are not part of the visible UI.
- Make symbols of model elements that may contain other elements transparent to allow showing parts inside the main symbol. The alternative is to create two

symbols for the same aggregate concept: one showing information when the concept is not used as an aggregate and another when other concepts are part of the element (moved inside the aggregate symbol).

- Use consistent notational elements to improve model readability. For example, a special icon or text element can indicate if a model element is described in more detail or from another aspect in another model.
- You may consider using special notational elements, such as an icon or a text element, to illustrate the status of the model or of its individual model elements. This typically includes showing an error label for an incorrect model in cases where it was not feasible to include the error checking rules in the metamodel. In the same way, notation can indicate missing properties, showing the model is incomplete. It is generally better to place these special notational elements close to the model element that has the error or missing information to give a context. Textual model-wide checking reports are, however, better for more complex cases since they allow giving an explanation for the error or even instructions to correct the specification in a model. A hybrid solution—requiring more work—is to have both: a special notational element indicating the error in model and a model checking report that explains in more detail the status of the model.
- You may also inspect your corporate documentation standards: applying their look-and-feel in the modeling language improves acceptance and makes the generated documents more readable. Examples of such guidelines are classification schemes for the status of the model, such as draft, and frozen.
- Finally, do not be afraid to ask for help in making the notation: good metamodelers are not necessarily good graphic designers.

10.7 TESTING THE LANGUAGES

It is always a good idea to have multiple concrete example cases to test the language early on. Actually, the whole language creation could be considered as incremental and test case driven: first, you define some of the language, then model a little, make some extensions to the language, model some more, and so on. During each iteration you should test the language. The size of the iteration can be the size of the test case, but ideally, if the tools support it, you should be able to immediately test even small changes of the metamodel. Early and frequent testing as a part of language creation is especially relevant when you are defining your first modeling languages as it minimizes the risks of going in the wrong direction. Test cases also enable user participation and learning more about the domain and its modeling.

Testing of the modeling language is best done using examples from the real world—usually, the more cases the better. Ideally the cases should use as many modeling concepts as possible. At first the cases can be small features of a larger application and over time they are extended to cover the whole target of the language. You may use for two major approaches for finding test cases: rebuilding already

developed applications or creating test cases from scratch just for language testing purposes. When testing is done by people other than the language developers, it is better to use known application features. You could even ask the developers who originally developed the application to test the defined language for the same application. Developers can then focus not on learning the application but on actual modeling. If generators of code, simulation, testing, and so on, are already available for the language, developers have extra motivation as models finally serve some other purpose than just documenting designs.

A large portion of the tests can be done by the language creators. This is natural, especially in the beginning, as the language is not necessarily complete, many changes may still be made, and multiple iterations can occur within a day. Later, when the modeling language is more complete, other developers should be involved. Their feedback is important not only for testing but also to smooth acceptance of the DSM solution.

At this stage, feedback is typically received about notation. Do the symbols look nice? As anybody can have an opinion on the notation, it is more relevant for DSM to focus on the language's capabilities—conceptual structure instead of representation. It is often effective to ask language users to correct representational issues, such as how symbols should look, and in this way try to move testing to more substantial issues. Another good practice is to gather experiences while modeling by including in the language a special comment element: modelers may thus highlight issues they found relevant, but that could not be described with the language, and so on.

There are not many publications on testing and validating modeling languages (e.g., Fitzgerald, 1991; Schipper and Joosten, 1996). While testing the language, we have found three issues to be relevant (Tolvanen, 1998).

1. Abstraction: expressive power when describing the problem domain
 2. Model consistency: organizing and keeping models consistent, guiding on reuse
 3. Support for generators: producing required output from models, typically the code
- (1) Abstraction deals with comparing how well the given test case or application domain in general can be captured with the defined language. Modelers easily get back to you if the expressiveness is not adequate, but you may also check issues like
 - Could the abstraction be higher? If the same kinds of patterns or combinations of model elements occur often, they should be replaced with a new modeling concept. Then, there is less to model and reuse of good practices is easier as the language provides concepts for them.
 - Does the language minimize the modeling effort? If there are repeating elements used just to make the model complete, these could be left to the

generator. For example, in the mobile phone case (Chapter 8) almost half of the navigation flows can be removed in typical applications by moving standard cancel navigation to the generator.

- Have modelers made their own extensions? Models including special naming policies, such as prefix, postfix, capital letters, and special characters, can indicate that modeling power is not adequate. You should find out what is behind this special naming. Sometimes, the naming can be done just to satisfy the naming policy for the generated target language, keeping the generator simpler. If the modelers are not programmers, it is usually, better to let the generator take care of naming, for example, removing spaces from entries in models that are used for variable naming.
- Are all modeling concepts needed? If the language includes concepts that are not applied, those could be removed as they add an extra burden for language users. Sometimes, during early phases of language creation there is a tendency to try to cover more of the domain than is actually needed. Finally getting a chance to master the modeling and code generation, you may go a bit too far.

(2) Model consistency deals with how well the models are kept “correct.”

- Do the specifications follow the language rules? Did we prevent making false specifications or specifications that would lead to poor performance? While metamodel-based rules should be naturally followed in a modern DSM tool, they are not necessarily specified correctly in the first place. Also rules that are supported by separate checks might not be followed at all. You may make checking mandatory by relating it to major modeling actions, forcing it to be run before versioning the model, or checking models while generating code, documentation, or other artifacts.
- How well is refactoring supported? If an element is changed in one model, are modelers forced to update other models manually? Could the same language, or different integrated languages, share concepts? In Chapter 8, the concept of a return variable is applied as the message element of an SMS sending object. Thus, if the name of the variable is changed in any model element and it is used in SMS sending as a message element, the update is automatic. There is no need for the modeler to update the SMS sending element. You can find a similar integration structure, for example, in the button concept in the watch example (Chapter 9). If the button definition is changed in the watch display specification, the change is automatically reflected in all those applications, where the changed button is used—even though they are based on a different language. Today, when projects have multiple developers and they reuse others’ work, this capability saves time and prevents errors that are otherwise easy to make.
- Did the language support reuse? Are all the modelers creating similar kinds of models instead of using existing models? You may detect the patterns

among models and update the language so it covers those patterns of reuse. Try to make it so model and even element boundaries fall in places that make natural units for reuse.

- Do we have rules that are too strict? Sometimes there is a tendency, perhaps because rules were easy to identify, to define language so strictly as to actually prevent modeling actions. You may, for example, correctly define the rules but they make sense only when the models are ready and remove freedom during early sketching or when making model modifications. A typical case here is when model elements can't be changed but need to be deleted before a new correct model element can be defined. As the temporary deletion may also delete related properties, relationships, and even submodels, modelers are forced to model part of the specification twice. Here, checking the consistency rules only on request instead of at modeling time usually solves the problem.
- (3) Automation with generators normally deals more with the capability of the generator than with a modeling language. However, sometimes certain generation actions simply can't be implemented since the initial input is not available or is not easily accessible from the models. Here, you should analyze examples that combine the original data in models and their implementation code as produced by the generators. We will discuss generators in detail in the next chapter.

10.8 MAINTAINING THE LANGUAGES

Languages do not remain static but evolve over time when the domain or generation needs change or when modelers, after having learned to use the language, see new opportunities for modeling. Whatever the change, it is usually best done in a similar manner to how the language was developed: test with sample models to see the influence. If the extension is large, changing many parts of the metamodel, then it is best to do a pilot study before introducing the language updates. The whole creation process of a DSM solution, of which the modeling language is a part, is discussed in Chapter 13.

Here we inspect changes to the abstract syntax, which we specify in the metamodel. Changes in the concrete syntax, the notation, should always be easy to do and reflect in the models. During maintenance you may then add, modify, or delete any of the previously defined modeling concepts.

10.8.1 Adding New Concepts

In general, adding new concepts to modeling languages is easy. After having updated the generator and possibly the domain framework, modelers can continue development—by creating new models or updating the old ones. For example, after using the modeling language to develop mobile phone applications (Chapter 8),

the underlying target environment and its API changed, adding support for threads: When an external application is called for browsing the web or showing images from the file system, another process can be initiated instead of running it in a single thread. This change was put in the language as a modeling concept and no further actions were needed: new applications could then use separate processes. If an existing model wants to use the functionality provided by the new concept, it can be done naturally with a small change to the model.

If the added concept changes rules and makes some existing models illegal, you must decide how to change the models, if at all. Often no model update is needed here since the models are still usable: the generator still produces working code with the old models. If you prefer that models are updated to follow the new constraint, the most typical decision is to inform modelers on those parts in the models that have become invalid. They can then see the model context and decide on actions for their update. If the added rule is well bounded and the model update policy is easy to formulate, you can use tools to define the model update transformation.

10.8.2 Removing Modeling Concepts

Removing and modifying existing concepts needs to be done more carefully: most likely these changes will have an effect on existing models. Good modeling tools should allow using existing models even if some elements of the metamodel are removed.

When the language is modified, the easy way out is to freeze the current language and continue its use only for application maintenance purposes. This is more likely an option if the target environment and domain framework are frozen too. It is more typical, however, that the project wants to upgrade to the newer language. Your task as the language definer is then to analyze in detail the possible side effects: how other data is modified too. It may be that removal of a concept has consequences and may remove more model elements than intended. For example, removing an object type may cause removal of some relationships too.

When after changing a metamodel, it is necessary to test the changes with real model data. After releasing the new version of the language, it is good practice to notify modelers of where the change has happened. This allows them to find the models that may need to be updated. Usually, the update can't be fully automated as the change needed in the model depends on the context, which the model creators know. Depending on the tool, the notification can be made available using model browsers or running a generator that reports on all the models where the change could be made. Depending on the change, it can also be the case that no further notification is needed: when the developers change the models, they will be updated gradually to the new metamodel.

While considering the possible changes you can also identify which changes need the most time. For example, if you need to delete an existing concept and replace it with two new ones, you can consider just refactoring the old one so that the model changing work can be minimized. Naturally the old one is modified, if possible, to resemble the more common concept in models. A DSM tool then makes the update at least partly automated.

10.9 SUMMARY

The goal of defining a domain-specific language is to provide the software modelers and developers with a higher level language with which they can build systems. The best advice is to forget the implementation and code structures, at least in the beginning, and think about the problem domain. This raises the abstraction most and leads to fundamental productivity improvements. If you focus on the code from the start, the possibilities for automation may be easier to define but the gains are small, 10–30% improvements on current manual practices.

While it makes sense to develop the whole DSM solution at the same time, language definition can be started even when other elements of the DSM solution, or even the target environment for the resulting software, are not yet known. Modelers can then start development work, while others implement the supporting target environment, its libraries, and components along with code generators.

When identifying the modeling concepts, it is of key importance to focus on a narrow application domain and your actual needs for it, knowing that you can change the language when your requirements change. This support for language evolution is essential when it comes to making a choice of what DSM tooling to choose. Good environments allow such evolution, automatically updating all the models created previously with the language, whereas with less mature environments you can end up having to freeze or rebuild your models.

CHAPTER 11

GENERATOR DEFINITION

Building a generator is one of the two main tasks when creating a Domain-Specific Modeling (DSM) solution. The idea of code generation is by no means new: most developers will be familiar with the fixed generators supplied with generic modeling tools, and many readers may well have written their own little scripts or macros to generate repetitive code. A DSM generator is closer to the latter, because it is built by you for your specific purpose, rather than by a tool vendor aiming for the widest possible market. It must however go further than most scripts or macros, since it is intended for automated use by people other than its creator, and its output will not normally be edited—or even checked—by hand.

Although a DSM generator must aspire to higher quality than *ad hoc* scripts and macros, a problem in a DSM generator is less severe than with fixed generators supplied with modeling tools. Unlike these prepackaged generators, you remain in control, and so can fix the problem—on your timetable rather than when it happens to suit the vendor. DSM generators also solve the problems of the “sausage factory” generators, which churn out multiple similar blocks of boilerplate code to be hand edited. Unlike those, a DSM generator can later be changed and all code regenerated: the equivalent of changing your recipe and having all sausages already delivered to shops update automatically!

For most developers, one of the hardest tasks in Domain-Specific Modeling is to *not* build the generator. Why would we want to avoid building the generator: surely that is a vital part of a DSM solution? Absolutely—and it is also one dear to the heart of developers, being close to the code they are used to writing. Thanks to the

overenthusiastic hyping of 1980s and 1990s CASE tools with their fixed modeling languages and “one size fits all” generators, the generator is also the area that many developers are most skeptical about.

These two factors lead many to want to jump in and build a generator right near the start, often hand in hand with developing the modeling language. Unfortunately, such solutions almost invariably turn into languages that describe code rather than things in the problem domain. The result looks like a bad rehash of UML, without even the chance to claim it is a standard. The level of abstraction is not raised, and there is no real way to generate full code from such models.

As we build our modeling language to take us to a higher level of abstraction, there can thus be no premature slipping back down the slope into code. When we have the language firmly established on a higher level of abstraction, it is refreshingly easy to build a path back down to the code level. The information from the models seems to flow down naturally, as if by some gravity of abstraction, into the form needed for the compiler. This ease should come as no surprise: it has always been easier to build a compiler than a reverse compiler.

Developers are often afraid that they will produce a modeling language that would have no sensible, easily performed mapping to code. While in theory this might be possible, in practice any expert who has coded applications in the domain would be most unlikely to do this. They will find themselves making decisions in the modeling language that, in spite of their best efforts, are at least informed by their experience of patterns of code in that domain. While that tendency and experience are of course valuable, it is still worth fighting them down to a reasonable extent. If the modeling language can be made 10% better to use, at the expense of making the creation of the code generator 10% harder, many modelers will benefit over many applications, while only one metamodeler needs to suffer once.

Still, as metamodelers we are all for reducing the suffering of fellow metamodelers where possible! One important way to do that when building generators is to have something concrete to work with: a working mini-application in exactly the format you want to generate. That is what we will look at in the next section, before moving on to the generators themselves.

11.1 “HERE’S ONE I MADE EARLIER”

Before we can build a generator, we need to know what we want to generate. The best way to do that is to have a working example of the output. Remember, DSM is about automating what we would otherwise be doing by hand. Unless we know what we want and are able to do that by hand, there is little hope of teaching somebody else to do it—let alone teaching something as stupid as a computer to do it. The good news is that we only need one example of each part of the output, and the computer learns quickly.

But wait! It is not enough to have just the output: since a generator is a kind of transformation, we must also have the corresponding input. We can either build a model that roughly corresponds to some existing code, or then build a small representative model and corresponding code.

Let's assume that you have no code to start with, so first you want to build a model. Such a model might have one instance of each of the three or four main object types. Sketch out an implementation of that which would work with your current code, and check that the model can give you the information you need. If it is at least close, that will be fine. You can allow yourself some assumptions for now: maybe an object type could map to one of two different kinds of code, but just choose the more common solution for now. Go ahead and write the application and get it to run. Aim to write code similar to current best practice in your applications, but do not waste time striving for perfection or minimalism. The best code at this point is the kind that you could explain easily to a new employee, not the kind that will save two bytes or ten milliseconds because of a clever trick with a certain version of a library. But have no fear: if you can specify when to perform that trick, you can later add a condition to that part of the generator and reap the saving every time it is possible.

Now you have a matching pair of a model and the desired output. The information in the model is distributed over objects, relationships, and properties. The output has the same semantic content, plus some content that is related to the output language syntax, and some fixed content that the generator will add. Clearly, the lion's share of the variability in the output will come from the model: the generator will be the same for all models, and its output for a given model will always be the same, so it cannot add any variability to the output that is not found in the model.

You can thus look at the output and identify which parts of it are fixed text that will always be present, and which parts are simply values from the model. Between these two extremes lies the work for the meat of the generator: parts that are included or left out depending on some information from the model, and parts that are repeated for each of a given structure in the model. These four kinds of parts cover the entirety of most generators, even for the most complex systems.

To cope with all possible generators, we need to add the possibility of massaging the values from the models—normally as a concession to the syntax of the output language and the ease of use of the modeling language. For instance, most languages require that names be composed only of alphanumeric and underscore characters, yet we may want to allow the names in the model to contain spaces. When outputting a name from the model, we may thus need to apply a filter to it that replaces nonalphanumerics with underscores.

A simplified process for building a generator is thus:

- (1) Paste the desired output code as the entire content of the generator.
- (2) Reduce each repeated section in the output code into one occurrence, with a generator loop that visits each model structure for which the section should occur.
- (3) For sections that have one or more alternative forms, surround them with generator code that chooses the correct form based on a condition in the model.
- (4) Replace those parts of the output that match property values in the model with generator code that outputs those property values, filtered as necessary.

In practice, steps 2–4 are often best performed in parallel on each chunk of output in turn.

This section has described the basic process of building a generator, regardless of the kind of model, output language, and language for writing generators. To give more detailed advice on building generators, we first need to know what kind of generator language we have.

11.2 TYPES OF GENERATOR FACILITIES

Czarnecki and Helsen (2003) identified two main approaches for generating code or other text from models: visitor-based and template-based. In practice, there are at least a couple more ways, ranging from simple programmatic access to the models to crawler-based generators outputting multiple streams, and even generators that are themselves the output of other generators.

11.2.1 Programming Language Accessing Models Through an API

The minimal facility necessary for generation is programmatic access to the models. In an open source modeling tool or model repository, the programmer can code generators directly against the model data structures. While initially such direct access may feel welcome, the low-level programming it requires, together with the unpleasantly high coupling between the generator and modeling tool, mean this is normally only used where an organization has hand coded the modeling tool themselves. Even then, as the tool matures the benefits of separating the implementation of modeling and generation become apparent. The need for higher-level commands to read, navigate, and output model structures also quickly leads to a separate generator component.

Direct access to model data structures also forces the generator to be built in the same programming language as the modeling tool and to run in the same memory space. To solve these problems and alleviate those mentioned above, tools often offer an Application Programming Interface (API) for model access. Most early APIs limited the generator to running on the same platform, and indeed the same machine, as the modeling tool, but more recent work removes these restrictions.

An external generator API can be either message-based or data-based. In the former, the model data remains in the modeling tool, and the tool's API only passes pointers or proxies for those objects to the generator. The generator can still be written as if it were operating on local data, but all calls are actually proxied through to the modeling tool, and only further proxies or primitive data types like strings or integers are returned.

In data-based APIs, calling an API function returns the actual data structures, or rather a copy of them, to the generator. This requires the duplication of the data structure types and functions in the generator. It also brings the problem of where to cut the model up: if the generator requests a graph, do we return the whole data structure of the graph, all its objects, all their subgraphs, and so on? If we do not, there must be some method to determine which parts of the data are to be returned in full and which as proxies.

A well-made API of either type will behave similarly to the other type: no long delays and running out of memory in a data-based API, or apparent instant access to all data in a message-based API. Both types are, however, only approximations. What is probably worse, though, is that both only offer standard generic programming languages to read, navigate, and output models.

While there are good programming languages for manipulating complex networks of objects, and good programming languages for transforming one text stream into another text stream, there are few if any generic programming languages well-suited to navigating a complex network of objects and outputting text. Hence the need for the languages and facilities specifically designed for generation, which will be covered in the rest of this section.

11.2.2 Model Visitors and Model-to-Model Transformations

The simplest kind of generator facility is a model visitor, which makes a mapping from structures in the modeling language to structures in the output language. Normally, each type in the modeling language is mapped to an output language structure. For example, a graph type “Watch Application” may be mapped to a class, an object type “State” may be mapped to a method, and another object type “Time Variable” may be mapped to a field.

In some cases, the mapping is focused on the modeling language, with a fair amount of freedom in what each type can map to. The generator visits each element in the model, calling the generator for that element’s type via the Visitor pattern.

In other cases, such as XMF-Mosaic, the mapping is made in three stages. The first stage maps the model elements to an intermediate set of concepts corresponding to a broad kind of programming language—most likely object-oriented languages. The second stage maps each of these generic object-oriented concepts to the corresponding concept in a particular object-oriented language. Finally, the third stage generates the relevant code for that concept in that language. Such an approach would be useful if a company needed to generate the same application in a growing number of object-oriented languages: they could add new second- and third-level mappings as the need for those languages appeared. However, it is limiting if the need is only for one or two languages: the generator must follow the patterns and structures laid down by the tool vendor.

The set of output language structures supported for mappings would normally include the major higher-level structures such as functions, classes, and modules. If strongly linked with a particular Integrated Development Environment (IDE) or IDEs it could also extend up to projects. If linked to a particular kind of language, it could also extend down to control structures. Building a mapping for such extended model visitors would however become increasingly difficult, as the various parts of, say, a C for loop must each be mapped to some structure in the model.

A model visitor can cope reasonably well for simple skeleton code generation from modeling languages like UML into the object-oriented programming languages it represents. For DSM languages, single-stage model visitors normally fall short, unless the output itself is a DSL with a similar semantic structuring to the

models. The basic idea of a model visitor is however useful in all types of generation facilities.

Model-to-Model Transformations Rather than aiming to produce textual output, a generator can also create or alter a model. This still leaves us with a model, which will need a further generator to produce the textual output required by compilers and other existing implementation tools. It is generally a particularly poor idea to create model transformations that produce models that users are still expected to edit, for the same reasons as generated code should not be edited. Some of the issues with editing generated models in the context of MDA were covered in Section 3.3.3.

The value of model-to-model transformations is thus better realized when they form one part of a chain of transformations, resulting eventually in textual output, and whose intermediate stages are invisible to the modeler. The decision of whether to generate in one step from models to text, or in several steps with intermediate transient models, is thus one that can be made by the metamodeler based on the tools available. If for instance the generation tools are not powerful enough to support a mapping from models to the required text in one step, intermediate phases may be useful. Similarly, if there already exist tried and tested transformations from a certain model format to code, it may be useful to translate DSM models to that model format. Particularly where these transformations cross tool boundaries, however, there is a danger of information being lost or twisted along the way, like a game of Chinese Whispers.

11.2.3 Output Template

An output template consists of the code you want as an output, but with parts that vary based on the model replaced by commands surrounded by some escape character sequence. This approach is familiar from web pages built dynamically on the client side with HTML containing JavaScript commands, or on the server side with PHP or ASP.

Template-based generators are probably the most common kind, and one of the oldest. Examples include JET (Java Emitter Templates), the Microsoft DSL Tools T4 engine, and the CodeWorker scripting language.

JET operates on input data consisting of a single Java object, and its command sequences are written in normal Java. It can only be run as part of Eclipse and is only designed for outputting Java classes. It uses `<%` and `%>` to escape its command sequences—the escape characters can be configured—and also `<%=` and `%>` to delimit a Java expression whose result is appended directly to the output. JET is run in two phases: translation and generation. The first phase translates the template to a Java program that will perform the actions specified in the template. The second phase invokes this program on the input to generate the output. The name of the file to be generated is specified in the first JET tag.

T4 operates on input data consisting of a single C# object, normally a DSL Tools model, and its commands are written in C#. It uses `<#` and `#>` to escape its command sequences, `<#=` and `#>` for direct output, and `<#+` and `#>` to escape whole functions or other class features.

The CodeWorker scripting language operates on input data consisting of a tree data structure, formed from the parse tree produced by a CodeWorker BNF template from textual input. It uses `<%` or `@` to delimit its commands, which are in CodeWorker's own textual DSL. As CodeWorker uses its own language, it also has to provide its own library of basic functions for manipulating strings, numbers, files, and so on.

One problem with an output template is that it only allows output to one file or stream. The structure and order of the generation is thus forced to follow that of each output file, with one generator per output file. This is inefficient in cases where the information needed for more than one file can be found in the same place in the models. The simplest case of this is `.h` and `.c(pp)` files in C(++) generation, but other examples abound even where the output language is not so quaintly repetitive.

For instance, for each source code file we output, we often want to add a line to a makefile. It is most convenient to be able to append that line to the make file at the point where we start to generate the corresponding source code file. Using output templates, we would be forced to duplicate the navigation code that brought us to the appropriate points in the model.

11.2.4 Crawler: Model Navigation and Output Streams

A DSM generator has two tasks: to navigate and read its way around the model, and to output text based on the model. Model visitors require the modeling language elements to match the output format elements. Output templates force the navigation to be sub-ordinate to the output format. A third kind of generator, a crawler, gives more freedom, while still allowing these two simpler kinds of generation where they are possible.

Where an output template consists primarily of fixed text, interspersed with escaped command sequences, a crawler consists primarily of command sequences, with fixed text escaped simply by placing it in string quotes. At its simplest, then, a crawler looks like an output template, but with the escaping turned inside out.

The generator language in a crawler will normally be a textual Domain-Specific Language, as is the case in say MERL, the MetaEdit+ Reporting Language. On the borders between true generator languages and general purpose languages used for generation are languages like DOME's Alter, which is an adaptation of Scheme. The benefits of a true DSL are similar to those of DSM itself: conciseness, a higher level of abstraction, and fewer errors.

A crawler maintains and operates on two stacks as it generates: one for model element navigation history and another for output streams. At any time, then, there is a current model element and a current output stream. This makes for a concise language: rather than have `"fileWriter.write(this.name())"` we simply have `"name."` The details of which element's name we want and which file we want to write it to are easily inferred from the stacks.

Similarly, crawlers can free us from error-prone explicit pushing and popping, or assigning iterator variables, which would otherwise take up a large part of the

generator text. Each navigation operation works like a loop, with an implicit push at the start and pop at the end.

For instance, if we ask the crawler to navigate from the current State to each of the States directly reachable from there, the top element on the stack will initially be the current state. As we process each reachable State, it will be pushed on the stack, then the output for it will be performed. Afterwards it will be popped off the stack, and we will move on to repeat the loop for the next reachable State.

Similarly, if we are generating a number of C files and a makefile for them, we might start with the makefile on the top of the output stack. We could then navigate to each model element from which a C file will be generated. For each element, we would output a line to the makefile for that element's C file, then push a new output stream directed to that C file. With that on the top of the stack, we would perform the output for the contents of the C file, then pop it off the stack before moving on to the next model element.

Using a DSL for the commands can give a major advantage over a standard programming language. This is especially so in the area of navigation, which takes up the bulk of the generator. Listing 11.1 is an example of a standard programming language, C#, used in a T4 template from the Microsoft DSL Tools forum (apparently an earlier version, since it uses <% rather than <#).

Listing 11.1 Generator written in a generic programming language.

```
<h2>ConceptBs</h2>
<%
foreach ( ConceptB b in this.ConceptA.Bs )
{
%>
<h3>Name: <%=b.Name%></h3>
<h3>Outgoing links:</h3>
<%
    MetaRoleInfo roleInfo = b.Store.MetaDataDirectory.
FindMetaRole (BReferencesB.ReferringBsMetaRoleGuid);
    if (roleInfo != null)
    {
        foreach (object link in b.GetElementLinks(roleInfo))
        {
            BReferencesB relationship = link as BReferencesB;
            if (relationship != null)
            {
%>
<p>Refers to: <%#relationship.ReferencedB.Name%></p>
<%
            }
        }
    }
%>
```

And here in Listing 11.2 is the same thing using a DSL, in the crawler-based MetaEdit+ Reporting Language.

Listing 11.2 Generator written in a language made for generating.

```
'<h2>ConceptBs</h2>
foreach .ConceptB
{
    '<h3>Name:'  :Name  '</h3>'
    '<h3>Outgoing links:</h3>'
    do ~From~To.ConceptB
    {
        '<p>Refers to:'  :Name  '</p>'
    }
}
```

The key difference here is in the line in the middle, “do ~From~To.ConceptB.” Starting from the outer foreach loop’s current ConceptB, it says to crawl along any From role and its To role into the next ConceptB. That one line replaces twelve lines from the version written in C# (everything between “Outgoing links” and “Refers to”).

This kind of pattern is very common in any code generation or reporting on a model. Those using standard programming languages as generators will thus quickly find their generators full of blocks of code similar to the above. Since DSM is intended to save developers from precisely this kind of unproductive code duplication, this is more than slightly ironic: the old adage of the shoemaker’s children comes inescapably to mind.

11.2.5 Generator Generators

DSM people tend to be happy with the idea of doing something on a meta level; some even get excited about it. For such people, the idea of a generator generator is positively intoxicating. Rather than drudge to build a generator that turns models into code, why not write a smarter program that would build the generator for us? This can be particularly attractive if the generator language is somewhat lacking in facilities for handling repeated similar cases: subgenerators, parameterization, reflection, refactoring tools, and so on.

The overhead in building a generator generator is however substantial. First there is the additional cognitive load of an extra meta level, and of an extra layer in which bugs can hide. Second comes the difficulty of three levels of embedded syntax: simple strings to be generated are first quoted for the actual code generator, and then the generator script containing those is quoted for inclusion in the generator generator. This situation is made worse when the three languages—those of the generator generator, the generator, and the output—all use similar punctuation and keywords. In the worst case, they might even be exactly the same language: at that point, nobody can read or write without constantly losing track of which level a particular keyword or punctuation character is on.

Finally, generator generators introduce new issues of version control. Unless the generated generators are created anew for each run, they should be versioned and

mechanisms put in place to make sure all users use the correct version, and all generated code mentions the version used. If the generated generators are processed by a different facility from the normal facility—an external compiler or interpreter, say—the version of that tool must also be recorded.

These drawbacks are unlikely to deter the kind of person who is attracted by meta solutions, compilers that compile themselves, modeling languages that can be used to model themselves, and tools that were built in themselves. While the particular gene or brain chemical responsible for this attraction is yet to be identified, two things are certain. First, the effects of it are very strong and second, there is at least some correlation between its level and how smart a person is. Maybe the smartest ones are those who are pragmatic enough to recognize when not to try this approach, but few would claim they feel no attraction.

While the purest form of this pursuit is generators that produce generators in the same language, it has also been used to good effect in integrating external generation tools. The first generator exports the models in some easily digested format, along with commands to an existing external generator. The existing generator parses that format and applies those commands on it to generate the output. The commands can vary greatly in their complexity. At one extreme, they can be a simple parameterization of a mostly fixed generator, in which case the model output format must be one which that generator can read. At the other extreme, they will be a full program to run on the model output format, in which case the external generator will be a compiler or interpreter that runs that program.

It is also possible to combine both models and commands into the same file. Here the model is represented directly as data structure initializers in the language of the external generator. At the opposite end of the scale, the situation can also result in models and commands being in similar formats, for instance if the models are exported as XML and the commands in the form of an XSLT transformation, itself an XML file.

An interesting variation on generator generators occurs where the second stage of generation is deferred until the runtime of the end system. As the system is started up, it generates some parts of itself. Clearly, this requires the use of a language that is sufficiently dynamic and has good support for reflection. It also brings us neatly to our next topic: looking at various patterns of code that are often found in generated solutions.

11.3 GENERATOR OUTPUT PATTERNS

Generators produce a wide variety of different kinds of output: simple text, documentation, .ini files, XML, database definitions, and of course various kinds of code. For generation to be possible, the output required from a given generator over different models must exhibit patterns. If there is nothing an expert human can recognize as a pattern common to several examples of the kind of output you want, there is no way to generate it. Fortunately, even a glance at the kind of code most of us are forced to write these days shows there is plenty of scope for removing duplication.

Even after removing duplication, a trained eye can spot many cases where textbook patterns have been applied, particularly in well-written code.

In a similar way, there are patterns to use when writing generators. Some kinds of tasks seem to crop up often in DSM generators, and knowing a good basic solution can save significant time.

11.3.1 Simple Text

Simple text files such as configuration files or script files are generally easy to generate. They tend to fall into three categories:

- (1) Largely boilerplate
- (2) Configuration and settings files
- (3) Script files

The easiest are of course files that consist largely of boilerplate text, where the majority of the text remains the same independent of the model contents. Any kind of generator will cope well with these, and the template-based generators are at their strongest here.

Models that generate configuration and settings files tend to use a simple mapping between the structures in the file and those in the model. For instance, sections in an ini file may correspond to object types, with each setting and value corresponding to a property slot in that object. The crawler-based approach in Listing 11.3 would generate such a file, treating empty properties as meaning they should not be generated, and thus whatever reads the resulting file will use a default value for them.

Listing 11.3 Crawler-based generator for .ini files.

```
foreach .() /* iterate over objects, expecting one instance
per type */
{
    '[' type ']' newline
    do :() /* iterate over all properties of the object */
    {
        if not id='' then
            type '=' id newline
        endif
    }
    newline
}
```

One thing that can be difficult in configuration files is that the output language, and the program that processes it, tend to have little intelligence. As the values in the file tend to be expressed in a machine-friendly format, this can require some translation from the human-friendly format used in the models. For instance, the models may specify numbers in decimal notation, but the configuration file may require them in

hexadecimal. If this situation had occurred when generating normal source code, the values would probably have been output in decimal, even if earlier handwritten code had used hexadecimal: the compiler can read either with no difficulty. Now, however, either the generator language must include commands to perform the conversion, or then a separate postprocessing step must be applied to the generated file.

This is thus a strong area for generator languages that use a generic programming language for commands, as such languages will generally already have a library function for such a conversion. Domain-specific generator languages may not fare so well: even Codeworker, with its built-in library of over 200 functions, has no direct answer for this, although it can handle some particular cases. All is not lost, however, as the necessary transformation can always be written as a postprocessing step in a generic programming language. If there is no great number of such cases, they can also be made as in-line calls from the generator to such a program, making the generators slower to run but easier to understand.

Script files, such as Windows batch files or Unix shell scripts, are also generally easy to generate. On the positive side, the interpreters for these files are smarter than those for configuration files. Transformations can thus often be output as the instructions to perform the transformation, rather than require the generator to perform the calculation and output just the answer. In some cases, this can also make the generated output easier to understand: rather than contain some mystical number, with no clear relation to the model, the output shows the calculation from the model values. Even batch files are capable of this, as shown in Listing 11.4.

Listing 11.4 Using batch files for simple arithmetic.

```
C:\>set FahrTemp = 0
C:\>set /A KelvinTemp = (FahrTemp-32) * 5 / 9 + 273
256
```

But there are surprises in store for the unwary, for instance if you try to bit-shift the above result using the `>>` operator, as in Listing 11.5.

Listing 11.5 Pushing the limits of arithmetic in batch files.

```
C:\>set /A KelvinTempHighByte = KelvinTemp >> 8
"Appends the result to the file called 8, so have to quote:"
C:\>set /A KelvinTempHighByte = "KelvinTemp >> 8"
1
C:\>set /A KelvinTempLowByte = "KelvinTemp % (1<<8)"
0
```

And if you put these in a batch file, you will notice that the result of the SET operation is no longer sent to standard out, and the modulus operator `%` needs to be doubled to avoid being interpreted as a batch file argument. . . Still, one of the joys of generation is learning all the little tricks and foibles of the language you are outputting!

11.3.2 Model Checking

The modeling language is the best place for rules that should always hold, and the DSM tool should offer most of the necessary types of checks that you can customize or parameterize. Where the DSM tool does not offer the checks you need, it is also possible to write a generator to check some property of a model and give feedback to the modeler.

In most aspects the generation of model checking reports is similar to the configuration files mentioned in the previous section. One difference is that the output is intended for display to the modeler, rather than for reading by a program, and for use at design time, rather than compile or run time. These factors favor the integrated generator facilities, which can then display the results of the checking as live links to the model elements. For instance in MetaEdit+ clicking on the name of an object in the generator output will take you to that object in the model, allowing you to correct the problem reported.

It may be possible to achieve something similar with other generator facilities if the checking report is output to HTML, and it is possible to make a hyperlink invoke the modeling tool to show a specific object. Some IDE-based DSM tools may also offer special integration of model checking reports, automatically running them when saving a model, and showing the results in a separate pane similar to that used for compiler errors. For instance, Microsoft's DSL Tools take this approach, although admittedly such model checking scripts are programmed in C# or using the GAT framework, rather than the T4 template language used for other generators.

Another aspect where model checking reports differ from other generators is that they often want to check the existence of a certain kind of item, or the number of those items. The DSM tool's modeling language facilities often support rules to check upper bounds: for example, a Start state may have at most one transition leaving it. Checking lower bounds is however harder, since such a rule could not be checked all the time: adding a Start state to an empty model would be illegal, making it impossible to get started on the model. Lower bound checks are thus often left to checking reports or generators, which the user can run when desired.

Overenthusiastic or bossy metamodelers may be tempted to make such checks compulsory, but that only invites the wrath of modelers when they need to save their work and get out of the office quickly. Such solutions invariably cause more problems than they solve: to pass the checks, the modeler will simply add fake elements to the model. Sometimes those elements will be overlooked the next time, and since no checking report will reveal them, they will remain until your customer finds them for you.

More specialized model verification and validation tasks may be better performed in existing external programs. For example, tools exist for validating that a system, expressed as a state machine, cannot exhibit various kinds of undesirable behavior. A generator could be used to export a state-like model into a format readable by such a tool, and then call the tool. Attempting to analyze the behavior directly with the generator facility would be a poor idea: generator facilities are not renowned for their speed, and these calculations are computation intensive.

11.3.3 Documentation

In DSM, the models form the best documentation of the system: expressed in high-level terms, yet precise and always up-to-date. The DSM tool also provides the best environment for accessing this documentation, supporting various views, browsers, and queries. However, not everybody will have access to the DSM tool, or be familiar with it, and in any case a strong tradition for paper, Word, or web-based documentation is something that many organizations will find takes time to overcome.

Around the time a DSM solution is being deployed to a pilot group, the first requests for documentation appear. While these are often for documentation of the modeling language, in many tools the modeling language is treated as just another model, and so there is a requirement to generate some traditional documentation format from a model. Certainly by the time the DSM solution is being deployed to a wider group, there will be a desire for producing traditional documentation from the DSM models themselves.

The requirement for modeling language documentation is probably objectively more important for project success. However, making models too available in a traditional documentation format makes management happier, and the resulting sponsorship is an even better determiner for the continuation of the project. We may not like it, but we would be unwise to ignore it: the alternative is to teach the manager to use the DSM tool, but that could be far more dangerous...

Plain text is rarely sufficient to qualify as a traditional documentation format: at least basic character formatting is required. Going beyond the lowest common denominator of plain text poses a problem. It is not sufficient to know that we want the first line to be bold and a larger font, we also need to pick a particular editor or viewer for this formatted or rich text. The choice of tool is influenced by many factors, including:

- who will use the documentation;
- in what environment and form;
- what tools work there or are already present; and
- what file formats the tool supports.

The last factor is of course vital for generation, since the most sensible way to output documentation from models is as a plain text file with syntax to express formatting. While it is possible to build generators outputting Microsoft Word's native binary format, and indeed that would be a fun—if somewhat masochistic—challenge, those electing to output plain text RTF need not feel they are setting the bar too low for themselves. (Sadly!)

Viable candidates for output formats include RTF, Microsoft Office open XML format, SGML, HTML, and LaTeX. These can be used directly, or then fed to an appropriate tool to generate Word DOC files, PDF files, Windows Help files, and so on. Also, taking a higher-level view for a second, we can note that the domain these

various languages aim to cover is essentially the same. There are good possibilities to convert documents from one of these formats to another, or produce other similar formats such as DocBook, Texinfo, and groff.

Despite the plethora of options, most projects make a simple choice: if they want to view their documents on a computer, they use HTML; if they want to print them, they use RTF. In both cases, there are three important areas in addition to the text: images, styles, and scripts.

Images Images of the models are of course a key part of the documentation. The generation language must offer facilities for outputting models to separate image files, and making the generated documentation refer to those files. Image files can be either bitmaps, which are generally best for computer viewing, or vector graphics, which are better suited to printing or print-like formats such as PDF.

For bitmap files of models, compression artifacts make JPEG a nonstarter. While the 256 color palette in GIF files is not a problem in most modeling languages, it becomes a limitation if the symbols contain photographic bitmaps or fountain fills. Perhaps the best format is Portable Network Graphics (PNG), since it offers small file size, lossless compression, and a 24-bit color palette. Until recently PNG support in other software such as browsers was weak, but now almost all support the PNG features needed for models.

Up until a few years ago, there was no satisfactory, platform-independent format for vector graphics. PostScript files, and for stand-alone images Encapsulated PostScript, was one possibility, but application support was poor. Most applications could not show the vector content of an EPS file, which would only appear when interpreted by a PostScript printer, but instead only displayed the bitmap thumbnail (if the generator had included one). Windows Metafiles, both WMF and the enhanced EMF, were only supported on Windows. The ancient Macintosh Paint program's format, PICT, became rather surprisingly one of the better choices. It was relatively simple in its capabilities, but sufficient for the needs of modeling languages, supported 24-bit colors, and could be read in the Macintosh, Windows, and Unix worlds.

The advent of XML saw the introduction in 1998 of two competing XML vector graphics formats, Microsoft's Vector Markup Language (VML) and Adobe's Precision Graphics Markup Language (PGML). For a while it looked like the situation would develop into a typical format war, where users are losers. Fortunately both sides, and their supporters, agreed to come together to work on Scalable Vector Graphics (SVG) (how refreshing to have an XML language that does not end in ML!). While Microsoft are still dragging their feet a little in browser support, Internet Explorer can view it with the Adobe plug-in, and Mozilla Firefox and Opera both support SVG natively.

An important part of viewing model images on a computer is the ability for the elements in the models to work as hyperlinks. This allows the user to click on an object in the image, and jump to the part of the documentation specifying that object. In HTML, this can be accomplished by generating an image map; in SVG, the elements themselves can have `<a href...>` tags. It should even be possible in RTF: invisible

drawing objects could be overlaid on a bitmap picture and given hyperlinks with `\hlloc`.

Styles The spirit of DSM almost forbids replicating formatting information throughout a generated document. Rather than define each header to be 14 point bold italic Arial with 12 point spacing above and below, the generated document simply contains the fact that this is a level one heading. The actual styling information is moved to a separate file: CSS for HTML, or a template for Word. These files would be handwritten once along with the modeling language, not generated each time.

Splitting the information from its representation in this way also allows one document generator to produce output for multiple purposes. For instance, there may be one CSS file that specifies fonts and layouts suitable for on-screen viewing, and a second CSS file with formatting better suited to hard copy: no hyperlink highlights, no sidebars, and so on.

Not specifying formatting in the generated output also makes it easier to build the generator. A similar effect could of course be obtained with a set of subgenerators to specify the formatting for each style. That tends to lead to rather baroque looking generators that spend more time calling subgenerators than outputting anything useful. The resulting documentation files are also fixed to a specific set of formatting: with a separate stylesheet or template, even old documentation outputs can be viewed with the latest styles.

Scripts Earlier we talked about how more intelligent output formats for plain text make the life of the generator easier. Rather than have to write commands in the generator to produce exactly the required output, the output can consist in part of commands to whatever application will read it. Similar possibilities exist for documentation output.

For output that will be read by a word processor, for example RTF for Word, it is possible to write macros in the Word template that will postprocess the output. One important aim for this in many organizations is to turn the RTF file into an honest-to-goodness native Word .DOC file. While there seems no real gain in this, “standard practice” is the Newton’s First Law of software organizations: “every object tends to remain in the same state unless an external force is applied to it.”

More usefully, Word macros can be used to build tables of contents and figures, along with paginating to fit the current default printer. They can also replace links to external image files with embedded copies of those files, making a more coherent single document for distribution and archival. Where formatting cannot be simply expressed as template styles, for example, table formatting with a different background color for alternate rows, a Word macro can easily apply the same Table Autoformat to all tables in the document. While Word’s security model is nowadays by default wary of macros in templates, an organization can cryptographically sign the templates to make them accepted within the organization. Macros can also be used to make the documents live: for example,

a button in the document could reopen the model in the modeling tool, either the latest version or the exact same version as in the picture, retrieved from the version control system.

In HTML, scripting is generally less necessary for building the documents. Unlike with Word, the documents tend to be distributed and archived exactly as they were generated. Where necessary, Javascript can of course access any part of the document with Document Object Model (DOM) calls, and most modern browsers support a wide range of operations on the document: parts can be hidden, revealed, replaced, reformatted, and so on. Scripting can also be used to good effect to add dynamic display of information: for instance, a list of an object's properties can be shown when the mouse hovers over that object in an image.

There are thus many possibilities for beautifying and enhancing your documentation. Since there is however no way to increase the amount of information contained in the models, it is probably wise to think of the models themselves as the real documentation. Rather than try to impress the boss with cool tool tips or professional looking document layout, it may be better to concentrate your efforts on the modeling language and code generators. After all, those will be used far more: nobody ever really reads documentation, do they? And with that rhetorical question, let us move on to another ostensibly human-readable output format: XML.

11.3.4 XML

Generating simple XML is beautiful in its simplicity. Generating watertight XML is an ugly mess. Generating real-world XML is somewhere in between, but sadly the pain increases the more you want to move the level of abstraction of the modeling language above that of the XML files.

Let's start at the easy end of the scale: imagine we have a domain-specific XML format in which developers currently have to write files by hand. We create a modeling language based on a simple but often effective pattern: major elements map to graphs, minor elements to objects, and attributes to properties. A generator for any such modeling language might look like Listing 11.6.

Listing 11.6 Simple XML generator.

```
'<' type '>' newline
foreach .()
{
    '<' type
    do :()
    {
        ' ' type '=' id ''
    }
    '/>' newline
}
'</' type '>' newline
```

The major element is named after the graph type and is opened at the start of the generation and closed at the end. We iterate over each object, outputting an element named after its type, and within that element we iterate over the properties of that object, outputting them as attributes. The generator is not even fragile with respect to changes in the modeling language: we can add object types and change property names, and the generator will still output XML. So far so good.

Element Names But what if the name of an object type consists of two words separated by a space? Then an XML parser reading the result will only parse the first word as the name of the element, and choke when it tries to parse the second word as an attribute name. We thus need to process the names of elements, filtering out any spaces and perhaps other characters that the XML parser would not like. So we take a quick look at the XML specification in the language of our choice... or more likely a rather longer look at the closest alternative (www.w3.org currently lists 16 translations, including Estonian and Interlingua, but “E-Z Read English” seems to be missing).

To cut a long story short, the XML specification for element names is a mess. It may be a very well founded attempt to break the stranglehold of US-ASCII alphanumerics on the ways we are allowed to name things, but from our point of view as we build a generator, it is a mess. If we take just the first character of a name, which has the smallest set of legal characters, it can be a “Letter,” an underscore or a colon. A “Letter” can be a BaseChar or an Ideographic. A BaseChar is any member of over 200 Unicode 2.0 character classes such as [#x0041 – #x005A]: that is the first class, and perhaps more commonly expressed as [A–Z]. At this point the value of a pre-existing XML schema rises appreciably: let the schema determine the object type names, rather than the other way round. This also saves us from the next gotcha: a name may not begin with the characters “xml”, in any mixture of case: those names are reserved.

While accepting an XML schema as given saves us from the task of mapping names in the metamodel to a form acceptable to XML parsers, our problems are only just beginning. After all, we were only planning on having one or two metamodels, but of course many more models. And while we could perhaps have envisaged our metamodelers remembering to only ever give names that were legal in an XML context, that may be somewhat optimistic when it comes to modelers. We therefore come to our next task, outputting property values as XML attribute values.

Attribute Values Even a casual acquaintance with XML will prepare us for the fact that attribute values are quoted. As eternal optimists, we assume that we will thus have to escape or double any quote marks in the value, and then the problem will be solved. After all, if there is only one character out of the whole Unicode gamut that can close an attribute value, surely the parser just reads all characters up to there and then stops?

That, however, would be tantamount to anarchy, and the World Wide Web Consortium, as a standards body, is firmly opposed to anarchy. Standards bodies like to make sure that everybody confirms to a rigidly defined standard format, and the

process of making sure of this has a name: normalization. Here is how XML attribute values are normalized, for those who are still innocents:

- Line feeds, carriage returns, and tabs in the value are replaced with spaces.
- Character references such as 	 (tab) and € (the Euro sign) are converted.
- Entity references such as & (&) and < (<) are converted, and the result of the reference is recursively converted.

Since one of the first problems encountered when using XML to contain values from models is its tendency to mess with white space, we should make sure we know exactly what happens and how to avoid it. The specification states the following—translation is left as an exercise for the reader:

Note that if the unnormalized attribute value contains a character reference to a white space character other than space (#x20), the normalized value contains the referenced character itself (#xD, #xA or #x9). This contrasts with the case where the unnormalized value contains a white space character (not a reference), which is replaced with a space character (#x20) in the normalized value and also contrasts with the case where the unnormalized value contains an entity reference whose replacement text contains a white space character; being recursively processed, the white space character is replaced with a space character (#x20) in the normalized value.

Back to our original question about attribute values: how do we cope with a quote character in a value? The specification allows strings to be quoted either with single quotes or double quotes, but this is little help in our case. Even if we made our generator check each value to see which way to quote it, we would still eventually find a value containing both kinds of quote. Best is to stick with the more commonly found double quotes around the value, and escape any double quotes in the value with the " entity reference.

For the rest, we need to disguise the white space characters the W3C normalization sniper is looking to pick off, and any characters on which the parser would otherwise choke. We also need to check the values for anything that the XML parser is looking to help us out by transforming, such as entity or character references. For example, a property might contain a string “this function maps ampersand to &”. If that were placed directly as an attribute value, the XML parser would read “this function maps ampersand to &”.

Since all character and entity references begin with ampersands, any ampersands in the input must be disguised. We can disguise them as the entity reference & making our example above “map ampersand to &amp;”. While entity references are expanded recursively, the recursion only applies to the replacement text of the entity; otherwise &amp; would first become & and then just & again.

Since all our other disguises will also use the & character, we had better disguise any real & first as character references. For some reason, the convention seems to be to use a decimal character reference for &, hexadecimal for white space, and named

TABLE 11.1 Escaping Special Characters in XML

Input	Output		
	Entity Reference	Decimal Character Reference	Hexadecimal Character Reference
&	&	&	&x26;
<	<	<	&x3C;
>	>	>	&x3E;
"	"	"	&x22;
[tab]				&x9;
[line feed]		
	&xA;
[carriage return]			&xD;

entity references for the other characters. These are thus given in bold in Table 11.1, but any of the given forms may be used.

Text Elements Another way to store values in an XML file is of course as text between an element start tag and end tag: `<tag>like this</tag>`. In simple text sections like this the above mappings are also valid, although there is no need to map quote marks. White space is however at risk here too: how much depends on a variety of factors. While parsers are not allowed to attack it, applications are expected by default to normalize it. They can normalize it in any way that pleases them, but something similar to the above attribute value normalization is likely. Adding an attribute `xml:space="preserve"` is one way to tell applications to keep their hands off text in that element and its subelements. Even then, your spaces are not necessarily safe: for example, the SVG specification states that with `xml:space="preserve"` all white space characters are mapped to spaces, and leading and trailing white space is removed. And remember that this is just the specification: individual SVG applications may not necessarily follow it perfectly. Ah, the joys of having a simple standard like XML that guarantees interoperability!

XML also has a way to mark text sections so they would ostensibly be completely spared the attentions of parsers' normalizers and transformations. The text section must be enclosed in a CDATA section by surrounding it with the underlined text below:

```
<tag><! [CDATA[like this]]></tag>
```

Within a CDATA section, & and < may occur as normal: there is no need to escape them, nor indeed are any of the normal character or entity references even recognised. The only character sequence that cannot occur inside a CDATA section is `]]>`. One way to get around this problem is to split input containing that character sequence into two CDATA sections. “You cannot use `]]>` inside here” becomes

```
<! [CDATA[You cannot use _]]]><! [CDATA[_>inside here]]>
```

CDATA sections unfortunately also still mangle carriage returns: that normalization happens before the document is even parsed. Any carriage return or “carriage return + line feed” pair is mapped to a single line feed character. Sadly, there seems to be no solution here, since CDATA sections cannot protect their line breaks from the normalizer by disguising them as numeric character references.

Encoding Finally, as with any file output, there is the problem of character encoding. If your XML file will only be processed on the computer where it was generated, you may hope things would just work, and indeed in this case they just might. If you hope that your file would only ever contain good honest 7-bit ASCII characters, you will almost certainly be disappointed: aside from exotic foreign characters and exciting new currency symbols, somebody is bound to paste some “smart quotes” from Word into a model, opening Pandora’s box.

No single choice of encoding can be recommended over all others, because your needs will vary according to the language used in text in models, the operating system and byte order on the computers used to generate XML from the models and to parse that XML, and the tools used to transmit, process, and parse that XML. The only encodings that all XML parsers must recognize, however, are UTF-8 and UTF-16, so if you have no strong reasons to choose something else you should try one of those:

```
<?xml version="1.0" encoding="UTF-8"?>
```

11.3.5 Flow Machine

Now we have looked at plain text, formatted text, and structured text, we can move on to the more meaty subject of generating program code. The simplest kind of program is a linear sequence of instructions, so let us begin there. While such simple programs are rarely a major part of a DSM solution, almost all DSM solutions generating programs will include command sequences in some form.

Sequential In Chapter 7, we saw an example of a generator for simple sequential code in a microcontroller application. The modeling language showed the commands as objects, chained together into sequences with relationships. Perhaps surprisingly, building a generator for such cases requires a little thought.

Most generators work iteratively, often in a tree structure determined ahead of time by the modeling language rather than the model. An iterative approach will not work here, at least not in a DSL-based crawler generator. The smart iteration operators in such a generator are too specialized: what we would really want is a lower-level do-while block. This would print out the command for the current object, traverse to the next object, and repeat as long as that next object existed. In an API-based generator, it would look something like Listing 11.7.

Listing 11.7 API-based generator for sequential code.

```
do {
    System.out.println(currObj.Command());
    currObj = currObj.nextObject();
} while (currObj != null)
```

With a crawler we can use a recursive approach, as shown in Listing 11.8. We output the command for the current object, then traverse any outgoing relationship to the next object and recurse.

Listing 11.8 Crawler-based generator for sequential code.

```
Report 'handleObject'
:Command           /* output for this object */
do ~From~To.()    /* traverse to next object(s) */
{
    subreport 'handleObject' run /* recurse */
}
```

These approaches will work providing there are no cycles: in simple languages like this, cycles may well be forbidden by the modeling language anyway. A benefit of this simplicity is the minimal program size and execution time of the resulting code: it really is just a sequential list of commands.

Conditions and Jumps To be useful in practice, most programs must have more than sequential execution: we need conditions and jumps. At its simplest, this means the modeling language equivalent of IF and GOTO—and indeed if we are generating assembly language, that may be all we can use in the output.

In models, IF can be represented either as an object, from which True and False relationships exit, or else as an n-ary relationship. It can also be extended to a switch-case or ELSEIF structure, with one exiting relationship or role for each choice. If the control paths rejoin after being divided by the IF, as in the top half of Fig. 11.1, a naïve recursive generator will produce poor results. The resulting code will work, but will duplicate the rest of the program for both branches. In cases more complicated than the figure, even maintaining a list of visited objects will not be sufficient to avoid problems in a recursive generator.

A recursive generator can be made to work with IF structures by changing the modeling language. Rather than requiring both control paths to rejoin to the main control path, we can model the branches in an IF structure as subordinate to the IF statement. An example of this is shown in the bottom half of Fig. 11.1. The content of the “Foreign?” conditional statement includes its Yes and No branches, each of which is followed by its own new recursion. The next statement after the conditional is the Mail action, which the main sequential program recurses to. The resulting structure is thus more similar to how a compiler might parse the program: at the top level there are

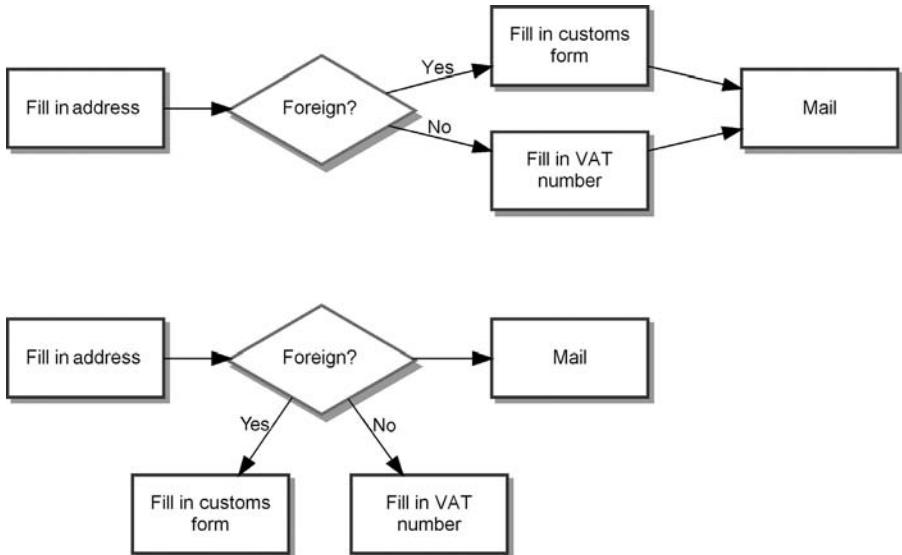


FIGURE 11.1 Rejoining versus subordinate condition

three instructions, the second of which contains its own subtree. A generator might look something like Listing 11.9.

Listing 11.9 Generating subordinate conditionals.

```

Report 'handleObject'
if type = 'IF' then          /* conditional object */
  'IF' :Condition 'THEN'
  do ~From>Yes~To.()        /* new recursion down Yes path */
  {
    subreport 'handleObject' run
  }
  'ELSE'
  do ~From>No~To.()        /* new recursion down No path */
  {
    subreport 'handleObject' run
  }
  'END IF'
else
  :Command                  /* simple object */
endif
do ~From~To.()                /* main flow recursion */
{
  subreport 'handleObject' run
}
  
```

As we will see below, we are not forced to change the modeling language to cope with IF: the other option is to change the generator structure. Before we look at that, we ought to consider GOTO. This may seem a terrible regression: surely DSM is meant to be about best practices, yet here we are talking about a structure that has been frowned on for nearly 40 years. Why then is GOTO not considered harmful in modeling languages? One main reason is that the path of control is far clearer in a graphical model: we can see a line from each GOTO call site to the code it calls. Contrast this with textual programming languages, where the reader may have to scan a large text for labels and compare each label to the GOTO argument. Interestingly, this provides an argument for the use of line numbers in code, both on each line and as the GOTO argument: while the semantics of the call become less clear, the target of the call is easier to find. Even that only helps in showing where the call goes to; graphical models also show all places that call a piece of code.

There is also a second reason why GOTO is not as bad in models as in code. While we can use GOTO to make spaghetti models as well as spaghetti code, in the models the spaghetti is instantly visible. The instant readability of the control paths warns the developer early if things are going too far.

A simple sequential flow is itself a kind of GOTO: in a program, the program counter is set to point to the next statement, just as it is set with a new value in a GOTO command. In a model, the sequential flow is shown by an arrow, and the same arrow can be used for a GOTO. The use of a GOTO is thus only really of interest in models when the target is something other than the next object: in other words, when the target is an object that can already be reached by another path.

Where paths join in this way, a recursive generator will encounter the same problem as in IF statements above. In this case, the solution is simpler and was already seen in Chapter 7. We can make a Label object connected to the start of the chain of commands we want to jump to, and connect our GOTO to that object. Since GOTO can only be connected to a Label object, we do not need to follow GOTOS recursively. If we also forbid normal sequential flow to Labels, we can generate the program by starting at all Labels that have no incoming relationships, and generating recursively from each. Where we would have liked a Label to be part of the sequential flow, we simply move it out and make that sequence call it via a GOTO.

Even this approach will only take us so far, just as it only took early programmers so far. As programs grew, they found that they needed more advanced constructs. However, such growth is by no means a certainty in DSM, and may be a symptom of trying to build a modeling language that mimics program language constructs, rather than focusing on the problem domain. If the growth does occur, the next logical step is GOSUB: GOTO with a return back to the same place. And following hard on GOSUB's heels are its more familiar and powerful implementations: functions.

Functions The problems of sequential generation, whether iterative or recursive, are largely caused by the same object being reachable by more than one path. While

maintaining some sort of collection of already generated objects may help in some cases, it does not solve the difficulty of finding where two complicated branches of an IF statement merge back into the main flow. If we want our modeling language to remain more like a flow chart than an abstract syntax tree, we need our generator to move beyond sequential lines of code.

The simplest approach is to map each object into its own function. The function will take care of the actual actions of that object, plus the specification of which function to call next, possibly depending on a condition. In a Visual Programming Language (VPL), this approach would generate large amounts of functions, each with only two or three simple lines of code. In a Domain-Specific Modeling language, one object represents more than just one atomic code operation, so the functions will each accomplish far more. Of course, we will often be able to spot patterns within the function bodies, and abstract those out into the domain framework, leaving just a simple parameterized component invocation behind. While the remaining function bodies may be only two or three lines long, they clearly accomplish far more than the atomic operations found in a generic VPL.

In simpler function-based output, each function explicitly calls another function at the end. In most applications this will be no problem, but it is worth noting that each call will add a new entry on the call stack. If stack space or memory is limited, this can be a problem. In particular, if there are objects connected in a cycle, going round the cycle will continually add new entries on the call stack, eventually running out of memory.

The solution to this problem is to implement a kind of tail recursion. Tail recursion may be familiar from functional languages, where many algorithms make use of recursive function calls. Each invocation of the function at a new level of recursion would normally add a new entry to the call stack. Only at the end of the algorithm would all the calls unwind, with each simply returning its value to its caller. To prevent stack overflow, functional programmers noticed that there is actually no need to maintain all the function invocations on the stack. The last step of the function is often just to return the value of calling itself recursively. In that case, it is enough to maintain one invocation of the function on the top of the stack, and each new call simply overwrites that copy. At the end of the algorithm, the deepest invocation thus returns its value directly to the caller of the first level of invocation.

While our function calls are not recursive, we can note that the last step in each function is to call the next function. Instead of making that call from within the function, we can have the function simply return the next function to be called. An outermost loop can then call the function, receive as a return value the next function to be called, call that, and so on. The call stack only ever has one element in it, and we can happily have long chains or even cycles with no worries of running out of space.

This is the approach taken in the Python example in Chapter 8. The `main()` function there calls the current function, receives the next function back as the return value, and calls that. Python is a sufficiently powerful language that functions are treated as first class objects, which can happily be passed around and called.

Other languages vary in their support for dynamic method invocation. Smalltalk, Lisp, and Ruby predictably have no problems, but even the lowly BASIC-like OPL used for end-user programming in Symbian is able to call functions whose name is held in a string variable. While not quite as powerful as having functions as first class objects, this is sufficient for our needs. C is happy to offer function pointers, but some more recent languages have more trouble. The Standard Edition of Java can cope via its invoke(), but this requires a fair bit more code than OPL or Smalltalk's perform:. Java on mobile phones, J2ME MIDP, unfortunately lacks the reflection libraries that contain invoke(). C# has Type.InvokeMember, which is rather wordy with a receiver and five arguments, but basically similar to Java's invoke(). While C++ has poor reflection capabilities, it is possible to drop down to C function pointers or then use member function pointers.

Overall, then, almost all languages support functions passing back the next function to call, rather than calling it directly. As the code to make the call will only appear once in the outermost loop of the program, even the frightening number of arguments of C#'s InvokeMember, or the equally unpleasant number of exceptions thrown by Java's invoke(), should not put us off.

11.3.6 State Machine

State machines are among the earliest theories of computation, dating back to Turing (1937), and even the most prevalent “modern” forms, Harel’s statecharts (1987) and SDL, are over 20 years old. They share many similarities with flow machines, and much of the discussion above is thus relevant to generation of state machines. There are however some important differences.

In normal flowcharts, each object type can have its own kinds of exiting roles. For instance, IF is exited via a True or False role, or a switch case is exited via any of the various case values or the default case. Many objects exit on a simple “done my stuff, time to move on” basis. These decisions are thus dependent on the object type and tend to be based on the internal state of the program.

In contrast, state machines have a global concept of what can trigger a transition from one state to another. Typically this is an external event, although it can also be an event caused internally. States thus operate on a more laid back “done my stuff, now I’ll just hang around here” basis, waiting for something to kick them out of their reverie. While states thus lead a more relaxed life than flow machines, not everything is easier. Whereas in a flow machine each object could decide itself when it was done, and where to go next, in a hierarchical state machine an event in a top-level model can cause us to jump up out of a state in a lower-level model, even though that state specified nothing about the event. When the man at the top says jump, you jump.

Actions in a state machine are most often specified along transitions, although they can also be specified in a state, for execution on entering or leaving that state. In a DSM language, some features of a state can be specified as properties of the state, as opposed to actions. Recall in the Watch example, each state specified whether a certain time unit was flashing while we were in that state, and which display function

to use while we were in that state. Other actions—icons turning on or off, time calculations, setting alarms—were specified along transitions.

State machines are considerably more complicated than the flow machines above, especially if we look at full Harel statecharts. Compared to those, the watch example had no conditions on transitions, no state enter and exit actions, and no state history, and event transitions in a higher model did not affect lower models. This is a common state of affairs in a DSM language: we implement only a fairly minimal subset of all the features a full theoretical modeling language might have. This helps considerably when building and using the modeling language, but also when building the generator and domain framework.

Switch–Case In embedded applications, the most commonly seen code for state machines consists of nested switch–case statements. The first switch is based on what state the application is currently in, and the second is based on what event has occurred. As we saw in Chapter 9, generating this kind of code from a state machine is simple. For ease of understanding, we can omit some details here and assume that `EntryAction` and `ExitAction` are properties of `State` objects, while `Condition`, `Event`, and `Action` are properties of `Transition` relationships—all expressed directly as legal C code. This will give us the basic generic generator in Listing 11.10, shown here in a hypothetical template-based crawler language.

Listing 11.10 Simple C generator for state machines.

```
switch (state)
{
<% do .State %>
    case <%=id%>:
        switch (event)
        {
<% do ~From>Transition %>
            case <%=:Event%>:
                if (<%=:Condition%>)
                {
                    <%=~From.State:ExitAction%>;
                    <%=:Action%>;
                    <%=~To.State:EntryAction%>;
                    state = <%=~To.State%>;
                }
                break;
        <% enddo %>
        default:
            break;
    }
<% enddo %>
default:
    break;
}
```

We start the switch statement, then iterate over each state. For each state we generate a case statement and an inner switch statement. Inside that we iterate

over each transition from that state, generating a case statement for each event. Within this innermost case we generate an if statement for the condition, and inside it the exit action for the previous state, the action along the transition, the entry action for the new state, and an assignment that sets the new state as the current state.

Transition Table A second way of implementing state machines is to represent them as data rather than code. This approach is seen more often in object-oriented languages, whereas the switch-case approach is most common in procedural languages like C. More complex state machine modeling languages also tend to be better represented as transition tables.

As we saw in the Watch example in Chapter 9, each model can map to a new subclass of a state machine class. The state machine class provides a variable to hold a table that maps each state to the transitions for that state, and the subclass constructor fills in that variable for this model. The Watch example also had other tables that mapped states to their decompositions, display functions, and blinking information, but these would often be better represented in similar variables in individual State objects. The information recorded for each state and transition is thus domain-specific, in contrast with the not infrequent writings proposing “the ultimate state machine implementation in Java/C/Snobol.”

While the details vary, the main task of the state machine superclass is to receive events and push them through the state machine. It has a variable for the current state, or possibly a stack if state history is to be implemented. Looking up the current state in the transition table returns a table that maps events to transitions. If the event received is in that table, the transition it points to is considered for evaluation. Any condition on the transition is evaluated, and if that passes, the current state’s exit action, the transition action, and the new state’s entry action are performed. Finally, the current state variable or stack is updated to reflect the new current state.

In the Watch example, the states and button events were represented simply as Strings and the transition tables were Hashtables. It is also possible to make State objects, whose contents will then closely reflect the properties of the State objects in the model. In a procedural language implementation we could move in the opposite direction, and use integers to represent states, with arrays replacing the Java Hashtables. While an attempt to write a generic state machine in C would have problems with such an approach because the size of the arrays could not be known, in our case the number of states in each model is known at generation time. By judicious use of enum variables, the implementation could thus be simultaneously easy to read, fast to execute, and small in code size.

The transition guard condition and the various actions are of course rather harder to map into data. Important here is to remember that in the models, these should not be represented as code fragments, but as domain concepts and actions. One useful approach is that for every kind of action, there is a corresponding modeling language concept—an object, role or relationship, depending on how the state machine is

modeled. Still, these actions may often be generated as direct inline code for readability, so the problem remains.

How then do we include pieces of code in a data structure? Some languages such as Smalltalk offer simple support for code as data, and in other languages it can often be accomplished with a little work, for example, in Java via inner classes. If that is not possible, the discussion on dynamic method invocation at the end of 11.3.5 above is worth revisiting: we can implement each piece of code as its own function, and just supply a reference to the function in the data structure. Where even that is not possible, we can use the simple approach seen in the Java Watch implementation: make one big function that contains all the pieces of code in the model as cases in a switch or IF–ELSEIF statement.

11.3.7 Integrating Handwritten Code

Even a passing acquaintance with code generators will have taught you the cardinal rule: never edit generated code. If you need different code than was generated, make the necessary changes to the model. If you cannot accomplish what you want just by editing the model, look at improving the generator or modeling language to handle this and similar cases. If the code needed for this particular case really does seem to be a one-off, the modeler will need to write the code by hand. Even then, the changes cannot be made *ad hoc* to the generated file, or they will simply be lost when regenerating after subsequent changes to the model.

It is important here to differentiate between handwritten code that will be the same for all (or many) applications in the domain, and code that must be written by hand for a single part of a single application. The former forms the domain framework, which is written by the DSM solution team, and will be covered in the next chapter. The latter is written by the modeler of the application in question, and it is on such handwritten application code that we focus here.

Handwritten application code can be integrated in three ways:

- Protected regions
- Handwritten code in models
- Files referenced by models

Protected Regions A protected region is a section of the generated file that is known by the generator to be possibly edited by hand. When code is regenerated, the generator reads that section of the file on disk. If it has been edited, the contents of the region on disk are preserved, overriding that section of the newly generated version of the file.

To make this possible, the regions are generally delimited by specially formatted comments. The generator recognizes such comments and can parse information from them to link that region with the corresponding region in the new output. The comments also contain a checksum of the original generated code, so the generator can check if the region has been edited. Other solutions are of course possible:

for example, for the DSM tool to remember the line numbers and checksums of the regions without adding comments to the files. This is however prone to problems: for example, if files are replaced so the file on disk is no longer the previous generated file whose information has been remembered by the tool. Region comments also make the task of editing the files easier, by clearly showing which regions are protected.

There are a number of other details still to be covered, but we shall skip them here because protected regions cause more problems than they solve. They force DSM users to work on the models, the generated code, and the hand-edited code, and mix the latter two up in a troublesome way. Both models and code files need to be versioned, so the same information is recorded twice. Although protected regions may feel familiar from round-trip UML tools, there are better ways of achieving the same results.

Handwritten Code in Models Rather than adding handwritten code to a generated file, the code can be added to the model. Obviously, this should be avoided as far as possible, but when handwritten application code is a necessity, putting it in the models allows you to keep a single source containing all necessary information. The details vary according to the need and the modeling language, but one solution is to add a “Hack” object type to the modeling language. The object contains a free form text field, into which the modeler can enter one or a few lines of code. The generator for these objects simply outputs their code at the appropriate places of the output file, as determined by the positions and relationships of the Hack objects in the model.

Having a single source can be good, in particular if the Hack objects are short enough to be read as part of the model without cluttering up the display. If they become too large or numerous, they will tend to reduce the efficiency of using the DSM language. Also, code longer than a line or two would benefit from being entered in an IDE, rather than a simple text box. It is thus time to look at the final way of integrating code and models.

Files Referenced by Models The obvious next step from code in models is to move the code to its own files and have the models refer to the files. This promotes the “Hack” object to an “External Function” object, with a simple string representing the file name. Each piece of code can be in its own file, and that file can be edited in an IDE—at best along with the current generated code, easing any necessary integration with that code. Of course, it is best to keep such coupling to a minimum: the generated code is allowed to expect certain things of the handwritten code, but generally not the reverse.

In the simplest cases, the generator can output include or import statements in the generated code to refer to the handwritten code. The generator should also produce a list of the referred files, and check that they exist. If there are more than a few such files per model, it may be easier to move the pieces of code from each being a file to each being a function, with all functions from one model contained in a single file.

At this point, we have moved into the territory of programming in general. Tactics to allow further expansion of this idea include automatically generating placeholder functions, which can be overridden by handwritten code in a concrete subclass. Moving further in this direction, however, takes us out of the domain of DSM and into the kinds of solutions we see in older modeling tools. As those tools have failed to raise developer productivity, if we find ourselves adopting their tactics we would do well to step back and reconsider our DSM solution. Handwritten application code is not quite the epitome of evil, but neither is it by any means a necessary evil.

11.4 GENERATOR STRUCTURE

In many ways, writing a generator is just a special case of writing a program: there are as many different ways to structure the same behavior as there are programmers. Some ways have however been found to be better than others: easier to create, debug, maintain, and understand. Many—but not all—of these ways of structuring code are also appropriate when writing generators, and the good developer will naturally use them. Generators also have some special requirements of their own, mainly because of the strong existing structure provided by the modeling language.

In this section we will look at various ways to structure generators. Each individual way can be taken and used on its own, or in any combination with the others. We have however found that building a hierarchy of generators top-down in the order specified here works particularly well.

A top-level “autobuild” generator is thus visible to the user, and is responsible for building the resulting application for testing. It calls generators defined on the various modeling languages, and these split their task up according to the various files to be produced from each model. Where necessary and possible, the file generators again split the task up according to the object types in the model. If more than one programming language is needed for the same set of models, this structure can be built in parallel for each.

11.4.1 Autobuild

The Agile movement has grasped the importance of continuous integration via automated builds, even though their second tenet is to favor “individuals and interaction over processes and tools.” The truth is, every application must be compiled and linked to be any use (substitute compile and link with appropriate terms for your language!). If you do not automate this process, developers will be forced to carry it out manually every time. In addition to being error prone and hard to reproduce, this is also boring, leading to developers not doing it when they should.

Our experience in DSM is that a working autobuild provides far more value than could be imagined. In particular, automatically running the generated application provides a new level of abstraction that has to be experienced to be believed. There is a real power in going directly from modeling to seeing the full application running—not a prototype or simulation, but the actual application. As the models are on a high level

of abstraction, often close to the concepts of the end user of the application, you really have the feeling of staying focused on what you actually want in the application, rather than the details of how to implement it.

As the saying has it, though, you can't make an omelette without breaking some eggs, so for others to be able to ignore the details of how to implement their programs, we need to look here at the details of how to implement such support. But hang on a minute, isn't there some clever meta trick we could apply here, so we can ignore the details of *this* task? Funny enough, there is, and you'll be forgiven your lack of surprise when we tell you it's using a domain-specific language. In the domain of building software out of an assorted collection of files, one simple word says it all: make.

The make tool allows you to specify what files your finished application is dependent on and what command is necessary to process those files into that application. There may be several stages and intermediate files, with different commands necessary at each stage. For instance, an executable is built by a linker from object files, but the object files themselves must be built with a compiler from source files. With a little work, it is possible to make a makefile (the input to the make tool) that simply lists the source files, the general command to compile source files, and the general command to link object files.

When writing software by hand, keeping such a make file up-to-date is just a matter of remembering to add a file name each time you create a new source file. When building software from models, we definitely do not want the modeler to have to know what source files we are creating, and remember to update a makefile. Since the generator is creating the source files—apart from some framework files that are fixed regardless of the models—it is easy enough to generate a list of the files for the make program. If your generator language supports it, the easiest way is to append the name of each source file to a variable or temporary file as you start generating it. After generating the source files, you can generate the make file, including the contents of that variable or temporary file at the appropriate place.

The alternative to using make, or its equivalents such as nmake and successors such as ant, is to generate and execute the command lines for the build commands. You can either execute the build commands directly from the generator or generate them into a batch file and execute that. In general, the latter approach is easier to work with, since the batch file also forms a record of exactly what commands were executed.

You also want to make sure you get a record of any errors that occurred, especially while you are still working on the generators. One good way is to redirect error output to a file, and open that file in a text editor if there were errors. Below is an example for Java on Windows NT/2000/XP. The first line runs the Java compiler, redirecting error output to an errors.txt file. The second line checks if the compiler returned an error code, and if so opens the errors.txt file and exits. The /b exits the batch file, as opposed to the whole command shell, and the 1 makes the batch file return an error code, so a calling program knows there was a problem.

```
javac *.java 2>errors.txt  
if errorlevel 1 start errors.txt & exit /b 1
```

The main difference between using make and directly calling build commands is in how platform dependent the build process is. Make largely insulates you from changes in build platform, and ant does so almost completely; a batch file or shell script will however only run on a Microsoft or Unix OS, respectively, and will quite probably be dependent on a particular version of that OS. If you find yourself having to write command scripts on several platforms a good resource is www.ss64.com, which documents the commands for Windows, Linux, and Mac OS X.

One main benefit of make is often lost when moving from writing source files by hand to generating them. Make is able to see which files have changed since the application was last built, and only update those intermediate files that are dependent on the changed files. Generators normally regenerate all files, since they cannot tell which models have changed (in particular where the same object is reused in multiple models). This means all files look new to make, and all will be compiled, sometimes taking much longer than would strictly be necessary. Avoiding this problem is simple, and the better DSM tools already support it automatically: if a generated file is identical to the file of the same name already on disk, leave that file untouched. If your DSM tool does not support this, it is possible to create the same effect by generating source file contents with a temporary extension, comparing them to the previous source file, and only copying them over the previous version if there is a difference.

While the bulk of work in an autobuild generator will be in creating and executing the makefile or build script, we must not forget the surrounding work. First, the autobuild generator must run the other generators to actually produce the code, before executing the build: the following sections will cover this. Finally, after a successful build the autobuild should open the resulting program. If the program is a desktop application intended to run on the same platform it is developed on, it can simply be started. If it is targeted for a different platform, it can often be opened in an emulator or simulator. As these are often slow to start up, it may be worth investigating if the emulator itself can stay open, with each autobuild simply updating the application in the emulator. If the generated program does not have a user interface, and hence there is no point opening it, the final step of autobuild could also usefully be to run automated tests against the program and display the results.

11.4.2 Generator per Modeling Language

A full DSM solution for a given domain will normally have more than one modeling language. The kinds of information needed to describe an application will be divided over a few modeling languages. These modeling languages, and the models made with them, should exhibit high cohesion and low coupling. If this is the case, the information needed by the generator at a given stage of generation will generally be found from the same model the generator is currently in. A generator will not need to hop back and forth much between different models in different modeling languages: if that were the case, the modeler would most likely have to perform similar mental acrobatics in creating and reading the models.

The partition of the domain into modeling languages tends to offer a good basis for partitioning the generator. Following the same partitioning closely also brings the benefits of simplicity to the metamodeler. Associating each generator with one modeling language gives an encapsulation of data and behavior similar to that found in object-oriented programming. The modeling language defines the data structure, and the generator is the code that operates on the data.

Autobuild thus often calls a top-level generator per modeling language. The top model may generate significant code itself, in which case it will have a top-level generator in addition to the autobuild generator. Alternatively, the top model may be more like a set of links to the real models, with little other content of its own. In that case autobuild will simply iterate over the links, calling the appropriate top-level generator for each target model.

Each model may contain further links to other models. If these submodels are in the same modeling language, the generation for them will most likely be a recursive call to the top-level generator, but now on that submodel. If the submodels are in a different modeling language, they are somewhat more likely to be including information by reference rather than containment. The generator may then simply pick up the information it needs, or generate a reference, without needing to process the full content of those submodels. The full content will be generated via some other path from the top-level models.

In some cases, the information partition that is best from the point of view of the problem domain may be at odds with that required by the implementation domain. The implementation language or frameworks may require a certain set of information in a file, but that information might be spread over several models in various modeling languages. In that case, it might make more sense to have autobuild call a generator per file or per file type, and have that generator call partial generators for each modeling language it needs to visit for information. Generally, though, the pattern is that a generator per modeling language will contain a number of subgenerators, one for each file type to be generated.

11.4.3 Generator per File Type

Each model may generate more than one file: as mentioned above, there may indeed be a many-to-many relationship between models and files. This is problematic for generators based on templates or the model visitor pattern. While generation into multiple files may be possible, you tend to find yourself working against the tool rather than with it. API-based generators can of course cope, and stream-based crawler generators come into their own.

There can be several different reasons why more than one file may be generated from a single model. In some languages, the implementation may require multiple files, for example, a .c and .h file for C programs. In other languages, standard practice may encourage them, for example, a Form1.cs C# program may have a separate partial class for its UI definition, Form1.Designer.cs. These cases are predictable, with one model always giving a pair of files. More complicated cases are also common, for

instance because the modeling language represents information in a significantly condensed form.

One DSM model may simply contain the same information content as several source code files. A fair part of this reduction is removal of redundancy, which can be split into two parts. First, each source code file contains significant similarities to other files of the same type. Constructors, accessors, database links, and serialization are often made up of the same few lines of code, with the only variation being the name of the piece of data in question. Second, many of today's systems require multiple types of code files for one conceptual entity: a C# interface, data service class, database table definition, database stored procedures, and so on. When these are written by hand, the names and other details of the attributes of the entity must be duplicated across all these files, but in the model each is represented only once.

In addition to code files, or code-like files such as database table definitions, there are also often various other files required as output from a model. For example, the Symbian S60 platform can require as many as 27 files for a simple "Hello world" application: menu and UI definition files, localization files, application descriptor files, icon lists, and so on. Some of these can be generated directly from the information in the top-level model of the application; others such as the localization files must recurse through all the models with UI information. This is easy for a crawler or API-based generator: they can simply open a new stream for each file and walk the structure of the models to visit all the required information. Template-based or strict model visitor generators may well require some extra work or hacks to produce the required output, as noted above.

11.4.4 Generator per Object Type

When generating from a single model, the form of the output required normally varies according to the type of the object in question. This is the basis of the model visitor pattern, which says that each object type should have its own generator. Where the strict model visitor implementations fall down is when information from the same object is required in several different places in different forms. This is no problem for other generators, but they would still do well to make a clear connection between the object type and the generators for it.

In most non-API generators, object types themselves do not have their own generators. Instead, the generators are stored globally or with each modeling language. In this case, the subgenerators for each object type can be named after the object type. Where the object type requires different output for different parts of the generation, the names can be formed by prefixing the kind of output required to the name of the object type, for example, Action_Alarm and Action_Icon for the Alarm and Icon types' actions in the Watch example.

This order is generally better than the reverse order—which would be more familiar from object-oriented programming—because there tend to be more similarities between all of the Action generators than all of the Alarm generators. Unlike object-oriented classes, the object types have no notion of hiding the

implementation of how they store their information. Access to the properties of the object is direct, and access to relationships and subgraphs of the objects is generic, working the same regardless of the type of the object.

Some DSM tools equate models and their elements closely with object-oriented classes: any information access must be carried out following the rules of statically typed languages. Aside from the extra work this entails when writing generators, such tools also often choose to make access to relationships happen through objects: an object stores its own relationships. The problem of this becomes apparent when you want to reuse the same object in a different model: you may well want it to have different relationships there, but the generator cannot tell which relationships belong to which model.

A more useful approach is to recognize that information about which relationships connect which objects belongs to the model. Since while generating we are always within a particular model at any given time, we can always follow the relationships of an object in that model. This also allows the generator language and generators to be simpler: there is one global mechanism for following relationships, specifying the type of the relationship as an argument, rather than having a separate message for each kind of relationship for each object.

It is also worthwhile noting that in some cases, relationships and even roles may have their own subgenerators. An example of this was in the Watch Application models in Chapter 9: the typing of actions was via the relationship type, and the details of the action were contained in the relationship or its roles. For instance, a Roll relationship connected to a Time Variable, and a Boolean property of the relationship specified whether the action incremented or decremented the value of the Variable. Similarly, a Set relationship could connect to the same Time Variable and, say, an object representing the system time: this would set the value of the Variable to be the current system time. Since the types of the objects involved in both cases were the same, and the type of the relationship was what determined the semantics of the action, it made sense in this case to have a generator per relationship type.

11.4.5 Parallel Generators per Programming Language

The preceding types of generator cover the output of a full working application, containing all the information from the models, marshaled into the required files and formats for the given platform, domain framework, and compiler. In some cases, even this is not enough: we may want to be able to generate the same application in more than one way. The most common need for this is where we want to generate the same application for more than one platform. This however is normally fairly easy to accomplish by using a different domain framework for each platform, but maintaining the same interface between the generated code and the framework. In many cases, the generated code can remain the same and only the framework changes; in others, one set of generators may still suffice, including a few conditional sections that depend on the platform for which we are generating.

A more difficult task presents itself if for some reason we must generate the same application, but in a different programming language. The information content of the generated output will be the same, but the distribution over files, ordering of the

information in the files, and syntax of those files will all change. In some cases, fortunately rare, the best architecture for the application will differ so greatly between the two languages that the code itself will be largely unrecognizable.

None of these of course presents any problems to the modelers: they simply build one model and choose which generator to use. The hard work of building two parallel generators is left to the metamodeler: once again, the one suffers for the sake of the many. The root cause of needing to output two different languages is of course worth investigating. Unfortunately, even where there seems little rational call for it, greater forces may be at work in the form of senior management and stakeholders. If so, console yourself with the thought that you can at least blow their socks off by using DSM to achieve the same net result of the holy grail of automatic translation between languages!

For model visitors and templates there will once again be the problem of how to allow multiple parallel generators for a given object type or file. Crawlers and API-based generators will simply duplicate the relevant generators. If the architectures and languages are similar, some higher level generators may be reusable between both languages. If they simply follow model structures and call subgenerators, as opposed to actually outputting code themselves, they can simply be parameterized to know which language version of the subgenerators to call.

The more different the architectures are, the less benefit there is to be gained from attempting to keep the generators similar: at some point it becomes easier to simply write a new set of generators from scratch for the second language. Whatever solution is chosen, there should be some process in place to ensure that changes made later for one language's generators are propagated to the other language's generators when necessary. The chance of there being any way to automate these changes is as slim as that of automatic translation becoming 100% reliable, but this doubtless will not stop some from trying. Before you embark on that route and rediscover the only workable solution—having some intermediate form from which you can generate both languages—stop for a second and remember that you already have precisely such an intermediate form: the models.

Automatic Translation and Intermediate Formats There have been attempts in some DSM tools to force the use of such an intermediate form between the models and the code. The background to such attempts is however not from the world of DSM, but rather from UML. In UML, the domain is object-oriented languages, and there is no expectation that working applications will be generated from models. Instead, it is sufficient for the UML tool to generate the class and function headers. As there is no problem domain semantics in the models, and hence no knowledge of what the code should actually do, the generation simply maps a graphical description of object-oriented classes into a textual format. At that level, object-oriented languages are similar enough that simple mappings can be made, and where the mapping is less than ideal, it does not matter anyway, since nobody is going to execute the code.

Coming from such a background, where all modeling languages are UML and all programming languages are object-oriented, it is tempting to envisage DSM

code generation as being first a transformation into UML, then a transformation per language from UML into code. There is generally an implicit assumption at this stage that there need only be one transformation written from UML to Java, say, and all DSM languages could use that same transformation. Had that been true, then UML would have provided what its hype implied: full applications from models.

Unfortunately, the model to code transformations for two different languages, even closely-related object-oriented languages, are rarely so similar that they could be considered the same transformation with just a few differences in syntax. Give the same problem description to a Java programmer and a C++ programmer, and the resulting programs will be distinctly different. Good programmers will take advantage of the features of the language—and have to work around its foibles and shortcomings. Automatic translation between object-oriented languages is a nice pipe dream, but not a practical approach; the same is true for universal intermediate formats.

11.5 PROCESS

Although Chapter 13 will discuss how generator creation fits into the overall DSM definition process, it seems useful here to examine a few details and pass on some tips. You can view Chapter 13 as the strategy or game plan for the whole DSM solution team, while this section looks at tactics and instructions for the generator builder.

11.5.1 Creating and Testing Generators

As we mentioned at the start of this chapter, an important part of the process of creating a generator is knowing when to start. Unless you know the input and output required for a program, you are unlikely to be successful in building it. It is thus important to first wait until the modeling language is in an acceptable state. All the major concepts should be there and you should have made a few example models.

These example models will show that the modeling language works from the point of view of the modeler, which is the most important factor. You should also make sure when making the example models that the modeling language allows you to capture all the information you would need to build the application. In fact, it is sufficient to have enough information to be able to build *an* application corresponding to the example model: there are always choices involved when hand coding.

Next you need a matching pair of an example model and the corresponding working code. We looked at this in more detail in Section 11.1: either model an existing application or write the code for an example model. If you have the existing framework code, you will of course use that; otherwise, do not worry too much if you find repeating sections of code. Breaking those out into a domain framework is the subject of the next chapter; for now it is fine to do a little light refactoring as you go, but also acceptable to have larger pieces of code as boilerplate in the generators.

When you have the working code, use that as a basis for the generator. Actually, the working code itself is your first generator: simply make the generator consist of one

large literal string! While that generator will produce exactly the right output for this model, we want to progressively break down that generator into smaller elements, with conditionals, loops, and reading values from the model. Follow the four steps of Section 11.1 to gradually reduce the generator down to its minimal form. You can run the generator and diff the output with the original handwritten code to check your progress.

Remember to keep the generator as simple as possible: push decisions and complexity down into the generated code (and later into the domain framework). If the generator needs more information, identify the extra problem domain knowledge that needs to be captured by the modeling language and find a minimal natural representation for it.

Once you have a working generator for that example model, take a snapshot of the model, generator, and code. Then try changing things in the model, and make sure the code still works but reflects the change. You can test both by running the code and by comparing the code with the snapshot. Start with simple changes like names that will be visible in the running application, and progress to more complex ones like adding an extra object. When you are satisfied that your generator works on the sample model, even when that is lightly modified, you are ready for the next step.

Building one application was never the plan, so go on and build a second, but keep the first application intact. Make the model for the second application similar to the first, varying by about the same amount as your boldest modifications did. Make sure that you can generate both applications and that both work. This may also be a good time to think about the modeling language and how you are going to enable reuse between models—a subject we cover in Chapter 15.

Working like this, you will soon have a fully functional generator. As you hit problems, use your ingenuity to find a way to solve them, remembering that there are several areas you can change: the models, the modeling language, the generator, the generated code, or the domain framework. One of these will almost always offer a good solution—if you are spoiled for choice, the later areas such as the domain framework are often the better places.

Later, when developers are using your modeling language and you want to enhance it or the generators, the same tactics still apply. You first need to know what information you want to capture and what is a sensible format for the modeler to input that and use it in models. Only after that do you look at what the code might be for an example usage, and work back from that code to the necessary generator change. Changes to the generator—whether refactorings or additions—can always be checked by comparing the output code from a known good model to its previous known good output.

You will probably find it useful to keep a set of example models and output for this purpose. If possible, do not let your work be based solely on these, but use real models built by the modelers. One good approach in the long run is to always use your example models to test changes, but each time use one other real model. Use a different real model each time: although you will spend extra time because that model will be unfamiliar, it forms a valuable part of the feedback loop between the language developer and users.

11.5.2 Sharing and Maintaining Generators

Generators are coupled with the modeling language on one side, and the domain framework, code architecture, and programming language on the other. A good principle of modularization is to keep coupled elements close together, particularly where a change in one often implies a corresponding change in the other. At the stage when you are first writing the generator, the first coupling is stronger: the modeling language is still likely to change, but for a given generator the programming language is unlikely to change.

The generator and the modeling language thus form a clear coupled pair, and it is good to keep them together. This guarantees that when a user receives an updated version of the modeling language, the latest version of the generator accompanies it—and vice versa, although that direction is less common.

While sometimes one may be updated without the other changing, there is no sense in allowing the generator and modeling language to lead separate lives. The vast majority of random combinations of generator versions with modeling language versions will simply fail to work. Separating them would be like separating a lex scanner from its yacc parser. While there is some degree of freedom, they are far from orthogonal.

The coupling of the generator with the domain framework is also notable, and once again it is mostly the generator that must change in response to changes in the framework. In this case, however, there may be several different parallel versions of the domain framework. The same generator produces code that runs with all of these parallel framework versions, each of which provides the same services. The different frameworks are used to abstract different underlying platforms, libraries, or components, hiding the differences behind a consistent interface.

An overview of the directions of information and change propagation is shown in Fig. 11.2. One modeling language, here the Watch DSM language, may have several generators (e.g., for different languages): the black arrows indicate the flow of information from the models in that language to those generators. Changes to the modeling language must be propagated to the generators, shown here by the thicker lines: information and change propagation flow in the same direction.

The situation on the framework side is more complex. One generator may produce output that works with several frameworks, but changes are propagated from the frameworks to the generator. In fact, the framework changes that must be propagated are those that change the interface between the framework and the generated code.

Keeping generators together with the modeling language—in the same tools and same files—ensures that when a generator is applied to a model, their versions are in sync and they share a consistent view of things. Building generators in the same tool as the modeling language also allows the generator editor to offer context-sensitive help and syntax checking on the many references to modeling language concepts. A tool could even update generators automatically if the name of a modeling language concept is changed.

A well-architected tool can thus ensure version synchronization between modeling language and generators. Any serious tool should ensure version synchronization

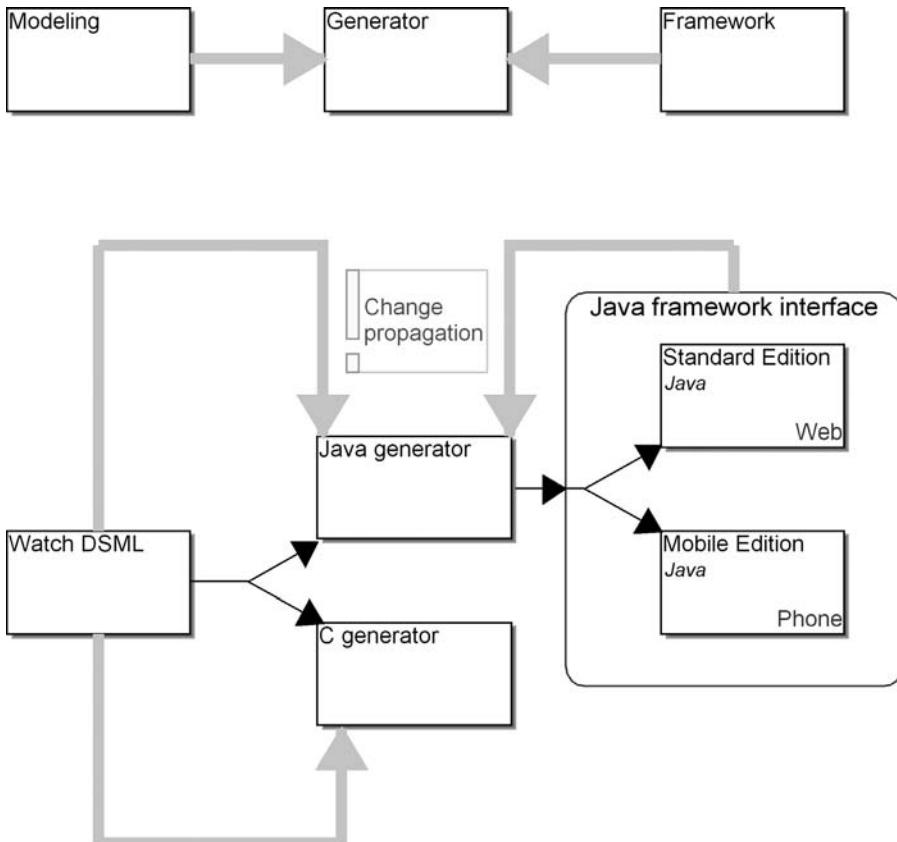


FIGURE 11.2 Coupling between modeling language, generators, and frameworks

between models and modeling language. This leaves version synchronization between code files, which is somewhat harder as these are normally pushed out away from the safety of the tool. A good idea is to make the generator output into the file the version information from all three sources: generator version, modeling language version, and model version.

Wherever possible, all the generated files needed to build an application should be produced at once, guaranteeing their version consistency. This is one of the additional benefits of a fully automated autobuild. Where that is not possible, one solution is simply to rely on the developers to do the right thing when linking together output from several generation runs. This, after all, is what has been the practice when hand coding—or indeed the situation was far worse, as different developers wrote different kinds of code for the same task based on a whole variety of personal and historical reasons.

The automated inclusion of version information by the generators does however also make it possible to reliably compare versions and ensure whatever level of correctness seems desirable.

11.5.3 Version Control of Generated Code

Do you version your .o files? No? Fine, let's move on!

As mentioned above, with automatic generation and build you should not need to version the generated files. They are an intermediate product and can always be reconstructed from the primary source, the models. If, however, your DSM tool does not ensure correctness or combined versioning of models with their modeling language and generators, there could still be some scope for doubt. In other words, a poor DSM tool may not produce the same output twice from the same model: the modeling language or generators may have changed, even though you took the same version of the model from version control. In that situation it is important when versioning the models to include version information of the modeling language and generators.

In some rare cases, it may also be useful to version the generated code. For instance, if generation from the DSM tool is time consuming, and there are centralized nightly builds from hundreds of developers' models, caching the generated code can speed things up. Another case could be where one set of developers builds the models and generates the output, and a second set takes that output and works with it—not changing it, but perhaps writing other code files by hand to work with it. It would generally be better to let this second group have access to the models, but this is not a perfect world.

The need for version control of generated code is thus the exception rather than the rule. In real world cases, our experience has been that the improved quality and consistency of generated code largely removes the need to worry about versioning—throughout the DSM solution. To a generation who have grown up believing in the safety net of version control, this may seem hard to believe at first. The fact is that current version control systems, practices, and requirements have grown out of the needs of hand-coded programs, and DSM removes many of those problems. Some parts of a DSM solution will still need to be versioned, but generally not the generated code.

11.6 SUMMARY

The generator forms the keystone of a DSM solution: it must interface with the modeling language, models, domain framework, and language to be generated. In some ways, building the generator is the most challenging task in DSM, since it requires a strong understanding of all of these areas and also of the generator facilities offered by the DSM tool. To ameliorate these potential difficulties, it is important to remember two things:

- Keep the generator simple: push complexity down into the domain framework.
- Do not try to build the generator too soon: wait until you have the modeling language, an example model, and the correct output to be generated from that model.

Making a proper code generator is definitely worth the effort. A sizable proportion of the productivity benefits of DSM are due to the generator, and in particular its ability to turn the declaratively specified solution from the models into the procedural form required by most of today's programming languages. While in theory, and sometimes in practice, it is possible to interpret the declarative form at runtime, performance constraints tend to rule this out. This is no surprise: when we moved from machine code to Assembler, or Assembler to 3GLs, the majority of languages chose to transform the newer form to the older once at compile time, rather than continually at runtime.

The different types of generator facilities have their strengths and weaknesses. The sweet spot at the moment seems to be crawlers, which offer more power than model visitors or templates, while keeping the generator on a higher level of abstraction than the model API approach. More advanced template languages are also a good choice, especially where they are integrated in the DSM tool. Solutions that involve a separate generator program will always incur the additional cost of outputting the model in a format the generator can read, and parsing that format to rebuild the model structures on which the generator will work.

Generation is primarily thought of as focusing on code, but may also include code-like or declarative formats such as SQL or XML, documentation files, and simple text files. It can also be used as the basis for model checking reports, although the modeling language may be a better place for rules that should always hold, and for certain specialized analysis tasks it may be better to use existing external programs.

When generating code, the kind of code you want is determined by what you would write by hand, with perhaps some concessions to ease of generation. While this would appear to leave little in the way of generic patterns for code generation, experience shows that almost all code can be seen as either a flow model or a state model. While simple flow models can be generated with a naïve recursive approach, anything more complicated tends to use functions, called either dynamically or statically, or then switch-case statements working similarly to functions.

Having the generator not only create the output files but also compile them and run them is a key ingredient in raising the abstraction level. Such an autobuild approach allows the modeler to go straight from models to the finished product, remaining at a high level of abstraction and thinking in the problem domain all the time.

Larger generators should be broken down as necessary into subgenerators for each modeling language, file type to be generated, and model element type. Repeated sections in generators should be refactored into subgenerators, just as with any program.

In many ways, the process of building generators is just a case of applying the generic parts of best programming practice. A generator takes a certain kind of input, normally an object structure, and processes it to produce a certain kind of output, normally one or many text streams. Only when the input and output formats are known, and an example of each is available, can generator construction sensibly begin.

The close coupling with the input format means the generators should be distributed and versioned with the modeling language. It is also good practice to have the generator annotate the output with version information about the models, modeling elements, and generator itself. While a good DSM tool will make most versioning issues invisible, having the information there may one day solve that mysterious “but it worked yesterday!” problem.

CHAPTER 12

DOMAIN FRAMEWORK

Domain Specific Modeling (DSM) is only one of a number of ways of improving developer productivity in a given domain. Like many of these ways, it takes advantage of some kind of repetition in applications built for that domain. A common approach is to build a library of reusable components, leaving the application code architecture to the developer. One step further is to build a framework for applications in that domain. As well as providing components, a framework fixes the architecture of the application, with the developer plugging in her own application-specific code at predefined points. Using a framework generally involves a large initial investment of time to learn how it is meant to be used—in the best case by reading extensive and accurate documentation, but too often by trial and error and user forums.

In an attempt to make it easier to get started with a framework, there is sometimes a wizard that will query the developer for necessary information and generate a partially completed skeleton application. This approach is particularly seen where the framework is integrated with an Integrated Development Environment (IDE), for example, early Graphical User Interface (GUI) builders. The problem in many cases with this approach is that it presents the developer with a mass of code that she has not seen, let alone written, and yet is expected to maintain. That differs from successful approaches to raising developer productivity, such as compilers, where the developer can remain on a higher level and not have to understand or work with the larger mass of code that is automatically generated on a lower level.

With DSM, the code that is generated remains firmly “under the hood.” The developer keeps working on the model level throughout development. There’s

no working on an unholy interlinear mix of autogenerated and handwritten code. What DSM does is raise the level of abstraction to a level where things are simple for the developer. Of course, underneath there is code, and underneath that is assembler, machine code, microcode, logic gates, electrons, and so on. The trick is how successfully we can hide the next level down and keep it hidden. DSM does it, most wizards do not.

If wizards represent something of a dead end in the evolution of reuse, how can we best use components as part of a DSM solution? In particular, given we quite probably already have a library of components for our domain, what extra do we need to take best advantage of them with DSM? In our experience, the best approach is to create a separate layer, the domain framework, between the generator and the components.

Fig. 12.1 compares manual coding, wizards, and DSM. The darker blocks represent the artifacts that the developer must work with. In the first approach, manual coding, all code is written and maintained by hand on top of an existing set of components and platform. In the second approach, the developer gives input to a wizard that generates some of the code. The developer has to add code and is also responsible for maintaining the code generated by the wizard. The input to the wizard is often lost, or the wizard cannot be repeated. Even in the best case, there are a bunch of gotchas, documented and undocumented, awaiting the developer who tries to repeat the wizard after significant manual coding. In the third case, the developer works solely with models in a DSM language. The code is generated from those models and takes advantage of a domain framework.

Unlike the component framework it rests upon, which is generic for all problem domains and/or all companies, the domain framework is specific to the problem domain and the particular company that is using the DSM solution. This allows the generated code to be smaller than the generated or handwritten code in previous approaches. Although the generated code and domain framework together will be

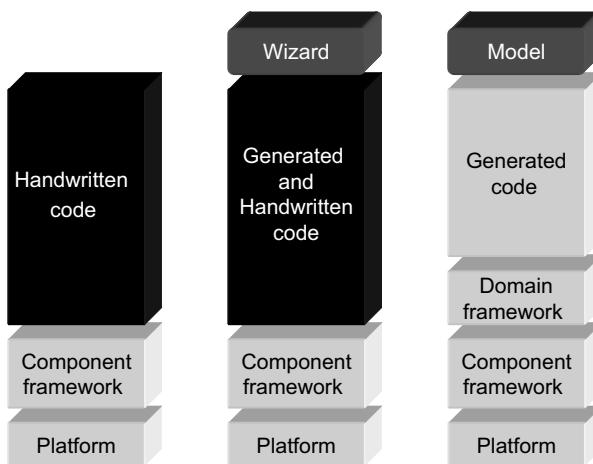


FIGURE 12.1 Manual coding versus Wizards versus DSM

around the same size as previous approaches, the amount of code across the range of products will be less since the domain framework is only counted once.

In addition to making the generated code smaller, benefiting the developer, the domain framework also helps the metamodeler. It insulates the generator from changes in the components, and at the same time allows us to make concessions to ease of generation, without changing the components themselves. In many cases, the components have arisen from spotting repeated patterns in the code, rather than being systematically architected for the problem domain. The domain framework thus also offers us a chance to provide a more domain-oriented, less implementation-oriented, interface to existing proven functionality.

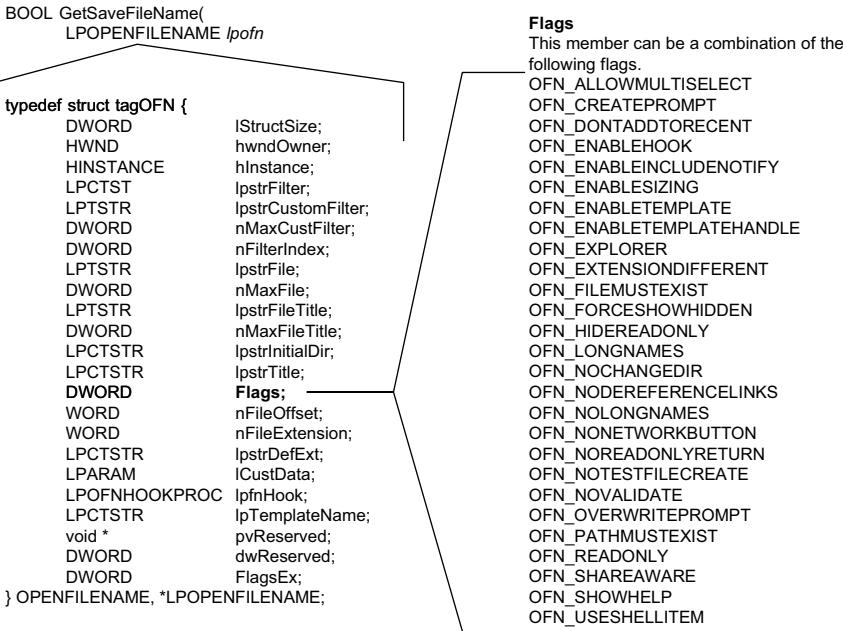
Before this starts sounding like too ambitious a project, let's reassure ourselves. The domain framework does not need to raise code to a rarefied level of abstraction, easily digestible by any third grader. We have two other layers above it, the generator and the modeling language, which bear the bulk of that burden: the domain framework will mostly be invisible to developers. Second, building a domain framework on top of existing components is not really a new kind of task, simply an extension of the already familiar task of condensing repeated code into reusable functions and components. Finally, as layers go, the domain framework is generally thin. Initially it can be completely empty, and you add to it when necessary, prompted by needs arising from building the generator.

12.1 REMOVING DUPLICATION FROM GENERATED CODE

In many ways, DSM is all about removing unnecessary duplication of effort, and in software development this generally translates into removing unnecessary duplication of manually written code. The generator plays a central role in this process, replacing many instances of copy–paste duplication with one copy of the relevant code in the generator. Even without DSM, component libraries have played an important part, allowing a repeated piece of code to be collected into a component and invoked each time via a significantly shorter piece of code. A domain framework makes it possible and cost effective to go further than normal in this area. Let us look first at a concrete example of how the economy of reuse changes in DSM.

The Windows call to open a “Save As” dialog effectively takes 23 parameters, one of which is any combination of a set of 27 flags (Fig. 12.2). This has two consequences: the number of different possible dialogs is astronomical and the code to produce any one of them is substantial. Any given application will however generally contain only one or a few such dialogs, so the code for the dialog is generally written by hand—or more likely copied and pasted from last year’s project. For many programmers, the number of calls to such a piece of code is simply not high enough to persuade them to create their own higher-level dialog function.

The code for a “Save As” dialog would thus be likely to be found as a fairly long boilerplate section in a DSM generator. Some features of the dialog may be picked up from the model and used to affect the generated code, either being output to become



From MSDN Library, Win32 and COM development,
© 2006 Microsoft Corporation.

FIGURE 12.2 Parameters to configure a Windows “Save As” dialog

parameters to the call, or being used as conditions in the generator to cause different fixed text parameters to be output.

Within any given product built with the DSM language and generator, this solution will look all right: the code will probably appear only once or twice. Across the whole product family, however, the code will appear many times. While this duplication also occurred in handwritten code, it was spread over so many developers and so much time that developers rarely felt a pressing need to refactor it.

In a way, the generator already provides a good refactoring: the boilerplate code appears only once in the generator definition, and all occurrences in the code are produced automatically from that. However, this still leaves many occurrences in the code. The time-honored principle of avoiding duplication is thus not yet fully applied. The principle is so well proven, and such an integral part of DSM, that a deep cost-benefit analysis is probably unnecessary: a few examples should suffice.

In an embedded product, the extra code size may be an issue, particularly where a whole product is built up of many smaller DSM models. In a distributed project, coherence of code generated with different versions of the generator may be a sticking point. In a multiplatform project—even just over several versions of Windows—the code to open the dialog would vary on each platform, so moving the variability from the generator to the framework would simplify adding new platforms. In all cases, Occam’s razor applies: source code explains to the computer what it should do, and the fewer words we can do that in, the better.

We would thus want to look at refactoring that repeated block of code out into its own function in the domain framework. The function would set up some desired defaults for the majority of parameters, and have its own parameters for a few features of the dialog that vary. The generated code will thus be much shorter in each case, easier to read, and easier to relate to the data in the model. At the same time this approach will also harmonize the use of file dialogs, removing unnecessary variation—necessary variation will of course be captured in the parameters and specified in the models.

12.2 HIDING PLATFORM DETAILS

Modern software development is based in a large part on existing general-purpose components, often supplied as part of a development platform such as Java JDK or Windows .NET. After a platform's early days, there is an implicit assumption that these components will be accurate, bug-free, and offer the current interface for the foreseeable future. There is also often the implicit assumption by the developers that the code they are working on will continue to use the same platform. Sadly, neither of these assumptions seems particularly well founded.

If the operating system, platform library, and SDK we rely on cannot, in the end, be relied on, how can we reduce our dependence on it? One way is to create a new framework: cross-platform, open source and hence bug-free (or at least if there is a bug you can quickly correct it), and switch to using that. Irony aside, this is indeed a popular approach, as can be seen from examples such as GTK+ or the lower-level Cairo Graphics library. However, to be successful, such projects need a large number of participants with different platforms and needs, and require a number of years to complete. While most of them never make it, the ones that do are a great addition to the tools available to software developers.

Another approach is to attack the other end of the problem: rather than trying to make a new implementation framework that is more generic, make a new framework layer that is more specific to your problem domain. The interface of this domain framework upwards to your generated code can remain the same, and only the interface down to the components needs change with the platform.

12.2.1 Bypassing Bugs and Platform Evolution

We saw an application of this approach in the Watch example in Chapter 9. At the start of the implementation, we were using JDK 1.0 for maximum compatibility. We quickly discovered a number of bugs and significant missing functionality in the Java Date class. Looking at JDK 1.1, some of these bugs were corrected, but new ones had been introduced. Despite the addition of extra Calendar, TimeZone and DateFormat classes, java.sql.Date and java.sql.Timestamp, there were still large areas of functionality missing. In particular, we needed the ability to add and subtract times as both points in time and periods of time—a fairly standard requirement. A more domain-specific requirement was to increment a single unit of a time with rollover, as

watches do when editing the time. For instance, 59 seconds would be followed by 00 seconds, without incrementing the minutes.

Normally we would have implemented our own calls over the top of the existing Java Date class. In this case, its poor quality, lack of stability, and focus shifting away from simple time toward esoteric calendars and time zones led us to create our own METime class. This class interfaced only with low-level platform primitives like the system millisecond clock and modular integer arithmetic. While as novices in both Java and the time domain, writing this class required more effort than it should have, it was still completed in around half the time we had spent fighting with the Java Date class.

The main value of METime is that it has remained unaffected by the significant changes to time and date handling in Java over the subsequent JDK versions. It also helped us to cope when the domain revealed surprising complexity. For instance, a World Time application works so that it shows the current local time plus a user-defined offset. When editing the World Time, the time displayed is the local time plus the offset, but the changes made while editing only affect the offset. Time units thus roll over when the local time plus the offset reaches the maximum value. If the offset is +6 hours, the local hours value equals 17, and the user increments the hours, the displayed figure must decrease from 23 to 0, but the offset must increase to +7; not for instance to -17, which has the same value modulo 24, but a very different effect on any existing timer alarms that might be counting down.

12.2.2 Extending a Framework from One Platform to Many

As we saw in Chapter 9, the Watch example was designed and built purely to run as a Java applet in a standard desktop OS. When we decided to extend it to work as a Java application in a cell phone, many of the initial assumptions were no longer true. We could no longer use text input fields to show the time, nor buttons that the user could click with a mouse. The main application class would no longer be an Applet, but a Midlet, which worked rather differently. Compilation would no longer be as simple as “javac *.java,” but would require a number of new phases and configuration files.

Perhaps most challenging, we wanted the same models to be able to generate an application for either platform. Or rather even more: we did not just want to be able to get to a state where one set of models could generate both, we wanted the *existing* models, built before any idea of MIDP was considered, to work on both platforms. In a real commercial case, this would have saved the modelers (of whom there would be many) having to rework each of their models. While achieving this may require more work in the DSM solution (the metamodel, generators, and framework), the benefit would be multiplied over many modelers, each with many models.

Of course, if the claims of DSM are true, and if the Watch DSM solution had been made according to good DSM principles, this requirement should be feasible. The watch models would contain only information about what the applications should do, and no implementation details of how they would do it or on what kind of platform. Happily, it turns out this was the case, and the results are seen in Fig. 12.3. You’ll just



FIGURE 12.3 Watch in Nokia, Motorola, and Sun MIDP emulators, and browser applet

have to take our word for it that the Watch example really was our first shot at making a DSM example to accompany MetaEdit+, and not for instance the only one out of 27 such examples that we could actually get to work!

You may recall that the authors of the existing framework, while experienced programmers, were Java novices. The original code was thus unsurprisingly in need of refactoring, and the new platform provided a good reason. First, we refactored out the mass of user-interface, control, and state machine behavior into their own classes. From this, it was easier to see what had to be done.

The majority of classes were platform independent, requiring only basic Java functionality. The user interface and control Application Program Interfaces (APIs) are different for MIDP, with only very basic widget support. The widgets for displaying the time and icons were thus replaced with a lower-level set of text drawing operations in a new WatchCanvas class. As the text and icons had to adapt to different MIDP devices' fonts and screen sizes, it was soon noticed that a similar WatchCanvas class could also replace the old widgets in the applet version. This resulted in smoother updating in the applet, as well as keeping the applet and MIDP versions more visually similar.

12.3 PROVIDING AN INTERFACE FOR THE GENERATOR

The generator takes input in a format specified by the modeling language, and produces output that must run under the domain framework. Since both modeling

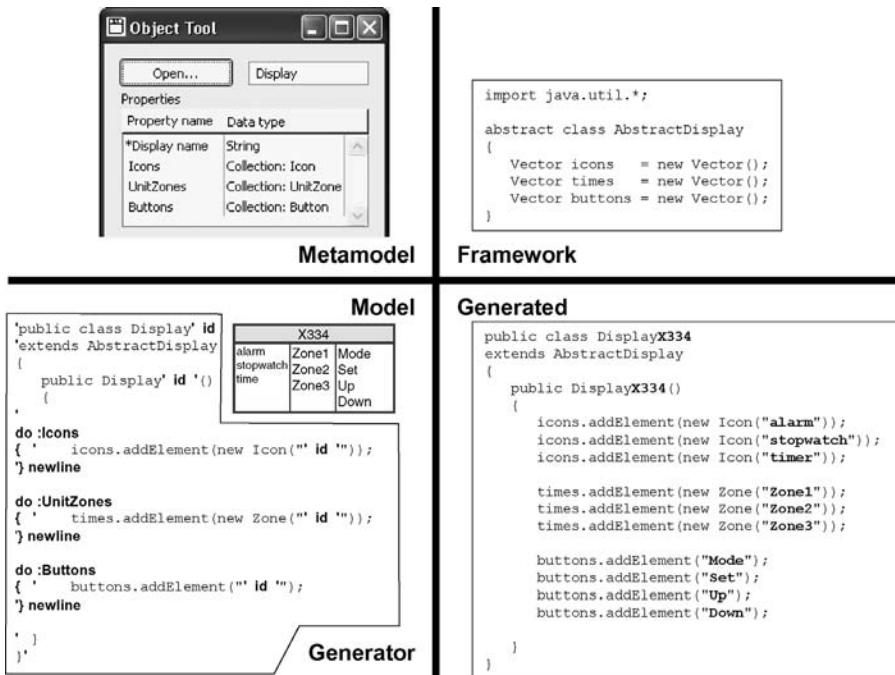


FIGURE 12.4 Parts of a DSM solution

language and framework are created specifically for the same problem domain, there is often a good correspondence between them. The better the correspondence is, the easier it is for the generator to map from its input to its output. In Fig. 12.4, there is a good correspondence between the Display modeling concept in the metamodel and the AbstractDisplay class in the framework. The three lists from the model can thus map one-to-one with the lists in the generated code: the only change is that the modeling language calls the middle list “UnitZones,” whereas the Java class calls it “times.”

An exact one-to-one correspondence may not however be desirable in all cases: It leads to a domain framework that has to be completely driven by data structures that mimic the model. While completely data-driven code will work, it requires a large amount of work in the domain framework, can be harder to follow, and is almost always slower than more traditional code. Parts of the generated code may well be data-driven, for instance, the basic structures of a state machine. The Watch example in Chapter 9 used a data-driven approach for the Java state machine, but more procedural code for the C state machine. As a general guideline, the lower the level of the target language you are generating, the less likely it is that you will choose a data-driven approach.

How can the domain framework offer an interface for the generator to use? Largely in the same way as any component framework: it can provide utility functions, data structures to be instantiated, ways to attach behavioral code generated from the

models, customizable and extensible components, and an architecture that works with all these. Utility functions will be familiar from other frameworks, and were covered in Section 12.1, so we will concentrate on the other ways.

12.3.1 Data Structures to be Instantiated

In many cases at least some of the modeling language concepts will appear in the domain framework as data structures with similar contents. This is obviously the case for a generator for a modeling language that offers a graphical front end for an XML schema. There each object type will tend to map to an element with attributes or subelements for each property, as we saw in Section 11.3.4. Generation of other textual DSLs will work similarly. Even where the generator produces mostly executable code rather than data, some parts of the model will normally be represented directly in data structures.

In an object-oriented target language, these data structures will be instances of classes defined by the domain framework. The class structure will mirror the structure of the corresponding type in the modeling language. Properties will be mapped into member variables of the class, and some of the relationships and similar links to other model elements may also become member variables. Relationship types themselves may sometimes map to their own class, with the main member variables holding the objects they connect, and extra variables for the properties of the relationship and roles.

Whatever the type of the target language, such data structures will have associated behavior. Perhaps more commonly than in traditional frameworks, an important part of that behavior will be the accessors and constructors. The model contains data in a format that is easy for a human to work with, but the code that will use these data structures will want to access it in a machine-friendly format. That transformation can be accomplished either by the generator, the constructor and/or setter accessor, or the getter accessor.

In all but the simplest cases, the generator would be a poor place to carry out this transformation: it is better to keep the code that produces the internal format in the same place where that format is defined. If performance is not a major concern for the data in question, it may be easiest to keep the internal format similar to the human-readable format and provide machine-friendly getters. This will keep the data structure definition in close correspondence with the modeling language, and keep instances in a format recognizable to the modeler (e.g., if he has to resort to source-level debugging). Should performance constraints arise later, the variable in question can be changed to a machine-friendly format. If there is a strong need to maintain the human-readable format, an extra variable can be added to cache the machine-friendly format.

Where performance is at all a concern, or when that area of the DSM solution has become stable and is no longer being changed, the domain framework can offer a constructor or setters that take the human-readable format as arguments, and fill the machine-readable format into the data structure. In most cases, such data structures are read-only after construction. If they will be modified later

by the framework—for example, if the model is simply specifying an initial set of values—there should also be direct setters, which would take an argument in the machine-readable format.

12.3.2 Integrating Code Specified by the Model

Only in simple cases can a framework be turned into an application purely by configuring it with data from models. More often, there is a need to specify new dynamic behavior per application. We have seen a number of ways of building modeling languages to capture such information, and in Chapter 11 we saw several approaches for turning that information into code. Now we must consider how the domain framework can best integrate such code.

First though we must address the myth that information can be divided into data and code. All too often we hear skeptics say: “sure, you can generate *that* part, but that’s just data: what about some *real code*?”. The truth is that there is a sliding scale of increasing complexity, with stereotypical data and stereotypical code at opposite ends. Good developers will generally work to push things toward the simpler, data-like end of the spectrum, and thus reduce complexity. In many cases we have seen in industry, the “need” for real code has in fact simply been evidence of a failure to take a step back and look at the wider picture.

To take a more iconoclastic view of things, all code is in fact data, used to configure some framework or engine. C code is the fodder for the compiler: one long string, or after parsing a simple tree of a few keywords with primitive string or integer values as leaves. Database guys can snicker at developers as they point out that even an online pet food store might have more complicated data structures. Going further down in search of Code For Real Men works no better: machine code is even simpler, and the processor just chews that stuff up like a 1960s BASIC interpreter. In a last attempt to salvage some pride, the Real Developer pulls out self-modifying code; the database guy looks on with pity: “so you finally have some data that’s not read only. Of course, we’ve never seen anything like that...”

Clearly, the search for a dividing line between code and data is fruitless—and not particularly flattering to our egos. Let’s abandon any attempt to find a higher truth here, and settle for a pragmatic definition. If a piece of generated text simply instantiates and provides values for a data structure, it’s data; otherwise, it’s code. How can a framework provide ways to integrate such code? Obviously, if the code calls the framework, the problem is trivial, so we shall concentrate on cases where we want the framework to call the code.

First, the framework can be made to call a specific function, and the generator produces that function. This will normally of course be extended to a number of such functions. An example can be seen from the “perform” function used in the Watch applications from Chapter 9, as seen in Fig. 12.5. The AbstractWatchApplication framework class implements a state machine, calling the “perform” method for the actions specified in transitions between states. It defines “perform” only as an abstract method. The code generated from the Stopwatch model is a subclass of that, and provides a concrete implementation of the “perform” method

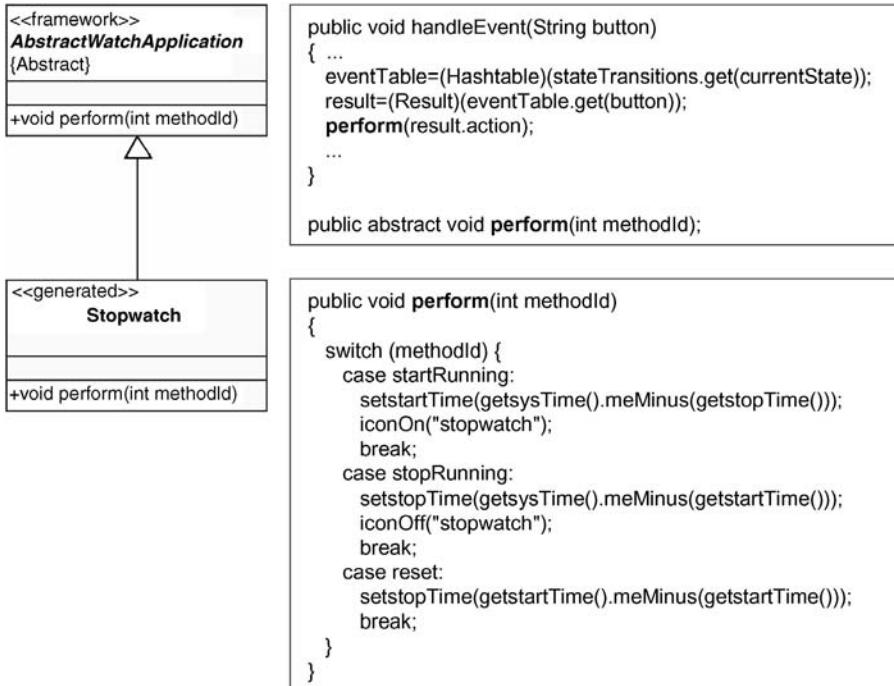


FIGURE 12.5 Framework calling a generated function

to specify its own actions: setting various time variables and turning the stopwatch icon on or off.

Second, the framework can provide hooks, events or callbacks, and the generated code can declare its generated functions as handlers for these. This allows a large degree of freedom and variation in how the application takes advantage of the framework. In third party component frameworks, such an approach demands good judgment and architecture from the framework provider: all too often the point at which you would need to step in and have your code executed is not offered by the framework. In a domain framework, that is easily remedied; a good architecture is however still needed.

Where the implementation language permits it, the second way can also be extended far enough that it could be considered a new, third way. The framework can offer data structures, some of whose members are pieces of code. Some languages, for example, C, allow named function pointers to be used in this way; others such as Smalltalk allow arbitrary anonymous blocks of code. Java can use inner classes, although their syntax and the multiplicity of compiled files leave something to be desired. Fig. 12.6 shows an example of inner classes being used to accomplish the same behavior as the overriding of the “perform” function above.

A fourth way to integrate generated code into the framework is via direct injection: changing the framework code itself. This is only viable where only one piece of the

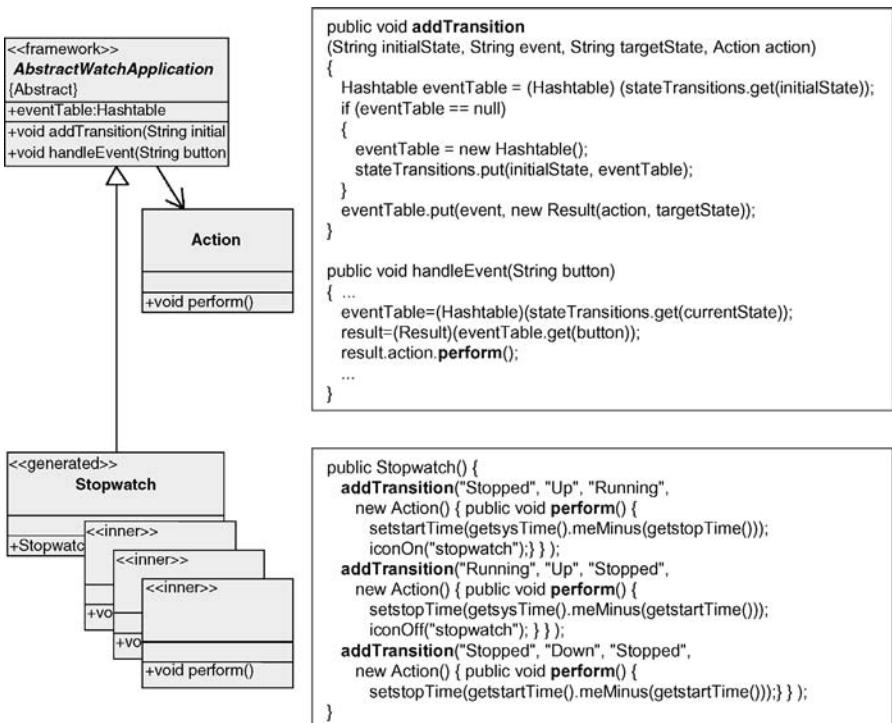


FIGURE 12.6 Generated function hooking itself into framework

application wants to change any given piece of the framework. If multiple conflicting changes were needed to the same piece of the framework, there would be no way to reconcile them to satisfy all the requirements. While the framework could in theory be duplicated for each case, making the application contain multiple edited copies of it, in that case it could hardly be considered a framework.

We have seen cases where a framework is literally edited via a script or macros, but this is best saved as a last resort (or for those who find themselves unaccountably short of war stories). Some languages, for instance C, offer a preprocessor which allows the definition of macros. The framework can then contain references to the macro, and the generator defines its code into the macro definition in a separate file. Aspect-Oriented Programming systems allow the use of separate code files which are then woven into the framework at specified points. Java is particularly strong in this field, in part because of the language's tendency to forbid many tricks commonly used to influence already compiled code. C# allows the use of partial class files: the framework could contain the main class file, and the generator could produce an extra partial class file for that class, adding new behavior.

These approaches are best used to provide reasonably limited amounts of code to the framework. Should you find the need to add significant amounts of code, or add code that also defines new or extended data structures, you will probably want to move on to the next section.

12.3.3 Customizable and Extensible Components

In many object-oriented component frameworks, a large share of the workload is shouldered by abstract classes in the framework. The developer building on the framework uses and extends these classes by subclassing them. This allows the full range of behavior customization, extension, and modification:

- A subclass can fill in data structures defined in the superclass.
- A subclass can provide concrete implementations of methods defined as abstract in the superclass.
- A subclass can add new data members for its code to use.
- A subclass can add new methods for its code to use.
- A subclass can override the accessors defined by the superclass, to retrieve data from elsewhere or manipulate the values returned.
- A subclass can override other methods defined by the superclass, to change its behavior.

This can all of course be carried out on a number of levels:

- An abstract class in the platform or component framework can be subclassed by generated code.
- An abstract class in the platform or component framework can be subclassed in the domain framework to provide the commonalities required by generated applications. The result could be a concrete subclass that is instantiated and used by generated code, or then itself an abstract class to be further subclassed by generated code.
- If parts of the model reuse other parts of the model, this may correspond to generating subclasses of generated subclasses.

As can be seen, this allows a vast array of different approaches and tactics, allowing you to produce applications in almost any way you want. This is of course good news for you the reader: you can generate full code without it becoming bulky (move common parts to the framework) or ugly (replace hacks with a good architecture in the domain framework). It is however bad news for us as authors: with such a broad vista of possibilities, it is hard to cover the ground or give concrete advice. Fortunately, at this level building a domain framework is similar to building any other kind of components, so your existing experience will be valid and useful, as will any good books on object-oriented programming.

If this is your first foray into DSM, you may well have found many other previous beliefs about standards, modeling, and code generation being challenged. Hearing that there is an area of DSM where things are pretty much “business as usual” will no doubt come as something of a relief! Without wishing to disturb you from your reverie, we do want to provide a few signposts to small ways in which the domain framework and generated code may be a little different from what you are used to.

Generated Code Contains More Singleton Classes When you look at code generated to run on a fair-sized domain framework, you may well notice a larger than average number of singleton classes. They will be subclasses of a domain framework class, often in quite a wide inheritance tree, and each may add only a small amount of code. This may appear disconcerting: if you saw the same in handwritten code, you would look suspiciously at it and try to apply various best practice strategies. What could be moved up into the superclass to allow that single class to be used for all of the instances? Or which of the subclasses was really the same as another? Or perhaps the subclasses could be arranged into a deeper hierarchy, with extra abstract classes containing the features shared by several subclasses?

In fact, assuming the framework and generator have been made well, you are actually seeing the end result of those very same strategies—just applied far more than normally happens with handwritten code. The subclasses look small because every last drop of commonality has been squeezed out of them into the superclass. None of them will be the same, because in that case the modeler would have reused the corresponding model element rather than creating a duplicate. Finally, the amount of common content in groups of subclasses would be so small that the weight of an extra class would normally not be justified.

The last two points need a little clarification: at any given state of the models, there may well actually exist complete duplicates or sets of classes with common elements. However, the duplication will be purely coincidental, and could change at any time. The duplication is more like that found when two Person objects have the same height and weight: it does not mean we need to separate that “commonality” into a new HeightAndWeight class, with both Person objects pointing to the same instance. Of course, there will be cases where you missed something in the domain analysis, and you really should make the granularity of reuse finer in the models. In our Person analogy, that might correspond to noticing that groups of people tended to share the same surname and address fields, leading you to posit a Family class.

Finally, there may be cases where the singleton classes only differ in name and the values they give to static variables. If that is the case, and the classes really are singletons, it will probably be an implementation choice whether to simply move the name and the static, class-side variables to be instance variables. Examples of this can be seen from the Watch Java code in Fig. 12.4: do we really need a new class DisplayX334, or could we simply change its superclass, AbstractDisplay, so it could be concrete?

Domain Frameworks Contain More Metaprogramming Since most bugs are found in code not data, good programmers generally look for chances to turn code into data. In simple cases this can be easy and natural, but in less obvious cases the code needed to read and enact the data becomes complicated. The use of a programming language with good support for metaprogramming can help a lot here, but still the code requires an above-average programmer. Unfortunately, the vast majority of programmers consider themselves “above average,” just as most automobile users consider themselves better than average drivers . . . To put it another way,

in the right hands metaprogramming is a powerful tool, but wielded inexpertly it can wreak havoc with the readability and quality of your code. For this reason, frameworks are a particularly good place for metaprogramming: it is fair to expect the developer of a framework to be a better programmer than the users of that framework.

DSM offers particularly fertile ground for metaprogramming. First, special cases are particularly irksome in metaprogramming. In normal code, an extra IF statement is neither here nor there, but in the compact yet convoluted coils of a piece of metaprogramming, any extra complication can easily render the whole hard to comprehend. Since DSM has already forced a more thorough domain analysis than normal, such cases can be taken into account in the modeling language or elsewhere, or at least will be known from the start rather than hacked in as an afterthought. Second, metaprogramming—like any data-driven code—is sensitive to illegal or wayward input in its data. The metaprogrammer must either write extensive code to check the data or then crank up their optimism a notch or two. In DSM, the structures and constraints of the modeling language can ensure that only well-formed and valid data can be entered.

12.3.4 Inversion of Control

A common pattern when using frameworks is to switch the basic execution architecture: rather than the application code running the show and calling framework code as needed, the framework itself runs and calls the application code provided to it. This is known as *inversion of control*.

The way the inversion works can perhaps be explained by analogy with sports team coaching. In standard component development, the job of the framework developer is just to provide a set of components for the developers to use as they see fit. Picture a naïve—or progressive—coach offering his team a selection of balls, goalposts, and a goodly supply of the apparently indispensable traffic cones, and letting them get on with things. When a new, more experienced—or authoritarian—coach appears, she runs things very differently: she leads the players through a series of drills, setting out the equipment to be used for each. In other words, she is saying to them, “I don’t just have the stuff you need; I really know how you should use it to get the best results”.

Obviously, this approach demands less intelligence of the team members, but more obedience. This perhaps explains why it is not a universally popular approach among framework users: we developers are not exactly renowned for our subservience, or for taking lightly any perceived slights on our ability. It is, in fact, an approach better suited for use by factory workers, drones, machines—or generators. A good generator might at best be described as smart, but it could never be accused of intelligence: the ability to come up with good, new, solutions to new classes of problems. It is however totally obedient and predictable, and copes remarkably well with the lack of intellectual stimulus in its working day.

Building a domain framework with inverted control can start right from the beginning of the code-related phase of DSM. Back in Section 11.1 at the start of the Generators chapter, we saw how writing a small example application in the domain was a useful first step. The simple approach shown there reduced this application into a generator that produced most of it, and a model that supplied the values that varied

between applications. Later, we saw how frequently encountered or long blocks of code in the generator could be separated out from the generator into components. However, this clearly leads to a situation where the generated code is calling the shots, invoking components as needed.

To invert control, we can look at the example application from a different point of view. What parts of it show the flow of control and execution architecture that will be common to all applications in the domain, or to an important subset of such applications? If a working generator has already been built, it may be easier to spot these parts by comparing several generated applications. You can also look at the generator itself, in particular for any pieces of boilerplate code longer than a line or two, and which are generated only once or a few times for each application. Such parts can also appear in the generator as a larger structure of code that remains the same for all applications, with short one or two line segments of boilerplate interspersed with the details filled in by the model.

For instance, in the switch-case pattern described in Section 11.3.6 there is a clear pattern of two nested layers of switch-case statements to implement a state machine. The structure always remains the same, with the models simply providing different states, events, actions, and transitions. This was also the approach used in the C generator in the Watch example in Chapter 9. To invert control, the underlying structure of the switch statements—see what state we are in and what event occurred, then do the requested action and follow the transition to a new state—is moved out of the generator into the domain framework. The generator simply supplies the definitions of the states, events, and transitions as data, and probably implements the actions as new functions. The framework runs its generic state machine over the data supplied by the generator, calling the generated action functions where necessary. This is the approach seen in the Java generator in the Watch example.

12.3.5 Engines Reading Models as Data

We can also take inversion of control further, to the point where the framework becomes an engine. In this approach, all possible behavior of applications in the domain is covered in the engine. Any given application specifies its subset of this behavior space as data, which is read and acted on dynamically by the engine. In a way, the engine is an interpreter for the data supplied by the models, as opposed to the earlier cases where the generator acted more like a compiler to turn the models into code.

Theoretically, the engine could read the models directly from the memory structures in the running modeling tool. Since this requires the modeling tool itself to be present at runtime of the applications, it is hardly ever used for deployment. It may however be useful in some cases for simulation: the engine runs the models directly, and interacts with the modeling tool to visually trace the execution in the models. This kind of visual trace is however also possible with more traditional generation.

The engine could also read the models directly from the files they are stored in on disk. In practice, the model files will normally contain a substantial amount of information that is redundant for the execution, for example, the layout coordinates of each model element and various documentation fields.

More normally, the models are exported by generators into code that instantiates framework data structures to recreate the relevant model structures, or into a simple file format that is easily read by the engine.

12.4 SUMMARY

The domain framework is a layer of code that sits between the generated code and the existing generic components and platform. Its main function is to avoid repetition and complexity in the generated code and also in the generator.

The generated code thus comes entirely from models, and the framework is entirely handwritten. This separation distinguishes DSM from code generated by wizards, where the application developer is expected to work with and around the generated code, and code that is common to all applications in that domain is duplicated into each application. In DSM, the application developer can ignore the generated code, and the domain framework is referred to by that code rather than being copied piecemeal throughout it. The separation thus naturally follows the separation of the different developer roles.

The domain framework also separates along the lines of generalizability. Below it are generic components, used by a wide range of applications on that platform. In the domain framework itself are the elements common to all applications in this narrow domain: the envisaged application space for this DSM solution. Above it is the information specific to individual applications, that is, what distinguishes each of those applications from each other. Building a domain framework is probably the part of DSM that will be most familiar to developers from their previous experience. It is similar to building components and more general frameworks: by no means an easy task, but certainly a skill that can be learned. The main difference between building a domain framework and more traditional frameworks is the target audience. The “users” of a domain framework will be the generator and its generated code, rather than developers and their handwritten code. This solves a problem commonly experienced by builders of traditional frameworks: they make a wonderful framework, but people often misuse it—or more often fail to take advantage of it or parts of it.

As the use of the domain framework is automated in the generator, domain frameworks are used more consistently than traditional frameworks. This makes it possible for them to go further than normal in removing duplication. For human developers and traditional frameworks the cost of one developer learning a framework feature must be recouped over the number of times that one developer uses that feature. In a domain framework, only one developer need learn that feature, applying it in the generator, yet the cost will be recouped over all developers using the framework. Often, the framework developer will also be the generator developer, so even that small cost becomes almost zero.

The domain framework also separates the generated code from the platform. It hides the platform details, and thus insulates the rest of the DSM solution from changes in a platform. The insulation works equally well whether the change is to a new version of the same platform or to a new platform altogether. This can also be applied when there

is a desire to support several platforms in parallel: the same modeling language, models, and generators can be used for all platforms, with just a new version of the framework being written for each platform. As the size of the framework will almost always be smaller than the amount of code that would be needed for a single application in that domain without DSM, the cost of supporting an additional platform will be comparable to the pre-DSM cost of building a single application. Furthermore, whereas without DSM building a single application on the new platform would offer precisely one application, with DSM every single application built for the previous platform will now also be available for the new platform.

While it is good to be aware of such possibilities, the main focus for the domain framework developer will be elsewhere. As in any framework, the domain framework should provide the algorithms and functions required in that domain. In addition to those, the domain framework should focus on matching well with the modeling language where possible, and on providing a good interface for the generator. The interface will consist of data structures to be filled in, places to add code generated from models, and components to be extended by generated code. The aim is to support the largest possible extent of the desired application space, for the smallest amount of work for the modeller—and to a lesser extent, the metamodeler.

Domain frameworks tend to differ somewhat from traditional frameworks, for instance, in greater use of inversion of control and metaprogramming. There is also often a tendency toward small singleton classes in the generated code using the domain framework. Depending on the domain, there may also be other considerations such as the framework size, generated code size, or total executable size or speed. These will influence the strategies and programming style used in the framework, but in a way already familiar to any experienced developer in that domain.

Building the domain framework can happen at any time in the overall DSM process: as the first step, after the modeling language, with the generator, or after the generator. We have seen and worked on successful cases in all of these different ways. Until you have the experience of a couple of successful DSM solutions under your belt, however, we would recommend that you leave the framework until after the first version of the generator. The main difficulty for people in stepping up to DSM is to leave the code world behind when building the modeling language. Since the benefits of DSM are largely due to the raise in the level of abstraction in the modeling language, it is best to avoid the risk of a familiar area such as the framework dragging the level back down toward the code.

Similarly for the generator: take the time to learn the new mindset required for using a generator language, rather than trying to take a “shortcut” by doing everything in the familiar language of the framework. After all, the generator language itself is (hopefully) domain-specific, honed for the task of generation. So, do not be in a hurry to return to the familiar world of code: DSM involves learning some new things, but a little abstinence from coding will whet your appetite for it. At the proper time, building the framework will more than satisfy that appetite, as you get to use all the skills of your trade. Best of all, as you update the generator to take advantage of the framework, your framework will for once have the discerning user you have always longed for.

CHAPTER 13

DSM DEFINITION PROCESS

In this chapter, we will look at the process of creating, introducing, and evolving the whole Domain-Specific Modeling (DSM) solution. Earlier, Chapters 10–12 have focused on individual parts of the solution, and the case studies in Chapters 5–9 have described what happened in specific instances. Here, we will give an overview of the process as a whole and also look at the various groups and roles involved.

The first steps are to identify a good area in which to apply DSM (Section 13.1) and assemble a team to carry out the work (Section 13.2). The choice of domain can be tested through a proof of concept implementation of part of the DSM solution (Section 13.3). If all looks good, a full DSM solution can be built for that domain (Section 13.4) and tried out in a real-life pilot project (Section 13.5). You can then polish and update the DSM solution based on feedback from the pilot project and deploy it to your organization (Section 13.6). The deployment will already have involved planning for the evolution of the DSM solution (Section 13.6.3), and this evolution will continue throughout the usage of the DSM solution (Section 13.4).

13.1 CHOOSING AMONG POSSIBLE CANDIDATE DOMAINS

Deciding whether to apply DSM in a given domain should take into account a number of factors. The domain itself has surprisingly little effect: the whole idea of DSM means it is highly adaptable to different domains. Similarly, the target platform is

largely irrelevant: anything that takes text files as input is fair game, whether it be a C compiler, a Python script, a DBMS, or an application reading a configuration file. In the areas of the domain and the target platform, DSM offers the leverage necessary to overcome any challenges. The broader contingencies of the situation, however, are areas over which DSM has little control. At least in your first DSM project, trying to radically change your market, business, or organization is generally biting off more than you can chew.

For many readers, there will be only one domain over which they currently have control. In this situation, there is limited choice available: as Yoda would put it, “Do, or do not.” The latter part of the quote—“There is no ‘try’”—is equally true in our experience: those who decide to make a real go at DSM, rather than just dabbling, are the ones who also make it succeed.

Other readers may have several domains within their purview. This generally implies a position in a larger organization, either in charge of a number of areas or as part of a research center or advanced development group. In such cases, the chances are that you will first want to try DSM out in one project—both to confirm it works and to be able to apply it subsequently with greater experience in other projects.

In both cases, it is important to understand the wider forces at work in your organization that bear on the introduction of DSM in a given domain. We will look at the forces through the form of a questionnaire, Table 13.1, together with rough scoring and commentary for different answers that will help you rank your candidate domains and the organizational units responsible for them. While by no means as accurate as similar tests for marital compatibility in women’s magazines, this should at least help you recognize the relevant issues and take their possible impact into account. We would like to thank Laurent Safa of Matsushita Electrical Works, Ltd., Japan, for the initial suggestion and draft of such a questionnaire.

13.2 ORGANIZING FOR DSM

DSM does not require a large amount of resources: the focus is on quality not quantity. When the initial DSM solution is in place, all developers’ work will be bounded by, guided by, and reliant on the work done by the initial team. The team must, therefore, be composed of people who really know their work. This is clearly an area where Brooks’ law applies more than most. Adding lower quality resources to the team will have a large negative effect: probably on the initial creation project and most certainly on the use of DSM.

While the quality of the team members must be high, any innate perfectionism must be reigned in, at least in the early stages. It is vital to try the modeling language out in practice, and for this only the core concepts and support for simple examples are needed. Elegance should certainly be aimed for, but if the domain is messy in parts then the best working solution will have to suffice.

The team will need a mix of domain knowledge and programming knowledge. By far the best mix is found in expert developers who have been working in that domain for some time. They have built up their own mental map of the domain concepts, a set of

TABLE 13.1 Choosing a Domain for DSM**How mature is the target business area in your company?**

- An established business +3 Clear business concepts, needs, stakeholders
- Turning an existing customizable product into a platform for use by others +1 Existing demand for product and its customisation; DSM makes this fast and palatable for those who are not in-house experts
- Development of a new product 0 Risk of the unknown balanced by agility stemming from lack of legacy code and processes
- Creating a new platform for use by others -1 Desire to offer familiarity and lowest common denominator works against DSM. Uncertain commitment, risk of project failure
- A research project -3 High risk of failure from other factors, and no commitment to project if it hits difficulties

How much are development and core business processes in the target area coupled to external organizations?

- Joint venture -3 Serious concerns over equality of commitment and ownership, ability to agree
- Internal except for third-party components and tools +1 Clear control over destiny, components give stability
- Totally internal, build all own components and tools -1 Possibly indicative of fear of commitment, can succeed only if DSM is truly seen as “theirs”

How closely is this domain related to other candidate domains?

- Simplest or smallest of a group of several similar domains +2 Good possibilities for extension to other domains later, or switching if this turns out to be a bad choice
- One of several similar domains +1 Indicative of an important business area
- Different domains work independently of each other -1 Sometimes indicates lack of desire to codify and build on what is known

How much do you customize or configure your software for each customer?

- Extensively +3 DSM will allow you to capture just the variable aspects, and code generation will improve delivery times and the quality of the finished product

(continued)

TABLE 13.1 (*Continued*)

● Only for large customers	+2	DSM can integrate many of these bespoke cases back into the mainstream software artifacts, benefiting both large and small customers
● Often but only in limited ways	+1	Possible small language to describe configuration
● Little or not at all	0	Useful only if DSM will be used for the main application

Do you have good source code examples available, for example for teaching new staff?

● Yes	+3	Offers a basis for the generator and is indicative of a mature domain where development is managed well
● No, but we are always saying we should make some	0	Basis exists, DSM can help bring about wanted change, but process will be harder
● Some available, but out of date and from different times	-1	Often indicative of a lack of stability and of reworking upon reworking, and hack upon hack. DSM may be an effective way out if there is a strong expert developer
● No, our staff are all expert developers	-3 if said by manager +1 if said by developer	

Do you have an in-house application framework?

● Yes, with established guidelines, best practices and sample code	+3	Clear existing framework, expected code generator output, domain concepts
● Yes, the previous version is the guideline	+2	Domain concepts clear, framework quite easy to build from existing code
● No, but we try to reuse legacy components	-2	Would have to define the framework, domain concepts probably also unclear
● No, we develop everything from scratch	Abort	"First of a kind" development, not worth targeting

TABLE 13.1 (*Continued*)**How would you describe the maturity of your software development process?**

- Precisely defined and developers must follow it 0 May be too caught up in CMM and processes to accept DSM or its subsequent evolution
- The majority is documented and we generally follow it +2 Shows a desire to take the effort to codify what is known and follow it as long as it is useful: fertile ground for DSM
- We have a good idea where we are at any given time +1 Developer-led
- No clear process -2 Hard to bring about organizational change without organization

Can you assign the following kinds of people to the DSM project?

- One of the top three who built the framework/first product +5
- An experienced developer +3 Can build code generator
- A small team of normal developers +2 Vital for piloting, can build example apps to be the input for creating generators
- No, but you can have a summer intern or two -3 May have brains and enthusiasm, but lack domain knowledge and software development experience in large teams and long projects
- No Abort DSM cannot be successful without the developers' experience

patterns for mapping combinations of those concepts into code, and a good knowledge of the existing framework and platform. There is also a certain mindset that says it is not enough to be able to churn out good working application code in the domain time after time: there must be a higher level way to express applications and automate their creation. People with this mindset are clearly well suited to the tasks of the DSM creation team. They are the kind of people who build text editor macros, word processing templates, and batch files or shell scripts. Most useful are those who have built such things for use by the whole team, as opposed to only for their own use.

Initial trials with the modeling language can happen within the DSM creation team, using lightweight formats such as pencil and paper or whiteboards. If there is a good existing framework and body of example code, and you have a fast DSM environment that supports modeling language evolution, you can move at an early stage to testing your ideas in practice. One way is to do a short proof of concept project, as described in the next section, and another way is to do an early version of the DSM pilot project

with a DSM use team. Even early pilot users require modeling tool support and normally also full code generation (although manual tweaks will be acceptable at this stage). If building tool support to such a stage would take longer than a few days, leave it for now and concentrate on the modeling language. Alternatively, consider using a faster DSM environment for now, even if later you plan to use one that requires more hand coding. This will allow you to verify your modeling language ideas in practice, before investing the time in coding them.

Take every opportunity to find people to try the DSM solution out in practice as you build it. Creation team members should draw models of small yet realistic situations, and you should press-gang other developers into at least looking at the models and thinking about them. While they may not yet have a formal part in the project, keeping the communication lines open is an important factor in later piloting and deployment. Making it clear within the team and to other developers that you are initially aiming only for the simple, common cases will help keep expectations realistic. It will also help to keep the creation team motivated and focused on delivering real value, so it can be used soon in a real pilot project to benefit developers.

As the modeling language and the code generator come close to being able to make simple real applications, start looking for a pilot project and team. The pilot project should be a real application that is important to the business, in the sense that it would have to be built anyway and will form part of a real product. It should not, however, be on the critical path or be strongly coupled with other projects. Wherever possible, the pilot project should be one that is already well understood: the kind of job that you might give to a new developer.

The key criteria for the pilot team are motivation and discipline. While the team need not be gung ho DSM enthusiasts, they must be willing to give it a go, and able to cope with the setbacks and delays that accompany any new venture. Since the modeling language and generators will be unfinished and changing rapidly, the pilot team must be disciplined in following instructions, updating to newer versions of the language and generators, and reporting the issues they encounter.

13.2.1 Management Support

The first steps with DSM can often be performed within the bounds of existing resource allocations, but full implementation will normally require management support. Since there is usually little chance for a team to select or instruct its own management, we will restrict ourselves to looking at what can be requested from and expected of the existing management. In the broadest terms, we are looking for support from the management for the process of evaluating and implementing DSM, for as long as DSM appears to be a valid solution for this area. Management support can have different forms:

- They propose candidate domains that they feel most fruitful to support with DSM.
- They agree to the implementation of a proof of concept and commit to following through if it proves successful.
- They allocate appropriate resources to the DSM process.
- They want to own the consequences of the findings.

The last point is particularly important, if hard to define: the more that management feel personally invested in the success of the DSM project, the more likely they are to allocate the resources needed to make it succeed.

13.3 PROOF OF CONCEPT

Moving an organization from coding to model-based development is rarely so fast that you can accomplish it before anyone notices and begins questioning the wisdom of the project. Many people have preconceived prejudices based on older approaches with fixed modeling languages: modeling is a waste of time, you should only use “standard” modeling languages, generated code is bloated, slow, and incomplete, and so on. You will almost certainly need to face such objections at some point, so you may as well get them out of the way early on. Aside from waving books like this and glossy vendor literature at people, it is also useful to show concrete results right from the start.

A good approach, assuming a sufficiently fast DSM creation environment, is to do a short proof of concept project within your DSM creation team. If you can get someone experienced in DSM, either an external consultant or someone from elsewhere in your organization, this need only take a couple of days. If it is your first DSM project and you have no such mentor, a week would be a reasonable time.

The goal of the proof of concept project is to define and implement a small, but still significant, part of your DSM solution. At the end of the project, you will have a meeting where you demonstrate the results and benefits with a concrete example to your executives and product developers. The result of the workshop is a partial modeling and code generation environment for your organization’s own domain-specific modeling language.

13.3.1 Preparation

Whether you are working on your own or with a mentor, the first phase is to outline the scope of your DSM solution and gather the materials that form its requirements. Table 13.2 shows a suggested template for collecting this information. Providing the answers is a good first task for your DSM creation team, and an effective way of checking that you have the necessary expertise on the team. If you find yourself having to ask others for some of the answers, you should consider whether those people or others with their knowledge should actually be a part of the team.

Set a time for the workshop and arrange a place where your team members can work uninterrupted by normal business matters. Most importantly, arrange a meeting with management and senior developers for right after the end of the workshop. When you need to produce concrete results rather than interminable wrangling, there’s nothing better than knowing that you and your work will shortly be on public display. The aim of the workshop is not to get everything right, but to get something working.

Before the workshop, you should make sure that all members of your team understand the ideas of DSM, that all have seen some real examples of it, and that the majority have at least basic experience with the tools and languages of your DSM creation environment.

TABLE 13.2 Proof of Concept Template

Proof of Concept Workshop

The aim of this document is to collect the key elements of your domain for the purposes of the domain-specific modeling language implementation workshop.

The goal of the workshop is to define and implement a small, but still significant, part of your domain-specific modeling language to demonstrate the benefits with concrete examples for executives and product developers. The result of the workshop is a partial modeling and code generation environment for your organization's own domain-specific modeling language.

As domains differ widely, not all questions may be strictly applicable in your domain: feel free to change the questions or omit irrelevant parts.

1. Introduction

Please give a short introduction to your domain area. What parts of the work in the domain do you want to include in your modeling language, and why are the benefits of domain-specific modeling important in these parts?

2. Usage

Describe briefly how you intend to use the modeling language: what is the input on which the models are based, who makes the models, what is generated, and what other tools are involved. Where possible, give approximate indications of how many users, models, files, and so on there are, and how much various parts are reused between products, features, or models.

3. Sample material

Pick a small but representative existing example whose implementation would take about one week for one developer. A good example is often the kind of feature you would give as a first task for a new programmer. The example should include the major, most central elements of the domain.

Describe the example briefly here, along with the approximate time to implement it, any notes about special features of this example, things that have changed since it was made, and how it fits in with other related applications. Then continue to fill in Sections 3.1–3.4 with more specific information (attach separate documents where necessary, making links to them from this document).

3.1 Sample requirements

Give the requirements for the example functionality, including how it interfaces with other parts of the system.

3.2 Sample design

Give the design documents, including graphical models and text where available. As far as possible, the design should be in step with the code below.

3.3 Sample code

Give the code for this example. If there are several files then also provide an overview of what each file does. Where possible, comment the code with references to the design documents. These comments will help in analyzing which parts of models could produce which parts of code.

TABLE 13.2 (*Continued*)

3.4 Sample user's manual	If the code will have a human user present when run, either using or monitoring the application, give user instructions for that person using this code.
4. Rough modeling language sketches	Draw the diagrams below in whatever is the easiest format: hand drawn is fine.
4.1 Product contents	Often a product (or range of products) consists of several parts (features, modules, etc.). Sketch a (partial) typical product along with its parts and subparts. Mark which parts stay largely the same between products (e.g., existing code libraries, components) and which parts are to be modeled in the other diagrams below.
4.2 Product behavior	Sketch a diagram of a typical feature (possibly the sample feature above) as might be drawn by designers on a whiteboard at an initial design session. Try to avoid "standard" design languages such as UML, which concentrate on the resulting code components, and use informal concepts commonly used and understood by designers. Sometimes there may need to be more than one diagram at this level, for example, one for the user interface and one for the behavior.
5. Other relevant material	

13.3.2 During the Workshop

The activities during the workshop will unsurprisingly be an application of the material in Chapters 10 and 11, and to a lesser extent in Chapter 12, following a miniature version of the process described below in Section 13.4. The key component is the modeling language, and the key way of judging it is that it captures all the information needed for the sample application, and that by changing some of that information you would be able to describe a reasonable range of different but similar applications. You must resist the temptation to work on the more familiar areas of the framework or code generation, or to spent too much time on the graphical representation of the modeling language.

On the basis of our experience over several dozen such workshops, generally over two days, we can give some rough indications of the timetable. On the first morning, we generally spend about 45 minutes getting to know people and making sure they understand the main principles of DSM. The rest of the morning is spent sketching out various decompositions of the information of the domain in general, and the sample application in particular. This gives us some possible starting points for the modeling language—or, if the domain seems to require it, modeling languages. We look at the code of the sample application to recognize parts that are repeated, common to many applications, or variable depending on the application. At this stage, we are not thinking about code generation but about making sure that the modeling language will be able to capture the information we need.

After lunch, we decide on a sketch of the modeling language and draw a sketch model for the sample application. As we get close to something that seems to work, we start defining the concepts of the modeling language: their names and the information that each has to store. So far, even with the fastest DSM tools, the work has generally been so likely to change in major ways that the most effective representation medium has been a whiteboard or paper. The whiteboard has the advantage that you can erase minor mistakes to keep things readable; flip charts are a little messier but give you effectively unlimited space, as you can tear off pages and stick them to walls.

When the concepts and connections of the modeling language have taken shape, we turn them into a metamodel by inputting them into the DSM environment. For now, the graphical representations of the objects will be minimal: colored geometrical shapes with text fields within them to show the values of their properties. While part of the team is inputting the metamodel, another part often fleshes out the sketch model of the sample application on a whiteboard.

When the metamodel is ready, it is passed off to the model group, who create this first model with the new modeling language in the tool. As they are working on that, the metamodel group start looking at code generation. By the end of the first day, there is always a metamodel and a model, and generally a code generator that produces a significant proportion of the code. Parts of the code produced will still be fixed, rather than varying properly based on the information in the model.

Since as visitors we are generally staying in a hotel for the night, we normally take the chance to work in the evening on things that are objectively less important but, in practice, give a disproportionate benefit: pretty symbols for the modeling language, and slides for the presentation in the meeting the next day. We sometimes do a little refactoring of the modeling language or generators if they need it, but nothing major.

The next morning the whole group gets to ooh and aah over the pretty symbols, but the mood is generally tense. The meeting is scheduled for shortly after lunch, and so far the generators seem far from producing full working code. Looking at the code produced and imagining the time it would take to write the rest by hand and get it working, the team is feeling understandably nervous. Quite often, on the second day some team members have to go off to do other urgent tasks, but the scope for parallel or group work at this point is more limited anyway.

One developer generally works on the generators, running them on the model and comparing the output with the known good application code. A second developer may work on a thin domain framework layer, writing functions to reduce duplication in the generated code. If there are other developers, they may be set to work normalizing the known good application code into the more consistent format the generator will produce, debugging problems in the current code produced by the generators, or writing batch or make files that will build the code into an executable.

If all goes according to plan, lunch will be a well-earned respite, safe in the knowledge that the code compiles and runs perfectly, and all that remains is to polish up the demo for the presentation. More often, there will be some problem that necessitates a short or nonexistent lunch break for at least some of the team (normally the leader, either internal or external), and panicked debugging to find out why the darned thing doesn't work. It's all part of the fun, and just goes to prove that DSM is part of the real world, not some utopian fantasy.

By the time of the meeting, at least some of the team are rapidly becoming DSM experts. The idea has clicked, and they are churning out new bits of generators or extensions to the modeling language at an astonishing rate. Nothing is more guaranteed to bring sweat to the brow of a DSM consultant giving his introductory spiel to senior management than to see a team member on fire, making changes to the modeling language and generators that had just been made to compile successfully five minutes earlier. And nothing is more gratifying than to be able to say “and you saw that when we pressed that new button, the whole user interface changed to Swedish—and that’s functionality that Fred there added from scratch while we’ve been sitting in this meeting.”

13.4 DEFINING THE DSM SOLUTION

After the proof of concept, the team building the solution can see that DSM is indeed applicable in their domain. Often this may come as something of a shock to some team members, who perhaps have turned a blind eye to the inefficiencies of previous practice and were certain that nothing major could be achieved. Management will also have seen the results and have hopefully allocated sufficient high-quality resources to make it possible to build the full DSM solution. The company is on the brink of making what is probably the largest increase in productivity in its history.

13.4.1 Danger: Pitfalls Ahead

...there is nothing more difficult to carry out, nor more doubtful of success, nor more dangerous to handle than to initiate a new order of things.

Machiavelli

All is not dancing on beds of roses, however—or at least the roses may be thornier at this point than you might imagine. The main dangers are ones of excess, of the team or management “losing their heads” in various ways. Let’s first deal with the easiest (and rarest): an excess of riches. This can be seen with anything new, where management goes overboard on the idea: the whole business must be turned around to follow this wonderful new approach. The best developers are press-ganged into a group tasked with creating a single all-embracing modeling language for every piece of development the company will ever do. This is simply a bad idea. Accept management’s enthusiasm, but try to curb their excess and focus instead on getting the best developers working on the best candidate domain. Once that is in production, there will be a lot more knowledge and experience to apply to subsequent domains. Since maneuvering managers is rather like herding cats, you may want to take a sneak peek ahead to the hints on organizational change in Section 13.6.1.

The second danger is to think that you are already home and dry: that the DSM solution from the proof of concept is the right one. Even in the best case, where the domain is well understood and you have had the assistance of someone experienced in DSM, there will be much work left to do. You may well be best considering the proof of concept as “build one to throw away.” After all, it only took you a few days to build,

so there is no great loss. Even if you try building a new language on a totally different tack, discover that it does not work, and return to something more like the proof of concept, you will have gained far more than you lost. Understanding how the DSM solution meshes with the domain, and the various forces that pull it and constrain it, will be invaluable during its further development and evolution. If, however, this understanding is not forthcoming and you feel you are thrashing around, you may want to consider getting someone experienced in DSM to come in and help you get off to a good start.

For the other dangers we could give you a dry bullet list, but we hope you will forgive us if we try something a little more colorful, based on a story by one of history's most famous public speakers.

³Listen! A farmer went out to sow his seed. ⁴As he was scattering the seed, some fell along the path, and the birds came and ate it up. ⁵Some fell on rocky places, where it did not have much soil. It sprang up quickly, because the soil was shallow. ⁶But when the sun came up, the plants were scorched, and they withered because they had no root. ⁷Other seed fell among thorns, which grew up and choked the plants, so that they did not bear grain. ⁸Still other seed fell on good soil. It came up, grew and produced a crop, multiplying thirty, sixty, or even a hundred times.

Mark 4:3–8, Holy Bible (NIV). © 1973–1984 IBS. Used by permission of Zondervan.

The proof of concept has sown the seed of DSM, but in many cases that seed is just left to fall on the ground where people stand. To make seed grow, it has to be planted and tended—just leaving it and hoping it will happen on its own does not work. Our attention flits from one topic to the next, and tomorrow will bring us back to earth with all the normal demands of our current project. These demands will eat up our attention like the birds ate up the seed. There must thus be a clear commitment from managers, accepted by the DSM development team, to actually make a serious project out of DSM. There may be other more pressing concerns, so the project need not start instantly. It is, however, certain that there will never be a time when there are no more immediately pressing issues. While the issues may be more pressing in the short term, it is unlikely that any of them could bring the return offered by DSM.

A second form of not taking DSM sufficiently seriously is to start a project, but only allocate a student, summer intern, or new employee to work on it. For all their brain power and enthusiasm, these people lack the necessary experience in the domain, the local code base, and the realities of team development. Because of this “shallow soil”, and the small size of the team, they may be able to build a modeling language quickly: there will be many things that simply will not occur to them as issues they need to take into account. Thus while the language may appear quickly, in the hard light of day it will not stand up to the test. If management or developers are particularly lacking in perception, this failure may even be perceived to be a failure of DSM in general, stopping further attempts with a more sensible team from getting off the ground for years to come.

What then do thorns correspond to? They are the things that are already growing there, which do not produce much in the way of fruit. In DSM, this generally comes

down to existing coding or modeling practices. If the development of the DSM solution starts to look too much like these, they will certainly choke its effectiveness. Sometimes the modeling language is built based on the concepts of the code world, either creating a new set of concepts or reusing those from UML. On other occasions, the generator becomes entangled in requirements that its output must be character for character identical with a given piece of existing code—as if all code in the organization were already that standardized. To be successful, DSM must build on existing code and knowledge, but it must also tear itself free from these constraining thorns where necessary. With suitable ingenuity you might be able to dress a thorn up to look like wheat, but you will never get it to produce a crop.

Thorns can also be things whose seeds are always floating around in the air, and which will take root quickly if someone has prepared some good soil. In many larger companies, there may well be a person or group with a pet theme such as “quality,” “process,” or “architecture.” While these are all good things in themselves, they can choke useful work if they become an end in themselves. As DSM includes elements of all these keywords, it can appear a ripe target for hijacking to further these ends. Enthusiasts for these topics can thus be welcomed on board, but not allowed to take over.

Finally, we have the case where the seed falls on good soil, and things are carried out in a reasonably sensible way. Then we see one of the minor miracles that DSM shares with farming: there can always be disasters, but in most cases, with reasonable effort, care, and attention, you can obtain remarkable results. Maybe your results will not be as glaringly brilliant as the county prize winner—we only know of a few cases where people have claimed a hundred-fold increase in productivity, and only one from our own experience—but anything that gives an improvement of over 100% has to be considered a great success.

13.4.2 First Things First: The Modeling Language

A DSM solution will consist of several elements, as we have already seen: a modeling language, code generators, and domain framework, and the processes and tools that support them. Chapters 10–12, 14, and 15 cover these in depth, including to some degree the process involved in creating them, and how they relate to other elements. Here we shall thus focus on how these elements work and how they are constructed as a whole.

The key element is the modeling language. This is the medium through which development work will take place: the code generators and framework are largely invisible to the modeler. The modeling language is also the schema in which products will be represented. A change to that schema will have larger implications for the existing models and the accumulated experience of the modelers than a change to the generators or domain framework. To put this another way: if we mess up when building the generator or domain framework, we will probably be able to correct the problem with little cost and with an instant and global effect; if we mess up in the modeling language, the cost of updating models will be significant, and that update may not be automatic.

As in the proof of concept, it is thus worth spending more time at the start on the modeling language. We can divide the modeling language into three areas: its concepts, its rules, and its visual notation.

Concepts Of the three areas, the main one to concentrate on is the concepts and their interrelations: the abstract syntax of the modeling language. The concepts define what information models in this language can store from the domain. The interrelations specify how the information is interlinked, for example, via relationships between objects or an object being described in more detail in a lower-level diagram.

Start by defining the concepts. Try to identify the main things that form part of each product or feature in the domain. Add the clearest and most central properties for each concept: what information needs to be stored each time that concept occurs. Look how occurrences of the concepts can be linked to form a whole product or feature, and classify the different kinds of linkages to create the relationships.

As the number of concepts and relationships increases, think a little about how you may want to reuse elements of a model in other models, and which clusters of concepts and relationships have high cohesion and low coupling with other clusters. This will give you hints as to whether you need more than one kind of graph, each with its own set of concepts and relationships (possibly with some overlap). Try also to think whether a model for a largish product or feature would likely become too large to be manageable. If so, consider adding a concept that simply points to a submodel, to allow model layering or a decomposition of a large model into a hierarchy of smaller models.

Finally—for the moment—consider the variability between products, features, and instances of concepts. What is it that needs to be recorded about these differences? What extra information is needed to generate code, or to help the generator decide which of several similar variant ways of doing something it should follow? Add support to the modeling language to capture this information, for example in the form of extra properties on the concepts, relationships, and models.

Rules Rules are distinct from the interrelations in that they do not increase what information can be stored, but instead they prevent certain kinds of information content from being stored, since it is considered illegal or undesirable in the domain. For many metamodelers, there is a considerable temptation to build many rules to protect the modelers. Now is not the time for that. Indeed, experience has shown that the number of rules in a modeling language tends to decrease over time. Modelers find that the tool prevents them from creating a certain structure, but they have a clear idea of what that structure would mean. The rule is thus removed from the modeling language, and handling for that situation is added to the generator, making the DSM solution simultaneously both simpler and more powerful.

Notation Some notation will be necessary to allow the DSM team and modelers to work sensibly with the developing modeling language. For now, the notation should concentrate on presenting the most relevant data content of each concept in a sensible and balanced fashion. The symbols for different concepts should be easily

distinguishable, with central concepts or relationships being more visible. With a good DSM environment, it should be quick and easy to achieve this with simple variations in shape, line and fill colors, and line width. Fortunately, it is rare for metamodelers to go overboard with wonderful gradient fills and flashy bitmaps at this stage: clearly, there is no sense in polishing the symbols until the set of concepts is stable. Basic symbols will suffice for now.

There are, however, two things to avoid even at this early stage. Do not make all concepts' symbols differ solely by color: the brain perceives such elements as representing minor variations of the same thing. Similarly, there is hardly ever a need to include the concept name as part of the symbol: DSM leverages the brain's ability to recognize things as in the real world. Real-world tables and chairs do not come with such labels, nor would the brain recognize them from the labels if they did (try switching the labels if you don't believe us!).

13.4.3 Example Models and DSM Use Scenarios

A language is nothing until you use it to say something. Even invented languages like Esperanto or Klingon stand or fall on whether there is anyone willing to speak them. The textual programming languages we use today give good examples of languages that have been designed well and had the rough corners knocked off them through use—and of languages where one or other of those processes has failed. As we saw in Section 10.7, DSM languages are no exception: we need to test the language by building example models with it.

If the example models seem to be working well, with the language providing a good way to model individual applications or features in the domain, we can consider the next step. If the domain is relatively new, the number of people that will use the language is relatively small, and your DSM experience relatively thin, our suggestion would be to move on to the generators and domain framework. In more challenging cases, and with a little more experience, there is another area it would pay to look at first: DSM use scenarios.

A DSM use scenario looks further than the issue of how the modeling language can be used to build a single application or feature. First, it includes the broader picture of the usage of the whole DSM solution: the modeler, the tool, the models, their representation in model files, the generated code, and the finished products. Second, it expands the scope to look at multiple instances of each of these: how modelers, models, and generated files for different features relate to each other.

Looking at the broader picture of the whole DSM solution lets us spot possible sticking points for the introduction of DSM to the organization. Comparing the various phases, entities, and artifacts to current code-based development also reveals what organizational change will be necessary to adopt DSM. Envisaging the development process of a single product with DSM helps us identify gaps in our thinking: things that need doing, but for which we have not yet assigned resources. It may also suggest ideas for further automation of development: phases that are currently carried out manually by rote, but which the greater precision of DSM would allow us to automate. Sometimes such improvements may be possible with the

information captured by the existing modeling language, but more often we will be able to accommodate them with minor additional properties in graphs.

Perhaps more important is looking at the issues that appear when we have multiple models. At its heart, this is a question of reuse. By its very nature, DSM has already provided a massive improvement in reuse and, in fact, has automated this so modelers do not even need to remember to reuse something. Putting an object in a diagram *is* reuse: all the work of the metamodeler in the modeling language, generator, and domain framework is being reused. Once we have multiple models, however, we open up the possibility for reuse of models or model elements.

Reuse of models or model elements is similar to reuse in normal code-based development. Developers need to be aware of what exists that can be reused, know how to reuse it, and how to make their own components available for reuse. Good modeling tools can, however, bring many improvements compared to code-based reuse. Perhaps the main improvement is that now components can be reused by direct reference, rather than by indirect reference through typing the same name. This goes beyond what even the best refactoring environments offer: nobody need ever think about the name change and what to do, everything just continues to work.

Reuse is discussed in more detail in Chapter 15: for now, the important thing is to make sure that if there are parts of a set of models that will be reused when building another product, these parts can be clearly separated. Reusing “every other object” from a diagram is about as possible as reusing “every other line” from a piece of code. This is thus another place to apply the ideas of commonality and variability that you have already used in defining your modeling language. This will allow your modelers to reuse a model by linking to it, rather than by duplicating it with copy–paste and changing parts of it.

13.4.4 A Balancing Act: Generators and Domain Framework

As we discussed in Chapter 11, the temptation is often to start work on the generators too early. Generators can safely be left until the modeling language and its metamodel are stable. There can still be minor changes, for example, adding a property or maybe a less important object type, but there should not be any notable refactoring of the names of concepts or the division of responsibility between the concepts. Similarly, wait until your example models prove you are happy with the central concepts and how they work in practice. If possible, wait until the use scenarios show that your language, its models, and use process will scale to fit your envisaged needs.

Starting on the generators too early will at best mean significant wasted work for you. At worst, the desire not to redo work on the generators will prevent you from making changes that need to be made to the modeling language. This would mean significant lost productivity over all modelers, over all products, and over the whole life span of the modeling language. Of course, if you find yourself in this situation, you must simply bite the bullet, make the changes to the modeling language, and update the generators—but holding off on the generators until the metamodel is stable is clearly the better strategy.

While the definition of the generators and the domain framework is divided into Chapters 11 and 12, respectively, in practice they are often developed in parallel or even in reverse order. The most common pattern, and the one we would recommend for first timers, is that after successfully writing a part of the generator, you notice that it produces similar blocks of code many times in the output. You abstract that block into its own function in the domain framework and replace the block in the generator with a call to that function. In other words, as you work on the generator, you effectively refactor the generated code, extracting a repeated block into the domain framework. Note that this is different from the refactoring you will do on the generator itself, where repeated blocks of *generator* commands will be refactored into subgenerators.

Since there are no hard and fast rules for ordering the creation of the generators and domain framework, you should treat them as a balancing act. If you find things tilting toward a heavy generator or heavy generated code, push back by transferring some of the burden to the domain framework. If you find the domain framework becoming unwieldy or in danger of turning into an end in itself, apply pressure in the other direction by simply focusing on getting working code out. If you find the generated code becoming obscure and unrecognizable to developers, and at least at this stage there may be a need for modelers to look at and debug the generated code, take a step backward toward generating the same kind of code as good developers previously wrote by hand.

Remember that you can always refactor the balance later: existing models will continue to generate code that works fine, even if you move blocks between the generator and the framework.

As we saw in Section 11.3, generators can be used for things other than code or other product source files such as XML. In most cases, generators for things such as model checking or documentation are not essential at this stage, and can be left to bubble away on the back burner during the pilot project. An autobuild generator, as discussed in 11.4.1, may however be an important element even in the pilot project.

13.5 PILOT PROJECT

While you will have drawn sketches of models while creating the modeling language, and built working example models before creating the generators, there is nothing like a real life project for testing out a DSM solution. In the pilot project, a separate team will be testing the prototype DSM solution to ensure that it works and is a good medium for building applications in the domain. The DSM solution team will be actively present throughout the project, helping the pilot team learn and use the DSM solution, and also fixing and improving the various parts of the solution as necessary.

You need to find a pilot team with a real product or piece of the product to build. As we mentioned above, the target should be something fairly standard in terms of the domain, and the project should not be on the critical path. The team itself should be motivated and disciplined.

The modeling language will have remained relatively stable while building the generators, but now it is time to open it up to change again. Of course, it will break your heart to change what in all likelihood will have become your “baby” by now. In addition, of course, there is a higher cost associated with changes at this point, since you will need to update not only the modeling language but also the generators and existing models. However, this heartache and cost are nothing compared to the heartache you will cause modelers if you do not make the changes, and the cost to the business in terms of lost productivity.

All too often at this stage we have seen people make mistakes that proved fatal to the whole introduction of DSM. Most often these mistakes are some kind of shortcut or ducking out in terms of time, resources, or responsibility. In a way, this is perfectly natural: defining the DSM solution was a challenging learning process, but in a fairly safe environment. The only people involved on a day-to-day basis were the team themselves, and the activity was largely technical. Now a new group of normal users are going to see and use what the team has built. This will certainly result in criticism, constructive or otherwise.

This is also the first phase where there is the start of the organizational changes that DSM entails, and many technical people are uninterested in doing the people-related work to make changes like that run smoothly. If management are skeptical or uncommitted, they may well have temporarily pulled key resources off the team during the earlier phases to fight fires elsewhere. The project will thus have taken longer, increasing the pressure to try to cut corners. This is not the stage to take your best people off the DSM team. You will need all your skills and resources to climb this next learning curve and become adept at managing the evolution of a DSM solution while it is in active use. This, after all, is the task of the DSM team in the future, and it is a lesson best learned now while things are still on a small scale.

At the start of the pilot project, you will need to provide the pilot team with training and documentation of the DSM solution, as well as the tool itself. One good training method is to walk through an existing example model, explaining what it means, and then generate and show the running application. You can then start with an empty model and rebuild the same example from scratch, so the team learn the tool and also get a feel for how long things take. If the team is used to coding in the domain, you can also show them the generated code, but remind them that the idea is to move above that level to think in terms of the domain rather than the code.

As the pilot team use the DSM solution, you will find bugs that need fixing straightaway, areas that need improvement, and things that could be better. It is by far the best if these changes can be made quickly, with minimal interruption to the work of the pilot team. As with later DSM use, this is an area where you will see real benefits to having a DSM environment that makes such evolution possible and easy. Useful features include being able to metamodel and model in the same environment, as opposed to different environments, and automatic updates of existing models as the metamodel changes.

While versioning in general has a smaller role in DSM than in code-based development (see Section 15.3), it is at its most important during the pilot project. At no other stage will the parts of the DSM solution be evolving so rapidly while there

are models that rely on them. You must know at all stages which version of the metamodel the models correspond to, and which versions of the metamodel, generators, and domain framework go together. In each of these elements, there are two kinds of changes: changes that require a change in other elements, and changes that do not (e.g., simply changing symbols or fixing a bug in a domain framework function). Use major and minor version numbers to separate the two kinds of change, and have elements' version comments refer to the versions of the other elements they depend on. In particular, the generator will depend on both the metamodel and the domain framework. As always, version comments can also provide a vital record of the design rationale behind decisions.

At the end of the pilot project, you will have your first working application built with DSM. You will also have a fully working and proven DSM solution, and a fair idea of what kind of use process works well with it. The pilot project also has a major role to play in the process of organizational change. It acts as both a marketing vehicle, showing management and other developers that your DSM solution works, and as an educational program, producing the first small crop of users who are familiar with building applications using DSM.

It is thus important to wrap up the pilot project properly. If there were any changes to the modeling language that you had to postpone, make them now. You should also apply these changes to the pilot project models, both for testing and also since you want them to be at the same version level as the models, both built during the next phase of wider deployment. Write up and present the experiences of the pilot team as a project retrospective, and also record the experiences of the DSM team: what kinds of evolution worked well, and what kinds gave problems for either the tools or their users.

While the special nature of the pilot project makes it less productive than later use, you can still gather some statistics on the time taken to create the product compared to code-based development. Although these figures will not be accurate or statistically significant, management is unlikely to fund the 25 pilot projects that would be necessary for statistical significance! A clear increase in productivity in the pilot project, accompanied by an analysis of problems faced and suggested solutions, should be enough for most pragmatic managers to give the go-ahead for wider deployment of DSM.

13.6 DSM DEPLOYMENT

After the pilot project, the structure of the modeling language and the behavior of the generators will be in good shape. Indeed, in these aspects the DSM solution should be able to make a good claim at production quality. There are, however, other aspects to the solution that are not yet at the same level of quality: it has not made sense to work on these until the core of the solution is stable. In this section, we will look at these areas and at what extra collateral is necessary to turn the DSM solution into what Geoffrey Moore (1999) would call a “whole product.” First though we will take a brief detour to look at the issues of organizational change involved in DSM adoption.

13.6.1 Learning about Organizational Change

For all our love of trying out new technology, we developers are not exactly renowned for our ability to cope with change, particular if it feels it is by imposition rather than choice. Fortunately, there exists good work elsewhere on the topic of introducing change, and that can be applied equally successfully to DSM introduction.

When thinking about deploying your DSM solution, take the time to sit down with two people. First, you need someone from your IT department, who has been responsible for rolling out new technology. If you look how much time and energy goes into something as simple in theory as updating Windows, you should get an understanding that deploying DSM is more than just putting a tool license on everyone's desk and the metamodel on a file server. Of course, Windows is an unfair example: most of the work there is checking all possible applications for compatibility with the new version. This is not a problem in DSM: the results of DSM—the source code—will look the same as previous source code and run on exactly the same platform. Instead, the change is more like moving from writing documents in LaTeX to writing them in a word processor, or from writing HTML in a text editor to creating it with FrontPage. You are moving from a textual language to a new tool with its own graphical user interface “language.” The IT department will understand the issues of tool deployment and training related to the new tool, language, and process, and you can learn a lot from them.

Second, talk to an old-timer from your development staff: someone who has lived through a major change in development tools and languages, for example, the move from assembler to third generation languages (if you are lucky!), or the move to object-oriented languages, or at least the introduction of UML tools. Only the first of these really corresponds well to introducing DSM, but you take what you can get. In addition to the external view of change introduction given by the IT department, this will give you the internal view of how the change actually took place in the minds—and hearts—of the developers.

Rather than turning aside from the topic of this book to the general psychology and techniques of change introduction, we will simply point you to the excellent set of patterns by Mary Lynn Manns and Linda Rising. They are now available as a book, *Fearless Change: Patterns for Introducing New Ideas*, and earlier versions are available online, for example, <http://www.cs.unca.edu/~manns/PC.DOC>.

13.6.2 Polishing the DSM Solution

The concepts and main rules of your modeling language are ready, and the generators produce good working code that runs on top of the nascent domain framework. Practically speaking everything works, and if you took it into production now, you would see the large productivity increases you expect from DSM. There does, however, remain some significant work to do. This work will not noticeably improve the productivity of the solution in use, but it will make a big difference to how well your DSM solution is accepted. Without that acceptance, getting the DSM solution into use may take much longer or even fail to happen.

Fancy symbols are, of course, completely unimportant to productivity. Provided the symbols let you easily distinguish the concepts, and present the main properties of their concepts in an easily readable format, you already have a good visual language. We developers, however, are finicky creatures, and the human race as a whole tends to judge by first impressions. A sloppy, ugly set of symbols will have a large impact on how your language is perceived. Even a perfectly decent set of symbols may look rather simplistic to eyes used to Vista or OS X. With access to the right tools and skills, you can turn a good language into a great one. For many of us, this is probably an area best left to (semi)professionals—providing that you can communicate to them the real needs of the situation and that in this case function must always come before form.

The symbols represent the static part of the language, how the users will see the models. On top of this is the issue of how the users will create, edit, and navigate around the models. This user interface will be highly reliant on the tool support facilities provided by the DSM framework or environment you use. We will look at this in more detail in the next chapter, but for now the main issue is that if there are things you can improve in the UI, now is the time to do that.

Next you should look to the other visible parts of the DSM solution. Check that the generated code and domain framework follow in-house standards for coding style, formatting, and so on. Although developers should not have to look at these, they will certainly be turning a sharp eye on them at this stage as they form their opinions about the DSM solution. The other side of the coin is process automation: making an autobuild script or generator so that developers can go straight from models to running the finished product, without having to look at the code or run commands on it.

If the preceding topics were about making things aesthetically pleasing and making visible “seams” in the process invisible, we now want to consider making things visible that are currently hard to see or invisible. For instance, there will be model structures that are not acceptable to the generator, but which the modeling language allows you to create. During earlier stages, we have advised you against adding too many rules to the modeling language. Now is the time to make sure that the necessary rules are in place. You are not trying to stop the developer from doing things, more to prevent her from making models that the code generator does not support. There will still be a large number of models that can be made which make little sense. Trying to prevent all of these is impossible and also undesirable. Some of them will turn out to be perfectly good ideas: creative ways of using the language that had not occurred to you yet.

You should also make sure that your rules do not make the language hard to use. There may be certain states of a model that are illegal, but would occur frequently on the shortest path for editing a model from one legal state to another. It is better to implement such rules as checking reports, which can be run by the user on demand or automatically when generating code.

If your organization has a strong requirement or tradition of documentation in Word files or similar, you will probably need a way to generate the models, their elements, and properties as a document. Good tools will offer you this already,

but even then you may want to edit the styles and layout to match existing standards. If the DSM environment provider has not done this work for you, and you need Word files rather than plain text or HTML, be aware that this can be a significant amount of work.

Finally, now is a good time to complete any refactoring on the generator definitions. This will obviously help you later in maintaining the generators, but will also make it easier to evolve the modeling language.

It will hopefully go without saying that you should also have whatever documentation is appropriate for the internals of the DSM solution itself. The project may be your baby now, but what about when you get that big promotion on the basis of your results here? Spare a thought for the poor soul who is given the task of taking over the DSM solution—and for yourself, if the project crashes and burns behind you. Do you really think you can spin it so it looks like you were 100% a hero and the new guy 100% a villain?

13.6.3 Introduction and Evolution Process

One of the more frustrating things for developers is the sheer amount of work that still remains to do once the product is “ready.” Polishing the product is one part, but even after this there is still the installer, download or other distribution system, user documentation, and training material. Introducing the DSM solution to your organization follows this pattern, with a few minor differences.

Because of the importance of evolution in DSM, we want to plan the distribution mechanism for the DSM solution before we finalize the user documentation. At the same time, we can also cover the initial distribution and introduction process. These need to be planned and tested well: while it will be a pain to have to install and uninstall the various components on a few machines, this pain will be nothing compared to the alternative. Imagine what would happen if you tried to push DSM out to all developers with a half-baked distribution system with poor instructions.

We have divided the DSM solution into several components in this book: the modeling language, generators, framework, and tool support. These components may not, however, match the actual files or other elements you need to install for users. For instance, in some DSM environments the language and generators may normally be delivered as a single file, whereas in others they may be delivered in separate files—or even a file for each diagram type and each generator.

In less mature environments, there may not be a separation between the generic modeling tool support, which is the same for all organizations using the tool, and the domain-specific language and generators of each company. Instead, the language and tool support, and possibly the generators, will be lumped together in one large executable or bytecode package. This is not such a problem for the initial introduction, but it makes evolution a harder process. In particular, looking at older versions of models in this situation is problematic: you will need to install the complete, correct old version of the tool. If the tool does not allow multiple versions to be installed simultaneously, this is something of a disaster. The situation becomes even worse if the tool is tightly coupled with your IDE or operating system version:

reinstalling the required compatible version of those is unlikely to be anyone's favorite pastime.

Whatever the situation for your particular case, you must identify the components you need to distribute to developers initially and when upgrading. For upgrades, consider which parts of the DSM solution may change, and whether you want to allow an upgrade just for that part or for a set of parts together. Use your experience from upgrades during the pilot project to come up with a scheme that makes sense for your situation. Remember that at the start the rate of evolution will still be fairly high, although not at the extremes seen in the pilot project: the DSM solution is now more mature, and the larger number of developers adds some inertia.

As you build a mechanism and process for distributing updates to developers, do not forget the importance of a communication channel in the other direction. Developers need a way to submit feedback about the DSM solution. There will probably still be minor bugs in the domain framework, generated code, and generators, since these are the most code-like parts of the solution. If the tool support includes handwritten code, or is produced by a DSM framework that is still under development, it may well contain outright bugs—in addition to the certainty in all tools that individual users may wish some things had been done differently. Finally, the modeling language itself will have areas for improvement and extension, both at the start and later as the domain evolves. Developers need to be able to report all of these, and ideally be informed when the problem is corrected.

At this early stage before the initial deployment of the DSM solution, it is of course impossible to create the perfect process for evolution. What is important now is that there is a tested and documented process that works technically: all developers can easily upgrade to a new version of the DSM solution, and all models update sensibly to work with the new version. This process itself will evolve over time as you gather more experience of what works well with your tools, language, and organization.

13.6.4 Learning Material and Training

The developers who will use the DSM solution are faced with a significant learning curve. The task is made easier since the domain will be familiar, and the modeling language concepts will normally map directly to the domain. On the other hand, the developers will probably not be accustomed to using a graphical modeling language to actually build products, as opposed to simply designing or documenting them. Part of the function of the learning material and training will thus be to help ease this shift in mindset.

Clearly, the most important element to teach developers about is the DSM language. Good DSM environments will allow you to document the individual language concepts inside the tool itself. This will serve as a good online reference to the modeling language, as well as providing context-sensitive help about each of the concepts. Additionally, it is often useful to have the language described in a more familiar format, either on paper or in some electronic document format.

Since the developers will not be metamodelers, it is probably not so useful to provide an actual metamodel of the modeling language. Instead, try to build an

example model or models that illustrate all the concepts and the way they are used. You can then describe the model and the way it works in the text. This tends to be more useful than a list of the concepts and their descriptions. Such a list can be added at the end as a reference, although keeping this documentation in the tool itself puts it closer to the developers and also reduces the risk that it will become out of date.

For the tool itself, first look what the tool provider offers in terms of user documentation. All but the most primitive DSM environments and frameworks should have abstracted tool behavior from issues relating to individual modeling languages, so the tool's functions and behavior can be described once for all modeling languages. In addition, it is also useful if you can provide some initial documentation that describes things in terms of your modeling language. This can help make things more concrete and reduce the danger of new users feeling unsure about which metalevel they are on. Something that has worked well in our experience is to provide step-by-step instructions to building a simple model, just a little more than "Hello world." Outlining a second task, without the step-by-step instructions, is a good way to nudge users onward from rote repetition to actually learning and understanding how to do things themselves.

Simply providing people with a tool, a language, and documentation goes a long way to helping most of them become productive in the new environment. Different people, however, learn in different ways, and for some the best way is a more formal training session. A disproportionately large number of software developers learn best by reading, but DSM also opens the doors to people who have not traditionally been perceived as developers. A good example of this was seen in the insurance domain experts from Chapter 6, many of whom had never programmed before. Rather than studying alone by reading, these people may prefer to learn by hearing, by seeing, or by doing, often as part of a group. Group training also has benefits for all kinds of learners: increased out-of-band communication, a sense of group identity, and learning to learn from one another and help one another.

Initially, any training will probably be carried out by the DSM solution team. It can also be a valuable process for them, as they see first hand any problems that new users have in getting to grips with the language and its tool support. Since there is no reason to expect the DSM solution team to be particularly gifted or comfortable as teachers, it may later be kindest to all to let them pass the baton to people with more experience in this role. While the odd fine piece of detail may be lost in this transition, overall it is the clear communication of the basics that is more important. More esoteric information and tricks of the trade can be passed on later in various creative ways, for example, by a "Tip of the Week" email.

13.7 DSM AS A CONTINUOUS PROCESS IN THE REAL WORLD

By now it should be abundantly clear that DSM is no silver bullet—nothing is—and certainly not a one-shot, fire-and-forget weapon. A large part of its power comes from its ability to evolve, and this in turn stems from its being close to the users

and needing to support only a bounded domain and set of users. As the domain evolves and the group of users grows, you will face again the questions of where to draw the boundary, what to include, and what to leave out. You will also face the question of evolution in your organization: fitting the needs of the organization with the desires of the people who make up that organization. Finally, there is the question of the fate of an individual DSM project: what happens at the far end of the life cycle.

13.7.1 Evolutionary Forces

Evolution in human organizations works best when it can find the right balance between change and continuity. At one extreme, the DSM solution could change so fast that nobody could keep up, or split into a myriad of languages, each for too narrow a group. At the other extreme, the DSM solution could stagnate, either not being maintained or because users do not update to newer versions. Most of the dangers here are obvious, but we will pick out a few to look at.

Here, we have mostly assumed a single DSM solution, a single group of users, and a single version of the DSM solution in use at a time. In reality, there may be pressure to have different variants of the DSM solution for different groups of users. This pressure mostly can and should be resisted if the variations are small. If a group has a clearly different domain, this may require either a new modeling language in the same DSM solution, or a new DSM solution entirely.

As individual projects move through their own life cycles, there will also be phases when they are unwilling to adopt new versions of the DSM solution, for fear of introducing unexpected changes or bugs. To some extent this is justified, although the risks must be re-evaluated compared to code-based development. Clearly, nobody would want significant changes to many pieces of code shortly before a release, and in a way a change to the generator produces that. However, the real danger is not in the number of changed lines of code, but in the number of places where a person might make a mistake. In the code-based world, this would be directly proportional to the number of changed lines. With DSM, the level of risk is measured largely by the number of lines changed in the generator. If changing that number of lines would be acceptable in manual coding at the current stage of the project, the generator change should also be acceptable.

Another possible comparison would be with upgrading to a new version of a compiler at a late stage of a project. This provides a better match with DSM upgrades, but there are still some important differences. The compiler is made by an outside agency, and any bugs found in this project will normally take far too long to be corrected. Since the DSM solution team is in-house, any bugs found in the new version can be corrected quickly. Also, the compiler is made by people unfamiliar with the domain and the code of this organization. The DSM solution is, of course, made by people at the opposite end of the scale, experts in both the domain and the code. The chance that the compiler team will introduce an error that affects this organization, because they were not aware of or did not consider its needs, is much higher than the corresponding risk for the DSM team.

13.7.2 The DSM Solution Team

An important source of continuity in DSM evolution is the DSM solution team. As far as possible, try to keep the team together. Since the team must be staffed by your top experienced developers, there are few enough people qualified to be part of it. The people who have served on it for the first round will probably also be your only people with experience of creating a DSM solution. Of course, if it has become clear that this was not the right place for a certain person, common sense must be applied. A good place to look for a possible replacement is the pilot project team. Not only have they had more experience with DSM and your solution than others, but also they have been more exposed to the internals of the solution and the process involved in creating it.

The size of the DSM solution team will largely be dependent on the DSM environment or framework you use. A high-level DSM environment will keep the amount of work low enough that the evolution of the DSM solution will require less than one person's full-time work, even for a large language. Lower-level frameworks will require a few full-time staff, possibly even several people. In addition to these, you may need one or more people working on the domain framework: whether these are considered part of the DSM solution team or your general in-house components team is perhaps a matter of taste.

If your organization has more than one domain for which DSM seems applicable, there is also the question of the next DSM project. Certainly the first DSM solution team will have plenty of experience that can be used beneficially on the next project. How best to use it is however dependent on the particular situation: how much the domains are related, what the people in the first DSM team want, how mature the first DSM solution is, and what kinds of people are available in the new domain.

In some ways, an external DSM consultant may still be the best solution for the second project. The leader of the first project has one success under her belt, but this may not yet mean she is best suited to teach others. If there are other applicable domains after the second one, and someone from the first team is keen to become the in-house expert, perhaps having her work alongside an external DSM expert on the second project is the best solution. This should help avoid the danger of falsely abstracting as a general principle something that was a particular feature of the first solution, while at the same time setting you up to become more self-sufficient in the future.

13.7.3 The Rise and Fall of a Domain

In our experience, as long as a given domain remains useful to an organization, and the DSM solution for it is allowed to evolve, there is no built-in aging process in DSM. On a technical level, it appears that DSM languages actually have a longer life span than generic modeling languages. Among our own customers, there are hundreds of people actively using DSM languages that predate UML, at a time when UML is clearly heading toward its sunset.

Perhaps the key difference is that a generic modeling language in use is only really able to grow, and eventually it either stagnates or collapses under its own weight. The

latter fate is perhaps more common for languages with an evangelistic bent, as they try to increase their range of applicability to an ever wider and more diffuse group of users. A DSM language can grow, but it can also evolve in other ways. If a change needs to be made, it can be: there may be some pain associated with it, but the gain to the organization can be seen to be greater. With a generic language, such a change will often be left undone through fear—and well-grounded fear at that. If the language changes, will the third-party tool vendors update their tools to support the new language, and provide a way to update models to match the new language? The tool vendors must face a similar fear: if we update the tool, will users choose to stay with the old version?

While a DSM solution will remain viable in its domain, even as that domain evolves, there is also the question of the viability of the domain itself. Businesses change, and it may be that at some point customers are no longer asking for products in that domain. Companies must also use their limited resources to best effect, and sometimes this may mean even a viable domain is shut down, and the developers are moved to a different project that offers a better return on investment or promise of growth. Unfortunately, we must also take into account the possibility that a new manager or owner may feel the need to assert themselves by making radical changes—even where those changes are not merited by the situation. Sometimes this means a wholesale change to different tools, languages, or processes, and that may mark the end of a good DSM project. Given the all too common alternative way of asserting authority—firing large numbers of people—this is perhaps the lesser of two evils.

In all of these cases, our suggestion would be to make a good end of it. If the change is inevitable, there is nothing sacred about a particular DSM solution. Fight bad ideas by all means, and make sure those in charge understand the productivity benefits of DSM, but look at the situation with a healthy dose of realism. If the use of this DSM solution is going to stop, it is time to wrap up the loose ends. Although there will obviously be limited resources for this task, make the most of what you have.

Make sure that the final versions of the DSM solution components are recorded and that the components themselves are stored. The models themselves are assumed to be under good version control and so should present no extra problems. If your tool support is based on a framework that is tightly integrated with an IDE, or for other reasons may later be challenging to reconstruct, it may be a good idea to make a disk image or a virtual machine containing a working environment. Should it later be necessary to resurrect the DSM environment, for example, to allow some final bug fixes to a released product, this will allow those changes to be made much faster and with more certainty that no unintended differences are introduced.

Although most of the history of the DSM solution development will be under version control, there may well be extra information that is only stored in the email folders or on the hard disks of the DSM solution team. A quick trawl through the relevant locations should turn up some useful gems to record for posterity.

In many ways, closing down a DSM project is more like putting it in mothballs than killing it off. With a traditional hand-coded project, changes in libraries, platforms, and even programming languages will quickly render the old code useless. With DSM models, the modeling language and models would remain valid even in the face of

such major changes: new generator and domain framework are all that would be needed. The same forces that resulted in the domain being retired could just as easily result in its resurrection. While a full-scale revival may be unlikely, the ability to quickly churn out a couple more products in the old range next year may be a useful option from a business point of view. In particular if a new manager's demand to revert to a "standard" programming and generic modeling approach is inexplicably taking longer than expected to produce results!

13.8 SUMMARY

This chapter has seen your DSM solution evolve from an initial idea to a full solution in production, reflected throughout in experience of its use. The evolution of the DSM solution has been a major factor, but the evolution of your organization has been even more important. The developers have learned to use the DSM solution to good effect, shifting their thinking to a higher level of abstraction, closer to the requirements and to the end users. The DSM solution team have had their first experience of creating, introducing, and maintaining their own domain-specific modeling language and generators, taking control possibly for the first time of their own development process, tools, and languages. No longer will the offerings of third-party vendors and outside organizations seem like the only possible "real" choices, with in-house solutions seen as little hacks. There will also probably be a change in attitude toward such third-party tools and solutions: any unrealistic feelings of their unassailable authority and superiority will have evaporated; any unrealistic demands for perfect solutions will have been replaced with some measure of respect for the trade offs those vendors are forced to make.

There will also be an increased maturity in the overall issue of introducing new technology to your organization. In some ways, the first DSM project will be the major change; in other ways, each subsequent DSM project will be its own change process. While such processes are not at the heart of what most top developers love about their work, most would acknowledge that such "soft," people-related issues are a major factor affecting the success of any project, even when the "hard" technology side is in great shape.

Most importantly, running through the process suggested above even once will have given you some insight into how to adapt its necessarily generic advice to the specific situation of your organization. Even on your first run through you will have tailored some parts and made your own adjustments. If the first project was successful, in subsequent domains you should find you have less need to market and prove the approach to management before getting the green light.

Should you go on to apply DSM in several domains within your organization, and it becomes a standard tool in your palette of approaches, tools, and languages, you will have entered a select band of a very few organizations in the world. If after a few DSM projects you have found changes or additions to the above process that seem to work in general, please do share your feedback with the authors or the wider DSM community.

CHAPTER 14

TOOLS FOR DSM

In this chapter we will look at how to obtain tool support for your Domain-Specific Modeling (DSM) solution. By analogy with the evolution of other kinds of software, Section 14.1 will show the steps on the path from laborious hand coding of tools to simple configuration in integrated environments. Section 14.2 will take us on a whistle stop tour of the history of DSM environments, the current best evidence against the theory of evolution. In Section 14.3, which forms the body of this chapter, we will look in detail at the features that make or break a DSM environment. Section 14.4 will briefly look at the prominent tools available today.

14.1 DIFFERENT APPROACHES TO BUILDING TOOL SUPPORT

There are a number of ways you can build tool support for a new modeling language. By putting an ordering on them, we can consider them as levels as follows:

- (1) Write your own modeling tool from scratch,
- (2) Write your own modeling tool based on frameworks,
- (3) Metamodel, generate a modeling tool skeleton over a framework, add code,
- (4) Metamodel, generate the full modeling tool over a framework,
- (5) Metamodel, output configuration data for a generic modeling tool,
- (6) Integrated metamodeling and modeling environment.

This is indeed a common pattern for the evolution of many kinds of systems. If we take an analogy with the output being a typeset document rather than a modeling tool, we can look at the history of word processing. While the match is not perfect, we can equate the modeling language to the formatting styles or template, the model to the document, the modeling tool to the editing software, and the generator to the printing software.

For word processing, levels 1 and 2 are lost in the mists of the past: it is a long time since anybody had to hand code their own editor or printer driver. PostScript, TeX, and LaTeX roughly correspond to the scale from levels 3 to 5. Only at level 6 with tools like Microsoft Word did it become generally possible to make on-the-fly changes to the styles, and have existing “instances” of those styles update instantly. Nowadays, virtually everyone is at level 6, with a few diehards proving their skills on level 5 or 4. Organizations have recognized the benefits and savings and created in-house templates for employees to use. Word processing is thus a mature technology, and mature tools have also been widely adopted.

We could also make an analogy with web applications: initially everybody had to write their own, then frameworks became available. The frameworks developed and moved from generating code to reading configuration files at start-up. Only with the advent of systems such as Seaside or Dabble, and to a lesser extent Zope or SharePoint, has it been commonly possible to create and test a web application in the same browser window. In this domain there may be good tools, perhaps even some that are both mature and good, but adoption has not yet progressed so far: many are still working with technology from the lower levels.

Regardless of the field, we would expect later levels to be better than earlier ones. Poor execution can however mar this: there are good and bad frameworks, and both kinds can be used well or poorly. All things being equal, a user who simply wants to get a job done will generally be best served by the more mature levels. Those who like to play with new technology for its own sake will be happy on all levels; those with a strong adherence to a particular vendor, language, or environment will take whatever is on offer there; those with a need for control will take the highest level where they still feel they have that control if they need it. Aside from personal predilection and “not invented here” prejudices, there can also be valid reasons for using a lower-level solution even when good higher-level solutions are available.

A key ingredient here is experience: for those creating their first word processing documents or stylesheets, Word is a better choice than TeX or PostScript. In many areas where the user finds Word does not support what he would like to try, the reason is more likely to be that it was a poor idea than that Word is somehow lacking. For instance, the user may find that the letter “i” looks too thin on his screen, and want to make it wider everywhere. This would be possible in PostScript, but Word offers no specific support for such an operation. The reasons are clear: fonts are generally designed so that character sizes are well balanced, and also any apparent imbalance is more likely just an artifact of the screen resolution. By not providing global per-character settings, Word embodies this piece of experience; the apparent freedom of PostScript would be a bad thing for most users. However, an experienced typographer may well want to experiment with different kernings, and then the freedom of

PostScript or TeX may be needed—at least until she finds an environment specifically built for typographers.

14.2 A BRIEF HISTORY OF TOOLS

Those who cannot remember the past are condemned to repeat it.

- George Santayana,

The Life of Reason (1905)

The history of tools for implementing support for new modeling languages has not been a pretty one. Unlike other software fields, such as operating system UIs, web browsers, compression or encryption, there has been little building on previous work. There seems to be no rational explanation for this; perhaps the main factor is to be found not from the field or tools themselves, but the mindset of the people who make them. Anyone who tries to define how others should build systems must be something of a control freak. Anyone who tries to define how such control freaks should define their modeling languages must be even more, ah, sure of themselves.

Looking back over the various tools clearly shows that later tools are by no means better than their earlier counterparts. Although there have been slight differences in emphasis, all have been attempting to solve the same problem. It thus seems useful to offer a potted history of the tools, their emphases, and how they fared. Particularly interesting in the context of DSM is whether the tools could be used by someone other than their creators.

14.2.1 1970s and 1980s

In the 1970s and 1980s, the search for a solution to the “software crisis” (Brooks, 1982) was focused in three directions:

- *analysis* of the whole process and concepts involved in building information systems (e.g., Chen, 1976; Brooks, 1982),
- application of more rigor in the early stages of projects by following explicit *methods*—modeling languages and associated processes (e.g., Gane and Sarson, 1979),
- *documentation* of the development of systems, often using computers—an idea going back as far as the early 1970s (Bubenko et al., 1971).

The branch of methods grew with astonishing rapidity, largely subsuming the other two branches, and producing huge numbers of methods of increasing complexity. Aside from leaving practitioners stranded in a “methodology jungle” (Avison and Fitzgerald, 1988), and for a long while forcing academics to limit their research to classifying the methods (Olle et al., 1982), rather than examining how they performed in practice, the growth of methods easily outstripped that of the computer tools that were built to help implement them.

The very early, text-based Computer-Aided Software Engineering (CASE) tools such as PDL (Caine and Gordon, 1975), PSL/PSA (Teichroew and Hershey, 1977), SEM (Teichroew et al., 1980), and SREM (Alford, 1977) had allowed changes to the modeling language supported, which gave users some possibility to maintain tool support in the face of rapid language evolution. However, newer languages and tools had adopted graphical representations and interfaces. While these were substantially easier to use, they were more complicated to specify, and thus CASE tools were no longer able to provide the user with facilities for changing the modeling language they supported. The CASE tools, heavily outnumbered by modeling languages, were thus forcing the users to adopt their built-in modeling language and process, rather than supporting the languages and processes from which the organization was already starting to see benefits. Conversely, the organization could continue with its own modeling language, substantially weakened by the lack of computer support, or even build its own CASE tool, a heavy investment in a venture in which it had no experience, and could often only make financially feasible by attempting to sell the resulting tool to others.

These conflicts, of course, did nothing to improve the image or practical benefit of the expected CASE revolution (Yourdon, 1986). CASE had been unrealistically trumpeted to be the “silver bullet” that would solve all software development problems. All too often, the response to the failure of CASE to provide such a solution was to blame the modeling language or the tool. This of course led to the development of yet more languages and tools: hardly likely to improve the situation.

14.2.2 1990s Overview

As we have observed, building a whole modeling tool from scratch is a large and complicated project, significantly too slow to keep pace with modeling language development. The solution to this conundrum thus lies in a tool that can be customized to support any modeling language. Two possible approaches to this were perceivable in 1990: effectively our levels 2 and 5 above. A modeling tool could be designed and built modularly, so that the minimum coding effort was required to change the part concerned with a particular modeling language. Alternatively, a tool could have the modeling language itself as data, rather than as code, and functionality could be provided for altering this data, in the same way as was done in the early text-based tools. A weakened form of the latter was a hybrid of the two: the modeling language was expressed in data or a mixture of data-like and code-like parts, which were transformed into code and compiled and linked with the generic modules.

The former solution was the one largely adopted by industry, in turning out new versions of an existing code base for new modeling languages. The approach however had flaws from the users’ point of view: only the vendor could make the changes, and the cost of such changes was high. While the reduction in work to make a modeling tool for a new modeling language was significant (one manufacturer claimed reuse as high as 90% (Rust, 1994)), the rate of such adaptation still proved insufficient to satisfy users’ needs. Furthermore, the cycle from requesting a change to using the modified tool was painfully long, and the customer was left highly dependent on the vendor.

The latter solution, called CASE shells (Bubenko, 1988) or metaCASE tools (Alderson, 1991), produced promising research prototypes and a few somewhat limited commercial products. These early tools were not widely taken into use, although the use of Systematica's Virtual Software Factory metaCASE tool (Pocock, 1991) by IBM in building its BSDM support tool (Haine, 1992), and again by Heym and Österle in the construction of their MEET method engineering environment (Heym and Österle, 1993), provided practical proof that such tools could be useful. With some success in the Francophone world, GraphTalk (Jeulin, 2005) offered good pointers to future directions, although initially it only ran on special LISP machine hardware. Comparisons of these metaCASE tools (e.g., Marttiin et al., 1993; Goldkuhl and Cronholm, 1993) revealed that the process of metamodeling (Teichroew and Hershey, 1977; Brinkkemper, 1990)—configuring the tools to support a new modeling language—could be improved, as could the accuracy and breadth of the support for the modeling language in the configured tool.

MetaCASE tools of the 1990s tended to separate the metamodeling part from the modeling part. The description of the modeling language was written in a textual language, possibly compiled to some other form, and then fed into the configurable modeling tool, which then supported the new modeling language. Some more advanced tools provided partial graphical support for the metamodeling process, for example, MetaEdit and ToolBuilder. In tools of this type the graphical metamodel was made with the same modeling tool functionality as the resulting tool, but using a special graphical metamodeling language (whose metamodel had been bootstrapped from a textual description, and often had special links to tool functionality reserved only for that language). The resulting metamodel was first transformed into a textual language, which was then compiled as above.

This separation of metamodeling and modeling had an important drawback: it was not possible to interactively test the results of metamodeling immediately, because of the long transform–compile–link–run cycle to move from metamodeling to modeling. This was a significant hindrance to the metamodeling process, as if a problem was spotted while testing a modeling tool, the user had to exit the tool, restart the metaCASE tool, read in the metamodel, edit it, transform, compile, and link it to form the resulting modeling tool. These steps often required separate commands, manual text editing and external tools. The resulting modeling tool could then be restarted, and the existing models generally needed to be explicitly updated to use the new metamodel before the change could be tested.

Below we shall take a look at three tools of the 1990s. Others of course existed, and some tools born in the 1990s are still going strong (we will see current tools later in the chapter). The tools below were selected on the basis of their use in real-world industrial projects, their coverage of both metamodeling and modeling, and the availability of reliable evidence.

14.2.3 DOME (Honeywell)

The Domain Modeling Environment (DOME) was created by Honeywell Labs for use in its own research and implementation projects. It began as a project to build a

Petri-Net editor but by 1992, after a few similar editors, it was apparent that it should be possible to generate such editors from a tool specification. This generation approach was called MetaDOME. The subsequent development of ProtoDOME and CyberDOME moved away from generation to interpreting the tool specification directly. This ability to store and edit metamodels directly as data meant that updates to the metamodels could be made while models were open.

DOME used a graphical metamodeling language called DSTL, which covered the main concepts and also some constraints and graphical behavior. The main concepts were Graph, Node, Port and Connection, but there were also specialized concepts related to Components and Interfaces, plus a special type, Archetype, for reusable objects. Extra behavior could be added in the LISP-like Alter language. Alter could also be used for transforming models, for example into code, but there was also a separate MetaScribe component for simpler generation. MetaScribe used a specification written in Word, apparently as a template-based generator, plus a separate output formatter to transform the output to a specific document format such as plain text or FrameMaker. DOME is described further in a number of articles (e.g., Engstrom and Krueger, 2000).

DOME was later released as open source, although the Smalltalk code has been static since 2000. Relying only on older Smalltalk versions meant the user interface was rather Spartan and nonstandard. The use of files for models was seen as a limiting factor for the adoption of DOME by larger teams: only a single user at a time could edit a file, and there was poor support for breaking a model into separately editable chunks. To address some of these limitations, a rearchitecture and reimplementations in Java has been underway since 2003.

14.2.4 MetaEdit (Jyväskylä University/MetaCase)

MetaEdit (Smolander et al., 1991) was the predecessor of MetaEdit+. It consisted of a generic modeling tool whose modeling language support was provided in binary metamodel files. Its metamodeling language was based on OPRR (Smolander, 1991), exhibiting a rare instance of reuse of existing work: OPRR was originally made by Welke (Welke, 1988), and used in the QuickSpec tool (Meta Systems, 1989).

While metamodels could be built purely textually, MetaEdit included a graphical modeling language containing the OPRR concepts, with which users could graphically define their own modeling languages. The symbols for the new modeling language were defined in a separate graphical symbol editor. The editors could generate these two parts of the metamodel in the textual metamodeling language, and the user would join these parts by hand to form one file. This file could then be run through the Moffer metamodel compiler to generate the binary metamodel file that drove the generic modeling tool part. If a metamodel was changed, existing models based on it could be explicitly updated to use the new metamodel: the actual update was performed automatically. The ease and incremental nature of metamodeling were recognized in the tool comparisons as the main advantages of MetaEdit.

MetaEdit had several limitations, mostly in its modeling tool functionality. Each model file was limited to a few tens of diagrams, all of the same type, and there was no

linking between model files. This prevented proper implementation of most modeling languages containing several integrated diagram types. An odd implementation of relationship metamodeling required creating several near-duplicates of relationship types in some not infrequent cases (Kelly, 1995), also complicating modeling use with extra types. The conceptual–representational distinction was rather weak: each model file was conceptually one large graph, of which subsets were visible in different diagrams. This ran contrary to the user perception of a model as consisting of several graphs, each represented in one diagram. There was no support for complex properties, effectively ruling out true modeling of the new object-oriented methods with their Classes containing collections of Attributes, each with its own properties. Complex object support was limited to a simple explosion construct, representing a free-form link between an object and another diagram in the same model.

Despite these limitations, MetaEdit proved somewhat successful as a metaCASE and modeling tool, with users numbering in the thousands. It was also used as a tool in other research projects, for example, Jarzabek and Ling (1996) found it useful in developing a BPR modeling language and its tool support. The ease of metamodeling seems to have been a major factor enabling others to use MetaEdit without intervention or hand holding from its creators.

14.2.5 TBK/ToolBuilder (Sunderland University/IPSYS/Lincoln)

The TBK (Tool Builders Kit) framework and ToolBuilder metaCASE system was originally reported in Alderson (1991) and was later commercialized by IPSYS. The system consisted of three components:

- The specification component—used to create the specification of the tool;
- The generation component—used to transform the specification into parameters for the generic tool;
- The run-time component—the generic modeling tool itself.

The metamodeling language was ER extended with some constraints and the ability to have attributes whose values were derived from other attributes. It allowed triggers on events applying to attributes and relationships. TBK definitions were in the form of textual files in four languages:

- Data definition language (DDL), which specified the basic objects and the attributes they had,
- Graph description language (GDL), which specified the graphical symbols, but also simple relationship and decomposition rules,
- Format description language (FDL), a low-level description of the individual windows, buttons, menus, and so on that make up the user interface;
- Text layout language (LL), describing a structured text format of user-enterable text fields interspersed with fixed text and punctuation.

DDL and GDL appear not dissimilar to the corresponding parts in MetaEdit's textual metamodels, although with a rather more involved syntax. The LL text layout language appears a useful addition, although little is mentioned about it in the documentation and articles. The need to specify tool behavior in FDL, however, seems to be a symptom of the level of abstraction being closer to that of a framework than a true metaCASE tool: it should be possible to provide a good generic interface, or generate one based solely on the DDL and GDL input.

Toolbuilder added a graphical front end, the METHS system, that could generate parts of the information captured by these textual languages. More detailed editing was performed in the textual languages, in the EASEL language (a rather simple 3GL), and through user functions written in C. The results were partially compiled and partially interpreted by DEASEL, the generic modeling tool runtime. DEASEL provided standard modeling tool functionality and supported multiple users on a true repository.

Thus ToolBuilder appears to have provided a usable metaCASE system, but the time required to build support for a modeling language was long, even according to IPSYS's own marketing material:

Even a completely new method can take only man-months to implement, with prototypes taking man-weeks. The rapid prototyping nature of ToolBuilder means that demonstrations can be created in man-days.

This slowness was probably because of the complicated textual languages, and the separation of the modeling and metamodeling tools. Also, while it was of course a benefit that DEASEL provided basic default modeling tool behavior, the description gave the impression that these defaults were probably insufficient for most actual modeling languages, requiring coding in EASEL or C to specify tool operations.

While ToolBuilder did not succeed as a metaCASE tool, two modeling tools based on the work on ToolBuilder had more success. A precursor of TBK was used to build the HOOD toolset used on the Eurofighter project, and ToolBuilder was also used to build the Kennedy-Carter I-OOA Shlaer-Mellor toolset. The latter took 12 months to build; the number of developers involved is not revealed.

According to one of the main figures behind ToolBuilder (Alderson, 1997), a major factor leading to IPSYS's eventual business failure was a lack of in-house expertise in the popular modeling languages of the time. This led to a series of projects where IPSYS would give a customer one of its developers in a project to create support for a new modeling language. The customer would normally receive free use of the resulting tool, and IPSYS would try to sell it to others. Sadly, in no case did this work as planned: either customers were not sufficiently invested in the modeling language, or IPSYS lacked the experience necessary to market and sell a tool for what was to them an unfamiliar modeling language. The remains of IPSYS were bought by Lincoln Software in 1995, who were in turn bought by PeerLogic in 1999, who themselves were bought by Critical Path the next year. ToolBuilder disappeared along the way, although there was a project to include part of the ToolBuilder-built Engineer product into IBM's WebSphere, to provide a web front end to CICS transaction servers.

Technically, the main problem with ToolBuilder seemed to be the large resource cost of building support for a new modeling language, coupled with the specialist knowledge required to use ToolBuilder. There seems to have been no case where customers were able to build anything other than academic prototype modeling language without experts from IPSYS working alongside them. As Isazadeh's thesis puts it (1997):

At run-time most of the functionality of the tools is provided by C functions [hand-written by the metamodeler]. . . . In general, a conclusion in working with ToolBuilder by a long-time user [Sunderland University's Ian Ferguson (1993)] is the extreme complexity of extending the functionality of the tools beyond the default operations which, together with poor documentation, requires a lengthy learning curve.

14.3 WHAT IS NEEDED IN A DSM ENVIRONMENT

When looking at what is needed in a DSM environment, we can take some things for granted. First let us look at the minimum facilities for metamodelers:

- Specify the object and relationship types declaratively,
- Specify declaratively a list of properties for each object or relationship type, with support for at least string and Boolean property data types,
- Specify basic rules for how objects can be connected by relationships,
- Specify symbols for types, whether graphically, declaratively, or in code,
- Ability for a generator to access the models,
- From these specifications, create a basic modeling tool.

The modeling tool thus created must offer modelers at least the following facilities:

- Store and retrieve a model from disk,
- Create new instances in models by choosing a type and filling in properties,
- Link objects via relationships,
- Lay out the objects and relationships, either by dragging or automatic layout,
- Edit properties of existing objects and relationships,
- Delete objects and relationships

The history of tool development contains numerous instances of small research projects creating this kind of prototype DSM environment (or metaCASE tool, in 1990s terms). Developing such a tool to the stage where you can produce screenshots for an article is relatively simple: 1–3 man-years is enough, depending on implementation technology and previous experience.

With the recent realization of the benefits of model-driven development, and the need for its modeling languages to be domain-specific, we are currently seeing a new explosion of such tools. Most of these are again from research projects, or their

modern-day distributed cousin, open source projects, but a few are also commercial projects. The year 2006 saw the release of the largest number of new tools like this, even exceeding the metaCASE boom of the 1990s.

These prototypes and “version 1.0” tools generally support a single user, one modeling language at a time, simple metamodels focusing on objects with basic properties and relationships, and symbols with a single graphical element and a label. Generator facilities will be based on text-to-text transformations or handwriting code to read models. The resulting modeling tool will normally be missing the majority of functions that users expect from a graphical editor.

Compared to creating the initial prototype, turning it into something that could be used in the real world is a much harder task. Building such a system without significant personal experience of industrial scale DSM appears to be a near impossibility. This is sad, but not surprising: if building a metamodel requires the top expert developer in a domain, building a meta-metamodel for all such top developers requires the very highest levels of experience as well as intelligence and skill. The only other hope is dumb luck, but then the current authors probably used most of that up already, leaving little for newcomers!

We can thus assume the basics: a tool maturity level of at least 3, and the basic metamodeling and modeling facilities mentioned above. In the rest of this section we will concentrate on what is needed after such a generic “version 1.0.” Since each tool has its own focus, even version 1.0 will contain some of these features, and possibly even several from a given category. There is no stigma to be associated with the term “version 1.0”: every tool will have its first version. Similarly, there is not necessarily a benefit to being at “version 7.5”: in many ways, the thought and understanding already discernable from the first couple of versions are a better indicator of success than a high version number.

Clearly, the authors are associated with one particular tool, which happens to be one that has been developed for longer than many others—nearly 20 years at the time of writing. Unsurprisingly, given that time frame, many of the things we identify as useful have already been implemented in the tool. Indeed, the majority of these can already be found from the original articles on MetaEdit and MetaEdit+ (Smolander et al., 1991; Kelly et al., 1996): this is thus not a cheap attempt to list all the features of the current MetaEdit+ 4.5. Conversely, some things we originally thought useful and implemented have turned out not to be so valuable, adding little or even being detrimental. Although those features may be found in earlier publications, and some borderline cases may even still be present in the tools for the sake of existing customers, they are thus omitted from this list. Finally, although the list will necessarily be most colored by our own experiences and subjective viewpoint, we have always encouraged a frank exchange of views with other tool developers, and hopefully have been able to learn from their experiences and solutions—whether or not we have implemented similar features in MetaEdit+.

We will divide the features into the following categories:

- Meta-metamodel
- Notation

- Generators
- Supporting the metamodeler
- Generic modeling tool functionality
- Tool integration

The first three of these are the most important, as they form the foundation on which the remaining three are built. For the first two items, Chapter 10 looked at how to create a modeling language and notation, but not so much at the details of what features tools should offer for these. Chapter 11 however already discussed the various kinds of generator facilities found in tools (Section 11.2), so here we shall focus mostly on the first two items: the meta-metamodel and notation facilities.

14.3.1 Meta-Metamodel

The meta-metamodel is in effect the metamodeling language: the set of concepts provided to the metamodeler for building his metamodel. Just as each element in a model is an instance of a concept in the metamodel, so also is each concept in the metamodel an instance of a concept in the meta-metamodel. If the reader can cope with a shift in metalevel, we can say that the meta-metamodel is in fact a domain-specific modeling language for describing modeling languages. As such, it should aim to have a good set of concepts for covering all the commonly found artifacts of modeling languages. It will probably also have to be built specifically for this task: an existing general-purpose modeling language will tend to be at too low a level of abstraction.

For a first attempt, many people will still try to use an existing general-purpose modeling language: ER in the 1980s and early 1990s, or UML (or its MOF subset) nowadays. We can assume something like this as offering the basics of a meta-metamodel, which we shall not cover further here. Table 14.1 shows these basic concepts from ER—designed for describing databases, UML—designed for describing object-oriented code, and OPRR—designed for describing modeling languages.

Below we shall look at the extra features whose support will make or break a meta-metamodel. “Make or break” may sound strong, but of all the categories this is the one that has the greatest effect. If a meta-metamodel is lacking in some area, you will have to twist your modeling language to fit it. This will obviously be a burden to you when metamodeling, probably when building generators, and certainly for the modelers. Worse, even if the tool is later updated to include the feature you would have needed, there will be little chance of making the wholesale changes necessary to take

TABLE 14.1 Basic Meta-Metamodel Concepts

	ER	UML	OPRR
The basic model elements:	Entity	Class	Object
... are connected by these:	Relationship	Association	Relationship
... and both store information in these:	Attribute	Attribute	Property

advantage of it. Updating existing models for such a large change in a metamodel will often be rather like porting object-oriented code to a functional programming language.

Contrast this with improvements in the other categories, for example, notation or modeling tool functionality: there your modelers can benefit from the updates easily, without having to update models. It is also worth noting that in our experience as tool vendors, the meta-metamodel is something that is hard to change later: all the code of the tool is implicitly dependent on it.

Explicit Concept of Graph Early versions of tools tend to handle the objects of a model as one large cluster. This is rather like writing a program in one code file: it will work fine for a small case, but will not scale to be useful in real-world cases. There must thus be an explicit concept of a graph: a set of objects and associated relationships. This allows a model to be divided into submodels, with obvious benefits for manageability and multiuser use. It also provides essential capabilities for reusing parts of models, for example, building product variants from a subset of all graphs.

Larger-scale concepts can also be added above the graph. Just as a related set of graphs forms a model, a related set of models can be collected into a project, and projects themselves can be collected into repositories. The exact details are not important, but something extra is often needed in addition to the basic concept of graph.

Objects as Property Values In a version 1.0 DSM environment, objects in models can often only have properties whose values are simple strings, numbers, or Booleans. Surprisingly, even the humble string can cover almost all the data storage needs of most modeling languages. Even numbers are relatively rare: they tend to be needed more in the metamodel constraints or in systems generated from models. In models, most fields that would commonly hold a number tend to benefit from being strings, as this allows the substitution of a variable or named constant rather than a literal number.

The most common type of property value after strings is a reference to another object. This provides a powerful way to integrate models: by referring directly to another object, rather than just typing its name, a modeler solves a major maintenance headache. If the name of the referred object should change, all other model elements using it will still refer to the correct object. In some cases, of course, the extra level of indirection provided by a string can be useful, but the facility to refer directly to an object when necessary is vital. Any tool supporting strings will allow references by name; only by also supporting direct references can the metamodeler be given the choice of which method is most appropriate in each case.

Constraints on Property Values Although most properties are strings, they are by no means all free-form strings. In early versions of tools, there is often no way to check the input for a given property, and generators tend to spit out the value as is. If the value contains characters that are illegal in the context of the output file, this will

lead to errors either during compilation or, worse, at runtime. For instance, a string property to be used as a variable name may contain a space.

There are two possible solutions, discounting the idea that modelers should simply remember which characters to use in which fields. Either the generator facility must allow filtering or translation of strings from the model, or the metamodeling language must allow the specification of checks on property values. The former is probably better, as it allows more human-readable names in the model, but the latter approach is also useful. There are various ways to provide such support: MetaEdit+ uses regular expressions to specify legal values; IPSYS TBK had a text layout language that could handle simple cases, for example, free-form text separated by commas.

Constraints on property values can also look beyond the string itself to the wider context of the object and even graph where it is being entered. For instance, there may be a constraint that no two objects of a certain type could have the same name, either globally or at least not in the same graph. While such constraints could be handwritten in code, they occur frequently enough that supporting them in the metamodeling language is useful.

Explicit Concept of Role Few first versions of tools pay sufficient attention to the connections between objects. Even a casual examination of areas where graphical languages provide benefits over their textual counterparts reveals that visual links between objects are a major advantage. Anybody who has tried to read an XML file where elements need to refer to each other across the tree structure will be aware of how difficult it is to see such information: a graphical representation shows this information instantly. Since the graphical representation must also show the tree structure, there will be more than one kind of link between objects. These links must be represented differently: different line thicknesses and colors, arrowheads, and various decorations.

The different kinds of links also provide an important source of the semantics and rules of the modeling language: the differences between them are more than skin deep. As the semantics often depend on the direction of a connection, there must be a distinction according to the role an object plays in the connection: is it a superclass or subclass, a container or contained element. If we restrict ourselves to binary relationships, it is possible to consider this information as part of the relationship. Each relationship will have its own information, and a set of information for each end: what object it connects to, and any properties describing that connection. If we look a little further and allow relationships connecting more than two objects, that set of information is seen more clearly as a concept in its own right: a role.

N-Ary Relationships In a binary relationship, it appears that the arrowheads are part of the definition of the role the object plays in the relationship, but the line itself is part of the relationship. Looking at a ternary relationship, a connection between three objects, we see that the line too is actually a feature of the role. The meeting point of the three lines in a ternary relationship is the relationship itself, which may have a graphical symbol. Each of the roles can also have its own properties, as can the relationship. Applying this back to the binary relationship, we can see that this more

general case applies perfectly well there too: the line type may almost always be the same for both roles, but the symbols and properties will often be different for each role and the relationship itself.

The simplest kind of n-ary relationship is one where a role can be repeated multiple times, for example, multiple “To” roles in a Data Flow Diagram, or multiple “Specialization” roles to subclasses in a Class Diagram. This can be specified as the cardinality of the role in that kind of relationship. More complicated cases, for example, the Transition in the WatchApplication modeling language in Chapter 9, will have several different roles, each with their own cardinality, and each with a different set of object types it can connect to.

Weaker metamodeling languages that only support binary relationships generally attempt to make the excuse that an n-ary relationship can be replaced by a new object with a relationship to each of the connected objects. This is a poor workaround, as it loses the ability to check the legality of the relationship, or at best makes such checks much harder. Of course, most relationship types in modeling languages will be binary, as will most instances of them in models; still, having the choice to use n-ary relationships when you need them is a useful feature.

Reuse of Objects in Models Being able to reuse objects directly by reference is an important part of the productivity gains of DSM. The same object can be visible in many places, but in each case it really is one and the same object. An object can be reused in many different ways:

- As a property value of several objects, as seen above.
- By appearing several times in the same graph representation, as a notational convenience. In a diagram, this can be to prevent lines crossing or becoming too long, as in the Watch examples in Chapter 9. In a matrix, the same object will often be found on both axes, to allow relationships with itself to be shown.
- By appearing once in each of several different representations of the same graph. This allows radically different views on the same graph, for example, as both a diagram and a matrix, or as one diagram emphasizing inheritance and a second diagram emphasizing aggregation.
- As an element of several different graphs of the same modeling language. This allows a similar kind of reuse to functions and components in textual programming languages. See, for example, the states representing WatchApplications in Chapter 9.
- As an element of several different graphs of different modeling languages. This allows method integration: building a large model with different viewpoints or aspects described in different modeling languages.

Reuse also changes some preconceptions that we may have from simpler cases of modeling. For instance, looking at the case where an object is reused in several graphs, it is clear that an object cannot itself store all of its connections to other objects. In one project an object A may connect to B and C, but when A is reused in a different project

that may have changed. The information about connections should thus be stored in each graph, and when we ask questions like “what are the subclasses of X,” we must provide extra context: “... in graphs of project 1.” To take a concrete example from Chapter 9, we cannot simply ask “what application follows the Time application?”, since the Time application is reused in several Logical Watches. We must provide more context: “... in the TASTW Logical Watch.”

Links to Subgraphs A main reason to support multiple graphs is to allow the partitioning of a large model into several graphs. An immediate extension of this is to build a top-level graph where each object links to the corresponding lower-level graph. This kind of structure is found in many modeling languages, for example Data Flow Diagrams (Gane and Sarson, 1979) and the various kinds of hierarchical state transition diagrams.

The metamodeler must be able to specify which object types can link to subgraphs, and of which types. In some cases, relationships and even roles can also have subgraphs. There is also a wide variety of semantics associated with the subgraph links: Iivari (1992) finds five Boolean dimensions, for example whether the subgraph is owned exclusively by one object, or several objects can all have the same shared subgraph.

The possible uses of subgraphs seem to vary along a scale of how formal the link is. At the stricter end of the scale an object may only have one subgraph, and that link is the same wherever the object is reused. At the freer end of the scale the link is more like a note or hyperlink: an object may link to several different graphs, and that set of graphs may be different when the object is reused elsewhere.

Explicit Concept of Port Modeling languages for describing hardware often mimic electronic circuit diagrams in allowing an object to have a set of ports that connections must attach to. These ports may have different semantics and rules, for example, a “power in” port on one object must be connected to a “power out” port on another object. A port is thus a feature of an object type in a metamodel, and for ease of use it is generally associated with a particular visible node on the perimeter of the object symbol. Ports lend themselves to rules that are expressed over the whole modeling language: “in” ports must always connect to “out” ports; a “5 V” port cannot connect to a “110 V” port.

Some software component modeling languages have adopted a similar convention for the services provided and required by various components. In contrast with the first kind of ports, which are defined as part of the modeling language, these component interface ports are added by the modeler to each object as he sees fit. Since the ports are then part of the model, not the metamodel, they cannot be constrained in the same way by rules. Similarly, all such ports have the same semantics, reducing the ability to generate useful code from them.

Modeling languages generally fall into two categories: those that primarily use role and relationship types to express rules, and those that use ports. Where ports are used, there is often no semantic need for different role types: distinguishing between “To” and “From” is not done by role type, but by “In” and “Out” ports. Ports and roles can

still usefully be combined for notational convenience: a separate “To” role type to show an arrowhead when connecting to an “In” port, or constraining a “From” role type to always leave from the right or bottom edge of a symbol.

Language for Arbitrary Constraints The rules for connecting objects via relationships, roles and ports are a vital part of a modeling language. A good metamodeling language will allow the metamodeler to easily specify a range of common rules. Since these types of rules will be known by the generic modeling tool, they can also be checked efficiently.

No matter how wide the set of rule types offered, it will always be possible to invent a rule that cannot be expressed with them. It can therefore be useful if the tool offers the metamodeler a language to express arbitrary constraints. Using this language will however be harder work for the metamodeler, and checking the constraints will be harder for the tool—particularly in large models or multiuser situations.

Various kinds of language have been used for this task:

- Tool programming language
- Higher-level or logic language
- OCL
- Generator language

The naïve initial choice is often the same programming language the tool itself is written in. While clearly easiest for the tool vendor, this language will not be well suited to the task: it knows nothing of the domain of modeling languages and models. Writing constraints will require a large amount of low-level code, and debugging this mass will be a major difficulty.

For this reason several tools have turned to higher-level programming languages such as LISP or Prolog. A problem here is that the metamodeler will normally have to learn this new language, and these languages are generally found to be hard to pick up. Perhaps the best such language for this task would be Smalltalk, since most object-oriented programmers can learn it quickly, and it allows domain-specific constructs to feel like a natural part of the language.

Several patterns from Smalltalk were indeed used as the basis for OCL from the OMG (2006). OCL includes Smalltalk-like collection operators for transforming and filtering collections, plus a limited set of basic string, arithmetic and collection operators. It also include some extensions useful for constraints, for example `one()`:

```
collection->one(iterator | body_expressions)
```

which returns a Boolean stating whether the `body_expressions` evaluated to true for exactly one element of the collection.

While OCL was intended primarily for use in models to specify constraints on running systems, it can also be used one metalevel higher: in metamodels to specify constraints on models. As might be expected when a language is used for a somewhat

different domain than intended, some tasks have to use a rather low-level approach, and in other areas the language appears rather bulky. Some tools such as GME or XMF-Mosaic have indeed found it necessary to invent their own “versions” of OCL to address these problems, raising the question of whether such a language can claim to be standard anymore. A significant problem with OCL is its tight coupling with UML and MOF: to use OCL, you must use MOF as your meta-metamodel; using MOF, you can only build modeling languages that share a number of similarities to UML.

A different approach is to use the same domain-specific language for constraints as for code generation. This allows the use of a language specifically designed for navigating over models, without forcing the user to learn yet another language. If the output of generators contains links back to the corresponding model elements, the warnings and error messages output by failed constraints can be clicked to go straight to the offending place in the model.

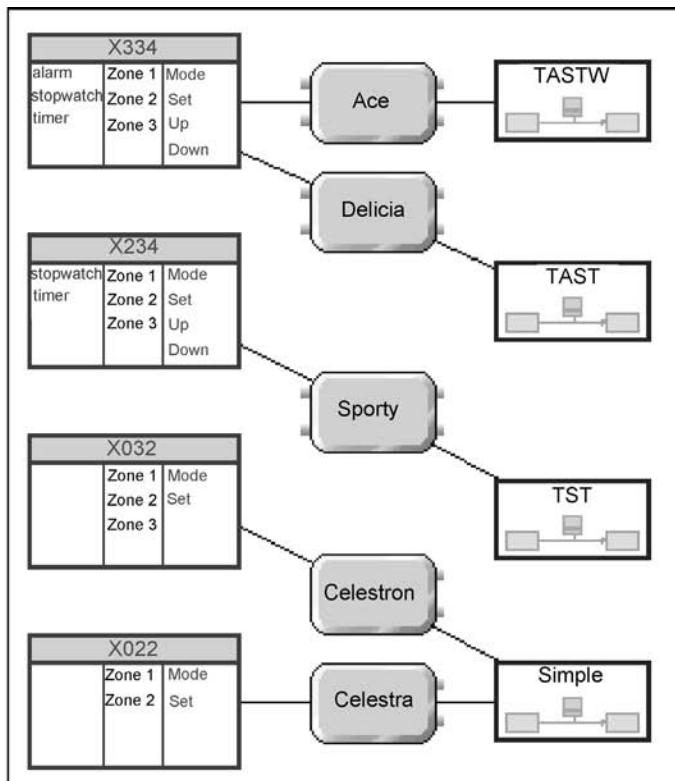
An interesting combination is to have the modeling language symbols directly contain elements for showing failed constraints. This places the error exactly where it should be, and makes it visible at the same time as it is caused—no need to explicitly run a constraints check. There are still cases where some constraints should be left to be run explicitly and as a test before generation: for example, if the constraint will often be broken, or checking it takes too long for execution on the fly.

14.3.2 Notation

If the meta-metamodel sets bounds on the abstract syntax of your modeling language, the notational features offered bound its concrete syntax. For tool vendors, the notation side definitely seems to be a harder task. Even the most basic part, the facilities for defining the graphical symbols, has proved troublesome to nearly all tool builders.

Representational Paradigms The clear majority of DSM languages have been graphical diagrams: nodes and edges, boxes and lines, bubbles and arcs. The same modeling language, or more exactly its abstract syntax, can however be represented in a number of different representational paradigms: as a matrix, table, or structured text.

A matrix representation shows the objects on the axes of the matrix, like row and column labels in a spreadsheet, and a relationship between a pair of objects is shown in the appropriate cell. The matrix is thus useful for focusing on relationships, particularly where there are binary relationships each connecting an object of one type to an object of another type. An example of this was seen in Figs. 9.5 and 9.6 for the Watch family diagram. In the former, each WatchModel object had properties referring to the Display object and LogicalWatch object from which it was composed. In the latter, these structure were turned into WatchModel relationships that directly connected Displays and LogicalWatches. Figure 14.1 shows a diagram and a matrix displaying the relationship approach, and a table displaying the property approach.



Model name	Display	Logical Watch
Ace	X334	TASTW
Delicia	X334	TAST
Sporty	X234	TST
Celestron	X032	Simple
Celestra	X022	Simple

FIGURE 14.1 Diagram, Matrix, and Table representations

As should be clear from the picture, sometimes the table and matrix representations have significant advantages in terms of compactness and scalability. Mapping between four Displays and four LogicalWatches works reasonably in a diagram: in this case the lines do not even cross each other. Somewhere between four and eight this kind of diagram would probably break down, if the WatchModel connections were random. The earlier diagram, Fig. 9.5, would scale slightly better, but is perhaps less clear visually. A matrix would cope with up to 15–20 of each kind of object, and many more WatchModel relationships, while providing a good view of both objects and relationships. A table would scale to over 50 WatchModels, but offers less help in visualizing which Displays and LogicalWatches are used where.

Graphical Symbol Editor Given that the direct manipulation vector graphic editor has been around since 1961 (Sutherland, 1963)—8 years before ARPANET and IBM’s first 1 KB RAM chip—it is hardly impressive to say that of all the tools, to our knowledge only two have provided metamodelers with a graphical symbol editor: MetaEdit and MetaEdit+. This must surely change in the near future, but the failure of earlier tools leads us to suspect that it may take a little time before the other current tools get it right.

Getting it right is however worth the effort, for the sake of both the metamodeler and modeler. Two of the most common criticisms of tools without a symbol editor are the lack of freedom in defining symbols using the facilities offered, and the massive amount of time it takes to program the display of such symbols by hand. Even with a symbol editor, significantly more time is spent building the symbols than building the abstract syntax of the metamodel: concepts will always be simpler to handle than representations.

Without a symbol editor, the amount of time required to define symbols will often approach the ridiculous. For instance, Eclipse’s GEF comes with a simple example modeling language for logic circuits: AND, OR and so on. The code for the editor comes to over 10,000 lines, mostly for the graphical symbols. For the average Java programmer, that represents a little over one man-year of work. In MetaEdit+, implementing the same language took just one hour: over 2000 times faster. Even an author of the IBM Redbook on EMF and GEF (Moore et al., 2004), Anna Gerber, accepted the difference of roughly three orders of magnitude, although she would expect the actual figure to be 600–700. Mind you, she is probably faster than the average Java programmer!

Some tools have tried offering a limited range of preprogrammed symbols, configurable by size, color, and so on. While a similar approach has worked well for rules, with symbols its limitations are reached too quickly. The difference probably lies in the fact that the kinds of rules that are needed are to be found from the domain of metamodeling, with elements such as objects, relationships, and properties, whereas the kinds of symbols that are needed are to be found from each modeling language’s own problem domain. Trying to fix the set of possible symbols ahead of time means restricting them to simple geometrical figures or those used in other, mostly generic, modeling languages. An important part of DSM is that the symbols should be

evocative of the domain concepts they represent, and to do that requires free combination of the basic graphical elements—lines, curves, text, and so on—to make domain-specific symbols.

There seems little point in describing the features of a graphical symbol editor here: it should simply behave like a normal vector graphics editor. The difference is in the integration with the modeling language definitions: a given symbol is defined for a particular object, relationship or role type. Text elements will thus normally be references to the properties defined in the type, and the editor should be integrated with the definitions to make selecting the right property easy.

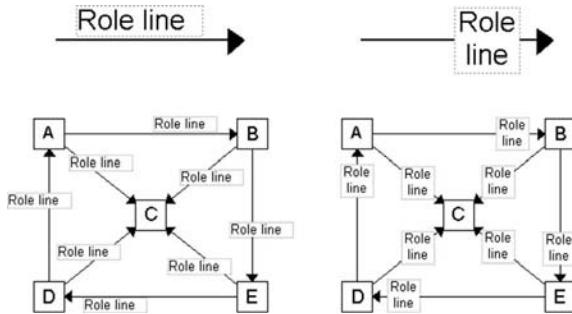
The editor should also be integrated with the modeling tools, so when a symbol is altered, existing models update immediately. Seeing a symbol in several places in a model, possibly with different scales and zooms, gives a much better impression of how well it works than seeing it once in the symbol editor, often zoomed to make editing easier. This once again emphasizes the importance of having the metamodeling and modeling tools integrated in the same environment.

Role Symbols The symbols for roles are something of a special case, because they must rotate with the line over which they are displayed. Tools without a true concept of role will often offer just a few simple choices of arrowheads, but even a cursory look at existing modeling languages will reveal far more variation. Since vector graphics can be rotated with relative ease, it seems simplest to allow the same freedom to roles as to other symbols.

Text elements are somewhat problematic in role symbols. Most modeling languages have found that text works best when displayed horizontally. A slight angle or incline to text along a role line is no great problem, providing the font can be rendered suitably smoothly (still a challenge in many cases). Anything over 45° of incline tends to render models much harder to read, with the user having to crick her neck back and forth at a different angle for each role. Since that represents half of the possible role line angles, rotating the actual text content tends to be a poor idea. With Manhattan (i.e., perpendicular) routing the situation is even worse: half of all role lines are at the absolute worst angle, vertical.

If the contents of text elements are to be displayed horizontally, there is the question of how their position and size react to changes in the role line angle. Most people's initial attempt will have the text offset above a horizontal line, but this leads to problems when the line is rotated to be vertical. If the text box is also rotated, it will now have many rows each of one character. If the text is not rotated, a long text stretching out perpendicular to a vertical role line will lead quickly to confusion, with texts crossing several such role lines. Even with shorter texts, having the text offset above a horizontal line will mean that when rotated to point upward, the text will be offset to the left of the line.

Some of these problems can be seen in Fig. 14.2, which shows two different symbols for a role at the top, and then effects in a model below. The tool here is using what we have found to be the best approach to rotating text elements in role symbols: do not rotate the content or boundary of the text, but allow its midpoint to rotate with

**FIGURE 14.2** Role symbol texts

the role line. The offset to the left mentioned above can be seen on the left in the role at A from D. The roles at C from A and E appear to be at the wrong end of the line: the midpoint of the text element is rather far from the role end in the symbol. A better solution overall is on the right, where the text element is square. The text remains symmetrically aligned over the line, and close to the correct role end. This approach will work even in tools that rotate the text boundary. The downside is that the role line will always cross the text element, unlike AB and ED with the previous symbol.

Role Line Routing An important part of the tool support for a graphical modeling language is how role lines connect to objects. A role line must follow a certain route, with the last leg pointing to a certain point in an object symbol, and stopping where it intersects the symbol.

The route of a role line is generally made up of straight lines between freely positioned points. Most commonly, the line will simply head directly from the relationship to the object with no intermediate points. This is the optimal case, as the eye can follow it most simply to read models: the line is the shortest distance between the objects it connects, and thus lies on the natural path of the eye between those objects.

Those with a background in chip design will tend to look for a perpendicular or Manhattan routing of lines: all segments of the line are either horizontal or vertical, with no diagonal segments. While this is harder to draw and to read, and is based on restrictions in electronic design that are not present in modeling languages, it may be hard to persuade users used to Manhattan routing to adopt a simpler approach.

A similar case is the use of curves for roles instead of lines. For some reason this is generally combined with a dotted or dashed line style, making the diagrams even harder to follow. Since editing these curves to look as desired is time consuming, and there is no evidence of any benefit from them, it is probably wisest to avoid them.

It is also possible to define algorithms for automatic routing of role lines, optionally also with layout of objects. A good tool will include some automatic algorithms, although these remain a challenging issue: since the tool cannot know the modeling language, the algorithm cannot take into account special requirements or semantics of different kinds of objects and relationships. DSM models differ in this from simple

generic modeling languages, for example, where a single kind of relationship forms a simple tree structure. Any decent algorithm is sufficiently complicated that writing a bespoke version for a DSM language is a major undertaking, and the results are unlikely to be reliably good. Short of bespoke algorithms, the best results so far have been achieved with configurable hybrid algorithms: the user can choose certain kinds of objects and relationships to be routed one way, for example, as a Manhattan tree, and others to be routed by a more generic least-line-length approach.

Connection Boundaries for Role Lines Whatever the route, the last leg of a role line will point toward a certain point on an object. Normally this should be the center of the object, since that is clearest for the eye to read, as well as being esthetically balanced. It also means that the modeler need do no extra work to keep separate the intersections of roles from various sources. As he moves connected objects around, their role lines automatically follow, pointing at the center all the time. In cases where the visual center of an object is not at the center of the total area of its elements, it should be possible to define a specific target point that role lines head to. For instance, in a Use Case Actor symbol you may want roles to connect either to the stick figure, and move the target point up there, or to the text label below, and move the target point down to the center of the label.

“Version 1.0” tools may not offer true center-based role directing, since it requires somewhat more work in calculating the intersection of the role line and the symbol. Instead, they may offer only a limited number of points, normally the midpoints of edges and/or corners of the enclosing rectangle of the symbol. All roles must connect to one of these predefined points, with less readable and esthetic results. The limited range of connection points also forces symbol designers to make sure those points are on the boundary of the symbol, otherwise role lines will stop somewhere in empty space before the symbol, for example, a horizontal line to the middle of an hourglass shape.

Rather than allowing a few fixed connection points, a tool should allow the user to specify a connection boundary around a symbol, at which incoming role lines will stop. In the simplest of cases, where a symbol consists of a single graphical element, that element is itself the connection boundary. Once there are more elements, defining a connection boundary becomes something best left to the metamodeler, although reasonable defaults should be provided. The boundary will generally be a polygon that follows the outline of the outermost edges of symbol elements. Incoming role lines will calculate the intersection point with the first edge of the polygon that they meet, and stop there.

More complicated cases occur where a symbol contains explicit ports, represented by visible nodes on the boundary of the symbol. When connecting to a port, the modeler must normally explicitly choose which port, for example, by clicking that area of the symbol in the model. Since the port is not normally a point, but a small vector graphics shape of its own, it too needs a way to specify where role lines head to and where they stop. Since the role lines cannot head to the object center—they would miss the port—the port should contain its own target point. Rather than draw one

complicated connection boundary that snakes its way around the ports, it is easier to allow each port to specify its own connection boundary. This also takes care of the case where a line to one port would also first happen to pass through another port's area: we do not want the line to stop there.

Generated Text Elements The majority of text elements in a modeling language will be simple references to property values. Sometimes, more flexibility will be needed, and the text content must be built up from more than one part. For instance, some labels show text in brackets, like UML's “{abstract}.” If the text in brackets can be of different lengths, even this simplest case cannot be handled by a couple of bracket characters as fixed text elements either side of the property.

A reasonable subset of such cases can be dealt with by the ability to specify a sequence of properties and fixed strings. Inventing a new mini language or GUI to specify such sequences should not, however, be necessary. Instead, it is better that the tool simply uses its existing generator language. The language is designed for navigating models and outputting text, which is exactly what this task calls for. If the language is at a high level of abstraction, even a graphic designer who is not a programmer could use it.

Using the generator language also allows us to go further afield in the search for text content to display. For instance, a role may pick up information from its object or port, to show more exactly what the connection is doing. Since the context of the whole graph is available while drawing a symbol, we can also use symbols to provide information about the graph as a whole: errors, warnings, metrics, and so on, as mentioned above.

The downside of complicated generators is that there is no way to know when their content may have changed. If the generator can read any information from this graph, and possibly any subgraph or supergraph, the content could change after almost any operation. If the generator language is powerful enough to pick up representational information (e.g., object positions) or external information (e.g., the results of a system call), there is no hope. A spreadsheet may be simple enough that a smart recalculate algorithm can cope with most cases, but a DSM tool is more complicated. There is no major problem with this: most often the display will be updated anyway, since most operations that affect it will be to closely related elements. In other cases, a simple “Refresh” key will suffice: users are accustomed to programs needing a little nudge in these areas. If the content is valuable, it should not be thrown out because the world is imperfect. Conversely, if the 100% accuracy of the content at all times is vital, we can simply move the generation to a regular generator that is run on demand to produce its output in a separate window.

Conditional Symbol Elements Property values and generated text elements give us most of the dynamic behavior we want in symbols, but in some cases we would like to “generate” something other than text, for example vector graphics elements. This may initially seem to be a task for tools where each symbol is hand programmed, but on closer inspection that is unnecessary. The elements that are to be added are

almost always known in advance, so they are not so much generated as displayed conditionally on some value from the model.

Most commonly such elements are small icons or single icon-like characters, displayed based on a property whose value is a Boolean or a selection from a fixed list or other enumeration. It should thus be possible to associate a condition with an element or group of elements, comparing a property to a fixed value. By allowing dynamic information to be displayed graphically rather than textually, conditional symbols can play an important role in creating a truly visual language.

More advanced cases may effectively replace the whole symbol with a different one, depending on the value of a property. This is however somewhat extreme, and normally motivated by problems elsewhere in the modeling language or tool. Rather than effectively merging two types into one, separating them by a new property, it is generally better to keep them as separate types, possibly with a common supertype. The desire for different symbols is probably telling you something about their different semantics, and that will probably later be revealed in differences in rules, properties, and so on.

For individual elements, the conditions may sometimes be more complicated than a single property. As with the contents of text elements, the next step after properties could well be the generator language: that can fetch whatever values are necessary for the condition. On other occasions the value may be from a single property, but the test will not be simple equality with one of a predetermined set of strings. For such cases it is useful that conditions can perform wildcard, regular expression, or numeric comparisons.

14.3.3 Generators

Section 11.2 already looked at different types of generator facilities and many of the features that are desirable in such tools and languages. Here we shall briefly recap the main desirable points of the languages, and concentrate on the tool support for their use.

High Level of Abstraction The language for defining generators should be at a high level of abstraction. While building a generator, the developer has to cope with the language of the models, as well as the language he is trying to output. Both of these take their own slice out of the available brain power, so the less required by the generator language the better.

If the syntax of the generator language is similar to the syntax of the output, it becomes difficult for the brain to separate fixed text parts from the generator language—even with good syntax highlighting. While there is obviously some benefit to using a familiar language, the worst possible case is when the language being generated is the same as the language used to generate it. Such a situation often leads to “metalevel blindness”, or in this case deafness: as we try to understand a program, we often read it to ourselves, and the lack of verbal cues to separate generator from generated easily leads to errors.

Concise, Powerful Navigation A large part of the task of a generator is to traverse the models, so the generator language must have concise yet powerful constructs for model navigation. All too often we see languages with virtually no navigation constructs, leaving the task of navigation to functions generated in each object type. For example, a Button object type may be given a function “triggeredTransitions(“), which answers all the transitions that are connected to this button. This will work fine in a single metamodel, in a single model, where each Button is used in only one graph. However, if the Button is reused in several graphs, or worse in several metamodels, there is no way there can be a single answer to that function: which transitions are triggered depends on which graph we are in, and in some types of graph Buttons may not be used at all for triggering transitions. Lack of language-level navigation constructs thus kills reuse, and inhibits metamodel evolution.

With language-level navigation, the context of the call is known at runtime—including which graph and object we are in. A command like “do >Transitions” will effectively evaluate to something like “currentGraph.relationships(currentObject, Transition).” (See “Reuse of Objects in Models” in Section 14.3.1 for more details on the underlying issues.)

Many-to-Many Mapping Between Graphs, Generators and Output Files A “version 1.0” tool will generally assume that a single model or single generator maps to a single output file. The first assumption drags the modeling language down toward the abstraction level of the implementation, forcing the same division of information as there. While partial classes or preprocessor include statements may help in some cases, it is much better if the generator facilities do not impose their own view on the relationship between the structures of models, generators, and output. A single generator should be able to output multiple files, a single file should be able to be built from information spread across multiple models, and so on.

Output Filtering After navigation, the second main task of the generator facility is output of information from models. If the information is already exactly in the format desired, that will be simple. Often, however, this would require modelers to know and remember exactly how a given string in a model will be used in the generated output: if it becomes a variable name, no spaces will be allowed; if it will be in an XML file, ampersands must be replaced with the & entity. Rather than burden the modeler and force models to be less readable, it is better that the generator language offers good facilities for translating output.

Generic functions like “toUpperCase(char *)” may be useful, but there will always be new cases and exceptions. Within a given DSM solution, certain of these translators will be used many times, so having a shorter syntax will also be useful. A good solution is to allow generators to specify reusable translators in a declarative fashion. It should be possible to use a translator while outputting just a single piece of model information, or to be able to turn them on for a longer block of the generator containing multiple commands.

Syntax Highlighting Most developers have come to expect syntax highlighting from their IDE. In generators this is even more important, as they are a mixture of three syntaxes: the generator language, modeling language, and output language. The generator language and modeling language can be highlighted easily, and at least a single color format can be applied to all fixed text elements—effectively the output language. In all but the simplest template cases, the language in which output is generated will be so split up in the generator as to make parsing it largely useless.

Currently, all generator languages separate fixed text elements from generator language elements with some character sequence. As that sequence will be used frequently, it is useful to keep it as short and unobtrusive as possible: a quote mark is faster to type and read than a template scripting style sequence like <% =. It would be interesting to experiment with an approach where the separation was achieved purely with formatting, for example a gray background versus a white background.

Metamodel Integration After the generator language itself, the main language visible in a generator is the modeling language. The generator editor should therefore allow easy navigation and selection of the concepts of the modeling language and their properties. In simple cases it is possible to update generators automatically when the names of modeling language concepts change.

The generators themselves will normally be associated with a specific modeling language: they can only sensibly be run on models of that type, and will contain references to the concepts of that modeling language. It would be possible to go further, and allow generators to be associated with individual object, relationship and even property types. In our experience, however, a full generator will normally consist of less than a few dozen subgenerators. As this is approximately equal to the number of concepts in the corresponding modeling language, scattering the parts of the generator across all types will make generator development less manageable.

Tracing from Output to Models and Generators Particularly while building generators, it is useful to be able to track a given piece of output back to the model element and generator command that produced it. In cases where the intention is not to hide generated code from modelers, being able to track back from output to model is even more useful. Rather than littering the output with numerous trace comments, it is much neater to be able to hide these links “behind” the visible text, in a similar way to hyperlinks in HTML.

Debugger As with most development tasks, building a generator is easy until things go wrong. The fact that generators produce copious output makes their behavior particularly visible: in a way the generator works like a huge set of debugging trace output commands. However, in cases where no output is produced where some was expected, having a true debugger is invaluable.

The debugger should take advantage of its knowledge of the task of the generator, providing clear visibility not only of the current generator call stack but also of the position and history of the model navigation, and the current state of the contents of

the output or outputs. Normal facilities for stepping through generators, and over or into called subgenerators, quickly answer most questions about why the generation is not working as expected.

Being able to place breakpoints in a generator saves time in getting to the offending point, if that is known. Where the part of the generator is not known, it may be possible to set a breakpoint on a model element: when the generator navigates to that element in the model, a break is triggered.

14.3.4 Supporting the Metamodeler

In “version 1.0” tools the metamodeler is often left to figure things out on his own, and to hand code things the tool builders were unable or unwilling to implement. Here are a few of the things that help make it possible to metamodel efficiently.

Stability and Compatibility A difficulty with most prototypes or initial versions is their stability. Here we are not referring primarily to the fact that such tools may crash, but to problems encountered when upgrading to the next version. At least so far, such tools have provided poor or nonexistent support for using modeling languages and models built with the previous version. Of course such tools must move forward and correct design or architecture decisions from the initial version, but still it should normally be possible to automatically upgrade old data to the new version. The fact that this has not been so is largely due to the low level of maturity of these tools on our scale of 1–6. Since the metamodeler needs to write extra code by hand, it is effectively impossible to provide automatic updates: data can be updated, but rarely code.

Documentation and Support It should go without saying that tools should be accompanied by good documentation. This should cover the tools themselves and the metamodeling and generator language they use, as well as tutorials to provide an easy path in. For most metamodelers, this will be their first attempt at creating a modeling language, so more background information and explanation will be necessary than for many other products or frameworks.

Good technical support for the DSM environment will be an important consideration for any serious project. In the early days of a tool, the few early adopters will often receive good support directly from the tool developers themselves. Should a tool prove even moderately successful, the increased number of users requiring support will often swamp the developers. It is thus important that systems are in place for providing technical support, training, and consultancy.

Existing Libraries For the new user, an existing library of DSM solutions can be vital to understanding how DSM works and how to do it well. Although by the nature of DSM only small parts of these could be reused unchanged, investigating how a well put together solution works can have a major effect on how quickly a newcomer gets up to speed.

Browsers and Info Tools Whether modeling languages are built as graphical models, forms or a textual language, the metamodeler requires powerful and flexible tools for browsing, searching and navigating the metamodels. In particular, integration between modeling and metamodeling tools is important: most ideas for improving a modeling language come while using it. The metamodeling experience can be significantly improved by good support for navigating from an element in an instance model to the corresponding concept in the modeling language.

On-the-Fly Metamodel Updates As we have mentioned elsewhere, one of the most important features is the ability to update metamodels on the fly and have models update automatically. The improvements this brings can be likened to moving from character-mapped displays to WYSIWYG word processing, or from debugging by inserting trace output statements to live debugging. These kinds of advances are technically challenging to implement, but the improvements in metamodeler productivity and user experience are undeniable. Of course, there are two levels: updates on the fly are important while the metamodeler tries things out, both when building and maintaining the language. When the modeling language is in production, the metamodeler will rarely be editing the language on the fly, but rather working off-line from the other modelers, and only releasing a complete and tested set of changes as an update to the modelers. The models must then update to match the new metamodel, but that is a slightly different issue from the instant feedback to the metamodeler of on-the-fly changes.

Automatic Icons, Palette, Menus, and Dialogs In early versions of tools, the metamodeler is often required to specify by hand the icons, type palette, and sometimes even menus for the resulting modeling tool. This is a clear case of duplication of data and effort: the metamodeler has already specified the object types, their symbols, and which object types and operations are allowed in a given modeling language.

The manual creation of bitmap icons for lists and palettes is particularly time consuming: such icons can be produced automatically from the symbol definitions with appropriate filtering algorithms. This also removes the problem of producing several different sizes of icons for different uses, platforms, or screen settings. Interestingly, today's graphical designers use this approach when building fixed-size bitmap icons: the icon is drawn as a vector graphic, and automatically scaled to the variety of sizes stored in a multibitmap icon file.

Generating all of these elements automatically ensures that all parts of the modeling tool remain synchronized with the definition of the modeling language. More importantly, this approach significantly reduces the work required of the metamodeler: the task is now simply defining the modeling language, as it should be, rather than building tool support for it. The metamodeler is the expert in the domain, but the creators of the DSM environment are the experts in modeling tool behavior. Any flaws in the behavior should be addressed by the creators of the DSM environment, helping all metamodelers, rather than separately by each metamodeler.

Automatic Property Dialog, Custom Layouts The definitions of the properties associated with a concept should be sufficient to enable automatic creation of proper forms or property dialogs for editing instances of this concept. A true form with the normal data entry widgets appropriate for each property's data type is the ideal format for editing property values.

Simple table-like property sheets can only handle short, simple values well, and are best saved for a quick read-only display of the current element's values. Where such sheets are constantly visible, their size and font size will always be a compromise between the usability of the sheet and the space left for modeling. Tiny fonts and short space for values may be acceptable for read-only display, but not for editing values. As user feedback is strongly in favor of proper forms with standard data entry widgets, the property sheets are presumably just a quick, cheap solution for the initial version of a DSM environment. The true costs of this saving are however passed on to the modelers, or then to the metamodelers who have to program a proper form by hand.

In-place editing of property values in symbols is an area that requires more investigation. DSM languages strive to avoid representing everything as simple text: if the entirety of a model is seen as text values in its diagrams, there seems little benefit to using a graphical representation. As the values within an object should remain mutually consistent, only editing one value while many are not visible tends to be a poor solution. The user interface for choosing to edit a property value in place, as opposed to simply selecting an object, also frequently causes headaches for both tool developers and users. The display and behavior of the value while it is edited are similarly problematic. Only if these issues can be overcome will in-place editing be a useful addition to true property dialogs.

Automatic Language Help Particularly for new users of a modeling language, good documentation of the language is essential. While nothing can beat or automate the creation of well-written introductory material, a tool can at least provide a textual, human-readable representation of the concepts and rules of the defined modeling language. The metamodeling language and tools should offer fields to add free-form textual documentation of the various concepts, to be incorporated into this automatic language documentation. Keeping this information with the definition of each concept gives the best chance that it will be kept up to date: unlikely if the description is written in a separate file.

14.3.5 Generic Modeling Tool Functionality

Unsurprisingly, the level of modeling tool functionality provided by most initial releases of DSM environments is modest at best. With research tools, this is as it should be: efforts should be focused on new, innovative ways of defining modeling languages, and even empirical trials will focus on real users working as metamodelers, not modelers. For tools intended for actual use in modeling, deficiencies in this area are however serious. Unfortunately, the consequences of these failings reflect poorly not only on the tool itself but also on DSM as a whole. As there are many modelers for

each metamodeler, the majority of experiences of DSM across the industry will be based on the modeling tools.

Basic Functionality To be considered for real use, tools must offer at least the basic functionality expected of any program these days, regardless of whether it is a graphical editor, email software, or word processor. These functions include multilevel undo and redo; cut, copy, and paste; direct manipulation of editable elements; and printing and exporting to various file formats.

When copying and pasting, there should also be the possibility to affect whether a direct reference or new copy is created. With complex objects, it must also be possible to specify how deeply contained elements should be copied: objects in properties, subgraphs, and so on.

Direct manipulation is particularly important for laying out diagrams so that they are easy to read and work with. With today's computers capable of billions of operations a second, it seems odd that so many tools fail to make relationships and role lines follow objects visually as they are moved or scaled.

Multiple Models For some reason, initial versions of tools often omit the ability to work on multiple models (graphs, diagrams, etc.) simultaneously. In a text editor, this may not be so important, as links between files are formed by typing the same sequence of characters as appears in the other file—for example, a function name. In modeling, one of the major benefits is that such links are now made by direct reference, ensuring that links are always synchronized. Tools that fail to support multiple simultaneous models must either fall back on error-prone string references, omit linking altogether, or make significant architectural changes to enable true linking in subsequent versions.

Multiple Modeling Languages Integration between models is not just between models of the same modeling language. Even early languages like Structured Analysis used multiple interlinked modeling languages (Gane and Sarson, 1979), and the more unwieldy of today's generic modeling languages may have over a dozen different diagram types. Modeling tools must thus support the simultaneous, integrated editing of multiple models of multiple modeling languages.

Multiple Users For all the research on collaboration and groupwork in the 1990s, surprisingly few tools support multiple simultaneous users. As real-time communication and collaboration has spread to the general populace through chat, online games and Internet video and phone calls, software developers have been left like the proverbial cobbler's children.

Allowing multiple simultaneous users to edit the same diagram seems unnecessary: attempts at this in the 1990s revealed more problems than benefits. It is however useful to allow multiple users to update the information contained in multiple distinct objects within that diagram: the objects may be reused in several places, and edited from

there. Everyone should be able to see the latest version of a diagram and reuse its elements, but only one user should be able to edit it.

When designing the meta-metamodel, or to be more exact the data structures that will be used for storing model data, the creators of the DSM environment must consider multiuser issues. They must choose a meta-metamodel that avoids creating hot spots: structures that become a target for parallel updates by multiple users. Where that is unavoidable, they must implement data structures that are appropriate for expected patterns of multiuser access (Kelly, 1998).

Browsers Information in a modeling tool forms part of a relatively complex structure, as with source code. In contrast with source code, much of the structure is made explicit in the internal data structures of the modeling tool, rather than having to be inferred by finding the same character sequence in multiple places. This makes it relatively easy to provide a variety of browsers showing the interrelations of models and their elements.

Elements can be divided by several hierarchies, dimensions, or link structures, for example,

- their types
- the projects they are defined in
- the graphs they are used in
- the hierarchy of graphs
- the hierarchy of objects used as properties

All of these structures can serve as a useful basis for browsing existing objects to examine the state of an application or look for reusable elements. A tool should thus offer browsers supporting hierarchical display along these dimensions, with filtering by project, graph, or type. As these browsers can be built independently of a particular modeling language, they form a good example of the kind of generic modeling tool functionality to be expected from a DSM environment.

Documentation Generation In many senses, a domain-specific model is its own documentation, and the modeling tool is the best way to view the documentation. However, not all users will have access to the modeling tool or be familiar with it, and existing organizational practices probably mandate the use of a particular kind of documentation such as HTML or word processor documents. It is thus useful for a modeling tool to offer export of a complete set of models as documentation. This is discussed in more detail in Section 11.3.3; here it suffices to say that a DSM environment should offer a good generic format for such documentation. Rather than hard coding the function, it should be provided through the tool's generator facilities, allowing customization for the requirements of a particular DSM solution.

User Documentation and Support As with the metamodeling tools, the modeling tools must also be documented and supported. The users of the modeling

tool will normally be unable to separate issues caused by the particular DSM solution from those common to all tools built with that DSM environment. Bug reports and technical support requests must thus often be directed through the DSM team. The quality and maturity of the generic modeling tool will determine whether this is a burden which that team is willing to bear.

14.3.6 Tool Integration

Compared to traditional modeling tools, DSM tools play a larger role in the toolset of the developer: since full code is normally generated, developers often need no longer interact directly with an IDE or compiler. Unlike the dead end of round-trip engineering faced by modeling tools working at the same level of abstraction as the code, DSM keeps code and models clearly separate yet in step. With DSM, each code file is either fully generated—in which case it is guaranteed to be synchronized and yet can also be thrown away at will—or fully handwritten, in which case there will be minimal coupling with the models.

Standalone versus IDE Integration The need for integration with an IDE is debatable: a recent independent survey of lead developers in Europe revealed that only 7% considered Eclipse integration an important feature for a modeling tool, and only 4% wanted VisualStudio integration.

As is often the case, we find good predictions from the move from assembly language to third generation languages. Just as compilers do not fill today's IDE projects with the assembly files generated from higher-level source code, so it is to be expected that the code produced from a DSM tool will be treated as an intermediate by-product, not a central component. In truly model-driven development, the development environments for models will be focused on modeling, with support for examining code either secondary or delegated to dedicated code IDEs. Unwieldy project trees will be replaced with more visual graphical modeling languages, tailored to the needs of the particular domain.

Call External Tools The integration of DSM tools with other tools and information sources and sinks in the development process thus tends to be one where the DSM tool is calling the shots. A common pattern is for the tool to generate source code files, invoke the compiler, and start up the resulting application for testing (see Section 11.4.1). The most important integration feature in a DSM tool is thus its ability to script the invocation of other tools. The natural medium for this would seem to be a generator language that can produce make files, build scripts, batch files or shell scripts as necessary, and invoke the command processors for these.

Be Called by External Tools Calling external tools will be the norm, but there are also cases where it is useful for other tools to invoke the DSM tool. One common case occurs in organizations with an existing practice of automated nightly builds. Since the primary source is now a model, it should be possible to invoke the tool on a

particular model, generate code from it, and exit. This kind of integration is relatively simple, requiring only support for appropriate command-line parameters.

A reference in documentation, version control or a bug management system may point to a particular model or part of a model, and being able to start the tool and open the relevant model can save time. For the first such invocation command-line parameters will suffice, but if there is a need to prevent multiple copies of the tool being opened there should be support for targeting an existing running instance.

API Some integration needs go beyond simple sequential invocation to cases where an external program works in tandem with the DSM tool. Common examples include emulators or simulators that run and animate the models from the DSM tool.

To support this, the DSM tool must offer an Application Programmer's Interface (API), which reveals a set of its functions that can be called externally. In DSM tools integrated in an existing IDE, an SDK offers tighter integration and lower-level access, but only to programs running in the same IDE process. Since external tools will by definition rarely fall into that category, a separate API is necessary here too.

The protocol used for API communication is something of a difficulty. Older tool APIs tended to be programming language or platform specific, reducing the range of tools that could use them. SOAP appears to offer the best solution currently, as support is available for all but the most esoteric platforms and languages. With SOAP, the interface is at a high level of abstraction. A single call is made to initiate a connection, and commands are issued to the remote tool in the same way as to local functions: the SOAP client takes care of the encoding and decoding of the various data types, as well as the details of the communication.

The second important consideration for an API is the method of data access: should the data structures of the meta-metamodel, metamodel, and model be duplicated in the client program, or should access be via commands to be performed in the remote tool. Duplicating data and data structures is time consuming, unsuited to deep, highly interlinked structures found in models, and causes problems in keeping remote data and local copies synchronized. A pure command-based solution requires a way to identify the model elements to be operated on, and a way to pass model elements as the result of a call.

An appropriate solution in many cases is provided by proxy objects, which are handled in the client program as if they were the real model element, but simply pass any operations sent to them back to the remote tool. There is thus no need to duplicate data or data structure definitions, and the issue of identity is encapsulated within the proxy objects. Functions returning model elements actually return proxies, which can be used as parameters in further calls. Functions returning strings or other primitive data types can return the actual string.

Model and Metamodel Interchange A tool should support the import and export of models and metamodels, for exchange of information with either other instances of the tool or other tools. If the format is to be used with other tools it must of course be tractable and documented. The predictable choice would be some form of

XML, but other simpler text formats have also been used. For exchanging models and metamodels with another instance of the same tool, a binary format will suffice.

When exporting to another instance, tools at levels 5 and 6 must solve the question of whether to provide the metamodel with a model—tools at earlier levels can do little more than explode if the file to import was made with a sufficiently different version of the metamodel. If the metamodel is included, the export format will be suitable for use with any instance of the tool, even one that has never seen that metamodel before.

Where an external program wants to operate on the majority of a model or set of models, importing a file is probably more efficient than attempting to work through an API. Working on the imported data and exporting it back to the same tool in another form is almost always a bad idea. A more appropriate use of model-to-model transformations is to process the information into the format supported by another tool that will display, analyze or further process it. In these cases the target “model” is generally not a model in our sense of the word, is read-only, and will be discarded after the process is complete.

Exporting a metamodel to a different tool may be useful when configuring a tool that is later to process exported models. Currently, exporting a metamodel from one DSM tool to another has predictable problems with loss of functionality: only the lowest common denominator is supported by systems such as KM3. For transfer between current “version 1.0” tools this is less of a problem: they have little beyond this, and what they have tends to be expressed as code anyway, which could not realistically be transformed. In a situation with several more mature, data-based tools, better results could be expected from bespoke translations between a pair of tools.

Where a tool supports metamodel import from a textual format, interesting possibilities exist for turning models into metamodels. A modeling language and generators can be built that produce this format, and the results can be imported back into the same tool (or another instance). Many “version 1.0” tools indeed use this as their standard way of creating metamodels, although there is normally some special handwritten support in the tool for the modeling language in question. In tools with the power to achieve this without special tricks or hand coding, it provides a rich and fertile ground for investigating different ways of metamodeling: DSM applied back on itself.

14.4 CURRENT TOOLS

A complete review of current tools is beyond the scope of this book. Given the background of the authors, any attempt would be subject to accusations of bias—founded or unfounded! We will thus look only at what we consider to be the top four environments. As a criterion for this selection we require that the tools form a coherent DSM environment, including metamodeling, graphical modeling, and generation.

We will include two established DSM environments, MetaEdit+ and GME, and two “version 1.0” tools: the DSM plug-ins recently released by Microsoft and Eclipse for their IDEs. Within Eclipse there are also some other “version 1.0” or prototype plug-ins that we shall mention briefly.

14.4.1 MetaEdit+ (MetaCase), 1995–

First released in 1995, MetaEdit+ is a descendant of the earlier MetaEdit in terms of its concepts and development team, but a fresh start in terms of its code. MetaEdit+ aimed to rectify several architectural decisions in MetaEdit that proved to restrict its scalability and efficacy. Some of the decisions had simply been due to limited resources, but others were based on false assumptions: what had seemed a good idea turned out to work poorly in practice. Interestingly, these same assumptions appear to be present in the current crop of “version 1.0” tools: perhaps this is a phase all such tools must go through.

Early development of MetaEdit+ concentrated on laying a strong architectural foundation, based on the experiences with MetaEdit and building on earlier work in QuickSpec and PSL/PSA. Initial development took place in the MetaPHOR research group, which had been formed at the university in 1987. The company, MetaCase, was founded in 1991, and development efforts progressively shifted from the university to the company. This move was completed when the core developers finished their doctorates in 1998. While much has happened since then, the academic history of the early days does mean that there are around 30 theses and 150 articles documenting the initial versions of MetaEdit+ (e.g., Kelly et al., 1996) and the ideas behind it (e.g., Tolvanen, 1998).

MetaEdit+ is at level 6 in our scale of tool evolution: metamodeling and modeling take place in the same integrated environment, running as a single or multiuser system on any of today’s major platforms. With version 4.5 released in 2006, it fulfills all the criteria we have identified above as desirable for a DSM environment—time to identify some new requirements! Humor aside, given the greater time we have had to evolve MetaEdit+, it would be surprising if we had not been able to develop it to fulfill what we see as necessary from our experience and understanding. The more interesting question is in which areas are there holes in our experience or blind spots in our understanding.

According to independent industry experts, such as Scott Ambler (2006) and Markus Völter (2006), MetaEdit+ is the most mature and sophisticated DSM environment. A recent independent survey in Europe by MediaDev (2006) revealed that it is also the most widely known, recognized by over 50% of lead developers. With customer projects ranging in size up to several hundred modelers and over 10 years of duration, MetaEdit+ is also widely recognized as having the most experience of DSM in use. Even so, we still feel there is much to improve and learn: current DSM practice and tools will face many challenges and choices as adoption spreads.

14.4.2 GME (Vanderbilt University), 2000–

The Generic Modeling Environment (GME) grew like MetaEdit+ out of an extensive background of earlier research and practical experience in metamodeling and modeling tools. The earlier work with MultiGraph Architectures at Vanderbilt had proved the value of DSM, and created a framework for building new modeling tools. Specifying the metamodel in textual files proved time consuming and inhibited the

evolution of the modeling languages in use. Nordstrom's thesis (1999) outlined a meta-metamodel suitable for defining DSM languages in his domain of electrical engineering, and a Generic Modeling Environment that could be configured by files generated from metamodels made with this language.

GME's meta-metamodel differs significantly from that of other tools. It has a strong concept of port coupled with a weak concept of relationship, probably because of the electrical engineering inheritance: wires are just wires and connect specific pins on components. The tool support also reveals this same balance: all connections between objects are autorouted as uneditable horizontal and vertical lines. There is no separate first-class concept for graph, but objects are allowed to contain other objects to form a hierarchy.

Metamodels are specified in a simple UML-like language, which makes extensive use of stereotypes to distinguish the various metatypes—an odd choice in a DSM environment where different concepts should normally be given different symbols. Constraints are specified in a dialect of OCL. There is no graphical symbol editor and only a simple macro-based system for generating reports: all symbols and most generators are written by hand in C++.

GME is at level 5 in our scale of tool evolution: a metamodel is turned into a binary file, which configures a second instance of GME. Models do not update automatically when the metamodel is updated: an explicit operation can be chosen to try to update them, but this will often be unable to complete the update. A secondary scheme involves exporting the model as XML, upgrading the metamodel, and then reimporting it; even this will fail if the old model is not completely valid in the new metamodel. In that case, the user must return to the older version and correct the model by hand to be valid in both old and new versions of the metamodel—presumably impossible in some cases except by deleting parts of the model—and only then can she update.

GME is only available on Windows. At least earlier, it also had a multiuser version using an ODBC connection to a Microsoft SQL server back end. A database can contain multiple projects, each containing multiple models. The models must all be from the same set of modeling languages. Integration with other handwritten code modules for model analysis or extra tool functionality is via a COM interface, or a C++ interface built on top of COM.

14.4.3 DSL Tools (Microsoft), 2006–

In the Visual Studio 2005 SDK 3.0, Microsoft released the first version of their Domain-Specific Language Tools: a combination of frameworks, languages, editors, generators, and wizards that allow users to specify their own modeling languages and tools. This support for DSM forms the major new element of Microsoft's wider Software Factories project. Model-driven development has backing from the highest levels of Microsoft, with Bill Gates claiming the approach will be the most important innovation in software over the next 10 years (Seeley, 2004).

As its metamodeling and modeling tools will both only work as part of Visual Studio, DSL Tools are only available on Windows. Worse, the license agreement

(since amended) demanded that no code generated from DSM tools built with DSL Tools may ever be run on a non-Microsoft platform:

Code Generation and Optimization Tools. You and your end users may not use any code generation or optimization tools included in the [Visual Studio SDK] or Visual Studio (such as compilers, linkers, assemblers, runtime code generators, or code generating design and modeling tools) to create programs, object code, libraries, assemblies, or executables to run on a platform other than Microsoft Platforms.

From an early stage, Microsoft decided that MOF and UML were unsuitable for describing modeling languages, stating clear arguments similar to those we have discussed elsewhere. While this could be imagined to be a thrust against the IBM–Eclipse–OMG trinity, many of the leading DSL Tools figures were previously closely involved with UML and the OMG. Indeed, the core of the team seems to have been brought in from outside Microsoft: Jack Greenfield was a Chief Architect at Rational, Keith Short was CTO at Texas Instruments, Steve Cook was a Chief Architect at IBM.

The resulting meta-metamodel is not, however, all that different from MOF, and indeed shares many of its failings. There is no clear concept of multiple graphs, no n-ary relationships, roles and ports are not first-class elements, and reuse is hampered by the requirement that each element be a child of exactly one strict aggregation. Ironically, the first example model in the documentation shows a clear n-ary relationship, faked by carefully overlaying four binary relationships in exactly the same place.

On the positive side, the lines in diagrams are recognized as being roles: a line between two objects thus consists of a relationship at the midpoint, one role line to one object and another to the other object. The meta-metamodel clearly separates aggregation or containment from normal relationships. The latter are however stored as facts about the objects involved, effectively preventing transparent object reuse.

Metamodels are specified in a graphical version of this meta-metamodel. Oddly, there is no way to start from an empty metamodel: they are always created from a wizard where the user must choose the closest existing metamodel “template”, and even the smallest already contains concepts. Users must thus delete these or preferably rename them. In an unusual decision, the metamodel is always automatically laid out in a vertical tree. To maintain the pure tree structure, if an element is referred to from more than one place, it must be duplicated as a reference element to elsewhere in the tree. This structure quickly becomes unwieldy, and it is hard to see what benefits the designers expected from this decision.

The concrete syntax is severely limited: symbols may consist of only a single geometrical figure, and the only supported figures are rounded rectangles or their extreme versions, rectangles and ellipses. Bitmap or other external images can be used, but do not scale, and compartment shapes are used to handle aggregations. In a novel decision, the elements representing graphical symbols are drawn in a “sidebar” to the main metamodel, paralleling the elements they represent. Confusingly, the natural representation of these elements is not used in the metamodel: whether it is a red rectangle or a blue ellipse, the element is always shown as a gray rounded rectangle.

DSL Tools are at level 3 in our scale of tool evolution: the metamodel is transformed by generators into C# and possibly C++ code. This code is extended with handwritten code for constraints, proper symbols, and improved editor behavior. The result is then compiled, built and opened as the DSM tool in a separate debugging instance of Visual Studio. When the metamodel is changed, regenerated and recompiled, this instance must be closed and a new instance opened.

Currently, there is little support for updating a modeling language. If the name of a modeling language concept is changed, the corresponding model elements all disappear, effectively being deleted when the model is opened. Similarly, if a property on a relationship in the metamodel is changed, existing relationships may give errors or disappear.

Generation facilities consist of a text template language, mostly consisting of C# or Visual Basic that targets the generated data structures of the modeling language and tool. Templates can read from multiple models, but each template can only produce one file. The template language's support for modularizing templates is limited to simple include commands.

14.4.4 Eclipse Modeling Project (Eclipse), 2006–

The Eclipse Modeling Framework (EMF) allows the generation of simple tree views and property sheets from metamodels specified in XML files. The Graphical Editing Framework (GEF) is a framework that allows graphical editors to be written using Draw2D for display. EMF and GEF are not integrated and GEF in particular operates at a low level of abstraction, leading to serious inefficiencies if used to create a DSM tool. To address these concerns, the Graphical Modeling Framework was developed. GMF integrated and extended an improved GEF from IBM with GEF code generators from Borland. The underlying models thus remain in EMF, but GMF subsumes and encloses GEF to provide the representational information and diagram editor.

GMF 1.0 was released in June 2006, as part of the Callisto release of Eclipse projects. The current 1.0 release seems to be regarded as falling somewhat short of what was intended, because of the time constraints of simultaneous release with other Eclipse projects. The direction toward the next version is however correct, with getting it right being preferred over backward compatibility. According to GMF project lead Richard Gronback (2006), “We anticipate a 2.0 release for the next release rather than a 1.1 perhaps because [sic] API breakage that we expect.”

Creating a simple editor with GMF consists of building five different XML files, most of which can be created with graphical or form-based modeling tools:

- Domain model: abstract syntax, Ecore model
- Graphical definition: concrete syntax, XML or tree view and property sheets
- Tooling definition: palette tools and actions, tree view and property sheets
- Mapping definition: links first three models, tree view and property sheets
- Generation model: customize plug-in and UI, tree view and property sheets

The use of Ecore—effectively MOF—for the abstract syntax leads to the normal difficulties: relationships in the DSM language may be classes or associations in the metamodel, and associations in the metamodel may represent relationships, roles or properties in the DSM language. These ambiguities cause the need for the tooling and mapping definitions, which provide information missing from the Ecore model. Having the information spread across so many models leads to significant duplication of data and effort, with little increase in useful flexibility.

The current releases of EMF and GMF are at level 3 on our scale of tool evolution, at least for real-world DSM languages: level 4 is possible for simpler examples. Many of the identified required features are missing. Providing a more detailed listing for a “version 1.0” tool in a book seems little use: readers should check the state of the current version themselves.

No native code generation facilities are included, but the separate JET project offers simple textual templates. More powerful facilities for generation or model-to-model transformations can be provided through third party tools or frameworks such as OAW or ATL. As these operate on essentially any XML input, they are of course also usable with a wide array of other tools.

Other Eclipse-based tools exist, with the largest growth being in tools or frameworks that aim to raise the relatively low level of abstraction of GMF. Two prominent ones, TOPCASED and GEMS, appear to predate GMF, and to have originally been intended to perform the same function as GMF. They have now been retargeted to use GMF, although this should in theory have little effect on their users, if they already hid the low level nature of another graphical editor framework. TOPCASED adds little new and is only at version 0.3.0, so we will look at GEMS.

GEMS is an extension to GMF built by Jules White at Vanderbilt University. It aims to remove some of the low-level work of using GMF, and focuses particularly on adding constraint solving to the resulting editors. Constraints can be expressed in Prolog, giving better expressive power than Java or OCL, and can be used either to provide a simple check of a model or to add new elements to a model automatically, so that the constraint is fulfilled.

A commercial, closed-source Eclipse plug-in with a similar approach to GMF is XMF-Mosaic from Xactium. Again, multiple modeling languages are used to describe a DSM solution. In this case, there are also languages for specifying code generation: modeling language concepts are mapped to simple object-oriented programming concepts like classes or methods, and separate generic mappings are provided from these concepts to specific languages like Java.

14.5 SUMMARY

DSM environments are one of the most conceptually complex types of software. When made well, they can allow one of the greatest ranges of behavior for relatively simple input. Turning such complexity and power into apparent simplicity is however a difficult task. Building the foundation of such an environment without significant experience in DSM is possible, but such foundations rarely have the necessary

qualities of stability and flexibility. Lack of these qualities will be most visible as use of the environment grows in time and size. On the time dimension, the evolution of metamodels—and the parallel evolution of the environment itself—will be slowed and soon stopped by the weight of unknown and unintended consequences of changes. On the size dimension, the tool will face difficulties in scaling to support multiple models, modeling languages, and users.

Most DSM environment developers have consistently shown a disregard for earlier work. Of the tools whose first version was released in 2006, not one matches even the power and simplicity of GraphTalk, released in 1988, let alone that of mature tools such as MetaEdit+ and GME. Paradoxically, it was the success of UML that rehabilitated modeling after the disillusionment caused by the failure of CASE, and yet that same success was instrumental in the decline and fall of many promising metamodeling tools of the 1990s. Dying before Google and the Wayback Machine, their experience is effectively lost to current developers, even if they published widely in journals and conferences.

If there is one area where the new DSM environments must stop and rethink, it is their meta-metamodels. All are strongly influenced by UML Class Diagrams, a language for a domain completely different from that of graphical modeling languages. All suffer heavily from this mismatch. It would be ironic if just at the point where our industry has acknowledged that UML has failed as a way of building applications (as opposed to describing them), and is moving toward DSM as the best way forward, DSM environments were to blinker themselves into offering UML as a way of building modeling applications.

CHAPTER 15

DSM IN USE

Section 13.4.3 already looked briefly at some of the issues of the use of the Domain-Specific Modeling (DSM) solution from the point of view of the modelers. In this chapter, we will discuss in more detail those aspects of modeling that are not already covered by the decisions made in the DSM solution, and also look at some such decisions that can only really be made when wider issues of use are considered.

In many of these areas, DSM works rather differently from code-based development. Although this is to be expected, it often comes as a surprise to find that some accepted wisdom and truths held as self-evident were actually only valid in the context of code-based development. Some of the “version 1.0” DSM environments have not yet discovered these differences and try to store the new wine of DSM in the old wineskins of textual programming languages. In this chapter, we will assume an object-oriented approach to models and model storage and seek the best solutions for DSM: once we know where we are heading, we can better decide whether we want to make transitional concessions to existing practices.

15.1 MODEL REUSE

In many ways, the productivity of DSM can be attributed to various kinds of reuse. Existing expert experience is used in building the modeling language, generators and domain framework, and all of these are reused automatically each time a developer makes a model.

DSM is however particularly well suited to further reuse: existing models and model elements can be applied and referred to when making new models. Such reuse should generally aim to be by reference, not repetition.

15.1.1 Copy by Value Versus Copy by Reference

Traditionally, there has been a distinction between two methods of copying: copy by value and copy by reference. Although this will be familiar ground to most readers, let us lay down some basics first. We will say that there is an existing element, already used at one place in the system. The element has some content, and we want to use it in a new place too. We will keep the details deliberately vague: element, content, place, and system could map to many things, depending on our development process and artifacts.

In copy by value, the content of the existing element is copied to the new place. The content thus exists as two separate copies, and changes to one will not affect the other. In copy by reference, the content of the existing element is simply referred to from a new place. Any later changes to the element are thus visible in both places where it is used.

The advantages and viable use cases for these different methods should be clear, at least for this limited, abstract view. Taking a wider view and demanding more details reveal some extra complexities.

15.1.2 Names and Copy by Reference

First, how do we actually make a reference to the existing element? The traditional method from coding has been to give the element a name. More precisely, the name is a sequence of characters that serves to uniquely identify a particular element in the contexts in which the name is found. This introduces a distinction between the place where the element is defined and where it is used: the definition includes the name and the content, the uses include just the name, or to be precise, the same sequence of characters.

The content of the element thus remains pure and simple: it exists in one place, so everything just works. The name of the element however exists in two or more places in the system. If the name is changed in one place, the same change must be applied to all other places, if the references are to remain intact. Ideally, this support would be provided automatically. This can prove to be surprisingly difficult: there may be no easy way to find all such places; in more complex cases, it may even be practically impossible.

In copy by reference, we can always change the content of the element, affecting all places where it is used. We can also always change one or more uses to point to a different element, simply by using the new element's name instead. The loose coupling between the various occurrences of the name also gives rise to a new possibility. We can deliberately change the name in the definition of the element and create a new element whose definition includes the old name. All users of the old element are now automatically users of the new element instead. This kind of

indirection can prove useful and powerful, but it introduces a problem: when changing the name in the element definition, a developer must remember to specify whether she wants to change the name in the references too.

15.1.3 Copy by Reference in Models

In a modeling tool, we can use copy by reference via the names of modeling elements, i.e., particular string properties. The tool itself need offer no particular support: it is enough if the generated code is formed so that the references lead to the right elements in the code.

In the dirtiest approach, it is even possible for copy by reference in a model to map to copy by value in the code: the content is included many times in the code. This is similar to the inlining performed by optimizing compilers: program size is increasing, but speed may also increase. Although any duplication like this should ring warning bells in developer's minds, the dangers here are significantly less than if such code were handwritten. As the code will not be edited by hand, we avoid the problems of copy by value such as loss of synchronization.

Using copy by reference in this way in a modeling tool may however be simply perpetuating a poor solution. In textual program languages, a name was the best available way to refer to an existing element. The atomic elements that programmers and compilers worked with were characters, and the next largest element was the file. In a modeling tool, we have at least characters, properties, objects, and graphs. Each of these has its own unique identity, regardless of its contents or any part of that content we consider its name. When a modeler reuses an element in a model, he most likely simply selects the existing element directly. The reference can thus easily and permanently be made to be to that particular element.

Modeling tools thus allow an even purer form of copy by reference: we can call this copy by direct reference. Since modeling tools also support the older form, we can call that copy by name reference. As the direct references remove many of the problems of name references, most reuse will happen that way.

The only benefit of name references over direct references is the extra level of indirection provided by the name. We will later see a situation in which we can take advantage of that, but for most uses equivalent results can be achieved by direct reference, without introducing the problems of name references. We can simply add an extra model element between the referrers and the reused element. The references are directly to this new element, which in turn refers directly to the reused element. At any stage, the new element can be altered to refer to a different element, thus affecting all of its references.

15.1.4 Copy by Value in Models

When programmers copy an element of code by value, that is, by copying and pasting it, it is immediately apparent which atomic elements have been copied: the particular characters of the selection. These characters are however later interpreted by the compiler, adding semantics above the level of characters. Some of the copied text

will map to individual primitive commands, which will be copied by value. Other parts will map to references by name to other functions, and while the references will be copied, the content of the functions will not. Copy by value thus always includes the idea that the copy is to a certain depth, at least if the copied content also makes use of copy by reference.

When copying by value in a model, there is a greater range of choices. Model elements link to other elements in a wide variety of ways: relationships, shared properties, objects as properties, subgraphs, and so on. The exact set of link types and their semantics will be determined by the tool's meta-metamodel. The metamodel may specify additional information about some of these links, for example, an object might be held in a property as a simple reference or as a strict aggregation. This information can be used to make choices about which subelements should also be copied and in which cases the copy should share the same subelement as the original.

However, even with this information from the meta-metamodel and metamodel, there will often be choices that could be left to the modeler. The choice to use copy by value rather than copy by reference tends to be a concession to pragmatic needs, and so too with the choice to copy deeper than normal. For these cases it is useful if a tool offers the ability to choose what kinds of links to follow and how deeply.

One possible cause for deep copying is ad hoc, opportunistic time saving: an element to be created may be similar to an existing element. Although many kinds of similarities like this can be factored out into their own variation points, allowing a simple reference or higher level object to make the difference, there will always be cases that the modeling language cannot handle perfectly. Even if the modeling language were perfect, there would still be times when two model fragments would be similar in a number of respects and different in others, but purely by chance rather than any underlying relationship between them.

Another possible cause is a factor outside of the scope of the DSM solution, for example, a business agreement or a divergent branch of development. A model for one client may be similar to that required for a subsequent client, yet the agreement specifies that the models must also be editable by the client. If the development for the previous client is not frozen, the second client obviously cannot be allowed to edit the same set of models. The models must therefore be copied deeply, at least for those that will be passed to the second client. Even if the client does not request the right to edit the models, a project-based organization will often prefer to keep the two projects disjoint. Although this is not ideal, the situation is still significantly better than with code-based development: the modeling language and generators ensure that the majority of commonalities are shared by all from one central source.

15.2 MODEL SHARING AND SPLITTING

So far, we have largely considered one set of models in one instance of a tool. This seems to be the theoretical ideal for DSM, taking the greatest advantage of the possibilities for model reuse. These advantages can and should be maintained when

scaling to multiple users. This has been demonstrably achieved through multiuser repositories, but not yet for disjoint files.

15.2.1 Disjoint Files

Splitting a set of models into individual files, one per model, may seem like the most natural approach. In the unlikely event that the models are completely disjoint, in terms of both references and semantics, this may be a viable choice. Even so, there are still several decisions to be made:

- Does a file hold one graph or several (e.g., its subgraphs)?
- Does the file also hold the representational information (e.g., the diagram)?
- Can a file hold graphs of more than one modeling language?

In the more likely case that the models contain references, there needs to be some way of expressing those references. For such a reference to work, it needs to express three things:

- (1) A unique reference to the file in which the other element is saved.
- (2) A unique reference to the element within that file.
- (3) A unique reference to a particular version of that file (optional).

Since the references are made to an element within a file, rather than to the element in memory, we must also ensure that the version on disk is the same as that in memory. Otherwise, we may refer to an object that has been created since the last save but deleted before the next save. In practice, this means that to make a reference, or at least to save a file containing a reference, we must also save any changed files for all other models whose elements it refers to. Similarly, we must also make sure that these files are placed immediately into version control: otherwise even though an element made it onto disk, it may never have made it into the version control system. If the referring file did then make it into version control, the version there would be inconsistent.

The reference to the element must be by some automatically generated unique identifier. In a textual programming language, it is clear to all developers which names may not be changed. In a graphical model, users are used to being able to change names freely and have other elements cope with the change. If the referring elements are within this model, that would indeed be true, but for elements outside it cannot be. Using a GUID or similar approach will allow users to change names freely where that is desired. Where it is not, the metamodel can use reuse by name reference anyway, and the file will contain no explicit link.

Using a GUID may also reduce the need to store the target file's version information in the link. If a simpler counter index is used, there is the risk that another version of the same file could contain the same index for a different object. Saving the maximum value of the counter so far in the file does not help: two branches from a file where the counter is 10 will both call their respective next elements 11, even though

they have nothing in common. A reference to 11 can thus later be misinterpreted to point to the wrong element.

When sharing these files, we must also examine what makes a good unique identifier for the file itself. It is probably safest to use the unique path or internal identifier from the version control system. It may well be wise to include this information within the file itself to provide an extra measure of security.

Whatever the official process, if you give developers text files some of them are guaranteed to try to hack them in some way by copying and pasting content between them or editing them. Since the chance of them being able to do this correctly when links are involved is vanishingly small, and the consequences of getting it wrong both large and possibly not noticed until later, links in disjoint files will always present a risk.

Even if developers can be prevented from editing the files (uninstall Notepad?!?) and the version control system pressed into service as a makeshift repository, there is still no guarantee that links will be in a consistent state. One reason is that version control systems do not protect the version that is the target of links. Your model A may link to an object X in model B version 1.0, on the same day as the user responsible for B has it checked out for changes. She may delete X and save B as version 1.1. This is bad enough, as X will disappear completely when you next look at your model (assuming your tool loads updates from other models—if it does not, there really is no hope of ever making a coherent build).

Unlike with textual programming languages, you will not even see the name of the object you referred to (presuming the tool can even open the file). If the tool has saved the version information of the link, you will at least know which version of B to look in. The tool might even theoretically be able to fetch a shadow of version 1.0 of B and somehow handle having both 1.0 and 1.1 in memory. Alternatively, links could always save at least some basic content information about their targets, for example, their name and type.

It does not take a particularly evil mind to imagine that your colleague might also have chosen to refer to one of your objects. Her model 1.1 may thus refer to your model 1.0, and your model 1.1 to her model 1.0: a cyclic dependency. While these are all perfectly normal problems with textual languages, trying to correct them at a textual level for models is something of a nightmare. The only real solution is that the tool be able to handle these issues, but current evidence shows this to be too much to expect: sadly, it is the “version 1.0” tools that use disjoint files. While some do not yet offer links between files, most of these say they are considering it.

For now, then, if you are stuck in a tool that offers intermodel links between disjoint files, but not the tool support for protecting you from the problems this may cause, our advice would be to avoid reuse by direct reference across model boundaries. This is a shame, as such reuse, when supported by a true repository, is something we have used to good effect in almost all cases of DSM we have worked on.

15.2.2 Multiuser Repositories

In a multiuser repository, all models are available in one coherent space. Models and their elements can be freely browsed and reused by direct reference. User permissions

and modeling language rules can be layered on top of this basic freedom to limit it where necessary. The repository and tool take care of the consistency of links between model elements. Elements are not deleted from the repository while others refer to them, so situations such as those in the previous section are avoided. In most cases, referring to an object that has been removed from another model can still generate good code. If reuse should only be allowed from a particular collection of objects, it is easy to add constraints or generator-based checks to ensure that this is so.

When multiple users are simultaneously logged in, the repository and tool also take care of locking issues, preventing one user's changes being overwritten by another user who has not seen them. As we saw in Section 14.3.5, the level of granularity of such locks should be as fine as possible to allow the maximum permeability and freedom of use. Changing an element should not prevent others from seeing the most recent committed version of that element: developers are used to working with the latest published version of interfaces or components.

Multiuser repositories are found in MetaEdit+ and GME, and at least MetaEdit+ fulfills these requirements (lack of experience with large-scale use of GME prevents us from talking about it in more detail). Both tools also have a single-user version, and both make the difference between the versions largely invisible to users. This indeed is a useful goal: it should be possible for a user to work on his area of a shared project mostly as if he were the only user. Making this possible requires sensible decisions in the meta-metamodel, the metamodel, and the models.

Software development has found modularization to be a powerful tool, and models too should be separated along lines based not only on functionality but also on developer responsibility. The modeling language and process should be designed to make this possible. With a DSM environment that has taken multiuser access into account in its meta-metamodel, achieving this tends to be relatively easy: the hard work has already been done. Any issues caused by the particular metamodel made for the modeling language should be revealed by the pilot project.

As with most multiuser databases, the repository will probably be stored physically as a set of files on a server. Although there are multiple files, the division into files is determined more by internal details of the tool, repository, or operating system than by any perceived semantic borders in the models. The files are thus treated as forming a single conceptual whole.

15.2.3 Multiple Repositories

For the smallest projects, or areas where only a little simple reuse of whole models is used, disjoint files may prove suitable. For larger projects, or where more reuse is required between models in the project, multiuser repositories are ideal. For the largest projects, or large projects with disconnected teams, the project can be split into disjoint multiuser repositories.

In the simplest case, each repository only contains the models made by its team. Each team works on its models, releasing the results as code and documentation to a central version control system, from where they are replicated to other teams. The interface between teams should be slight in this case, as it will be limited to the legacy

approach of design documentation and code. (Admittedly, the situation is thus no worse than it has been earlier, but it will certainly appear worse when it sits next to the shiny new DSM solution!)

In a more complicated case, there may be a set of core models that all users should have access to. These can be maintained in a separate repository and periodically exported and distributed to all satellite repositories. There they are imported, making them available for transparent access in those repositories. Since that access will almost certainly include reusing and referring to elements from those models, a simple import and export will not suffice.

When an updated version of the core models is reimported to the satellite repositories, the existing elements must be updated in place, so that any references from the local models to the core models will now point to the updated elements. Based on our experiences with a system like this, which has been in MetaEdit+ since 1997, such an approach works well, provided it is used as part of a defined process. Uncontrolled import and export in all directions is more likely to lead to a mess than to the miracle of making disconnected repositories appear connected, by automatically merging their differences into a cohesive whole.

15.3 MODEL VERSIONING

In moving from a single-user, separate-file approach to a multiuser repository, many practices need to be reexamined. A repository allows true reuse of elements across multiple models and multiple users. The smallest actionable unit is thus no longer a file—often corresponding to a model—but an object or even property in a model. Locking and versioning thus move away from the compromises required by files toward new solutions.

Looking at the origins of file-based version control from the days of C and FORTRAN, the real units of reuse tended to be functions within a file, rather than the file itself. The units of reuse do not directly correspond to the larger units of versioning or locking.

With object-oriented languages, we tend to think of the class as the unit of reuse, and often a class corresponds to a file. However, little actual reuse takes place at the level of the class itself: normally, just a single instantiation operation refers to it. Instead, the majority of references are to the operations and attributes within the class, so the situation is largely unchanged from early version control systems.

In models in a repository, the units of reuse correspond directly to the units of locking. In general, however, they do not correspond to visible units of versioning: the repository always reflects the current state of the whole project. An explicit version is made as a snapshot of the status of the whole repository. The difference in scale between the smallest units of reuse and the units of versioning is thus larger than with textual programming languages.

Are there any benefits to be gained by having these units aligned, or any disadvantages if they are not? Aligning reuse with locking seems useful: nobody can change the thing we are referring to. This removes problems similar to those at the end

of Section 15.2.1. Having such a large unit of versioning however tends to give cold shivers to most programmers used to source file version control. File-based versioning has been the way they have accomplished many tasks, and so the lack of it makes working with models seemingly impossible. Below we will look at the most common questions in the context of a multiuser repository: where repositories are single user or models are stored in files, little changes from traditional practices.

15.3.1 But How Do I Just Save My Work?

Normal development in file-based version control consists of checking out your own file or files for changes, then working with them for a while. If the results are satisfactory, you publish the changed files back to the version control system. If the results show this was a bad idea, you throw away the changes and release the checked out files. If you want to try again, you revert your checked out files to the versions in version control (possibly also updating to newer versions of other files).

With a repository, you open the models you want to work on. As you modify the models or elements, or alternatively when you open them for editing, you are given the lock on those models so that nobody else can change them. If the results of your editing are satisfactory, you commit your transaction. This publishes your changes to the repository, so other users can see them. It also updates your view of the repository to include any changes committed by other users. If these changes were a bad idea, or you want to try again, you abandon your transaction. This discards the changes, and as with commit it also updates your view of the repository.

15.3.2 But How Do I Branch?

Branching is used in code-based development to handle variants. Normally, many files will need to be branched to support a pair of variants, and each of those files must proceed in two parallel version branches. Because of the large costs associated with this, branching can only be used for a small number of variants.

With DSM, most situations that earlier required branching no longer occur. The modeling language factors out these changes, sometimes into the generator or domain framework and sometimes into small property or other changes in a top-level configuration model. DSM can thus support a much greater range of variants and expresses the variants in the simplest way for the modelers.

Even in the worst case, the branching is relatively painless: shallow copies are made of the relevant models, all referring to the exact same elements as before. The elements that must be changed are replaced individually by new elements (perhaps shallow copies for minor changes). A new top-level configuration model brings together the set of models needed for the variant. The variants are thus made explicit in the models and the coherent view of the project includes these variants, rather than only ever seeing one variant at a time. This broader view helps keep variants working when changes are made elsewhere, and also makes it easier to spot ways in which this method of handling variants could be improved towards one that requires less work.

15.3.3 But How Do I Merge Parallel Changes Made to the Same Model?

Deciding on the boundaries between files is an important task in code-based version control. The project architect will try to avoid the situation where the same file contains parts that will be worked on by multiple people. Always, however, cases like this will occur, even if the version control system uses some kind of locking to prevent them. Since the files are just text, developers can move them around and access old copies outside of version control. Merging multiple versions of the same file into one version containing all changes is a difficult task and has given rise to a whole mini-industry of diff and merge tools. None of these is as intuitive and informative as developers would like, but almost all of them are better than attempting the same task manually.

With a model repository, this problem simply does not occur: elements are locked when changed and no other user can change them until he has loaded the committed version of the user who obtained the lock. While normal modularization is good practice in DSM too, in a good tool it is perfectly possible for several users to edit different objects in the same model at the same time: each object is locked separately, so everyone can obtain the lock on just the object they need.

15.3.4 Where Do the Version Comments Go?

Version comments are an important part of development. In our experience, we refer frequently to version comments in both code- and model-based development. They provide information about why something changed, rather than just what changed, and allow a much quicker overview than even the best diff tool.

In a version of a multiuser repository, the number of changes made by all developers across all models will be too large and too disparate for some of the normal uses of version comments. Instead, it is better practice to keep version comments as part of the models, either as a visible object or as a property of the model. This is of course a practice familiar to many from programming, and brings the added benefit that the version history remains part of the model, even after export and import to another repository. The individual model version comments since the last repository version can easily be collected by a generator as the comment for the version of the whole repository.

15.3.5 How Does It Work in Practice?

Our experience is that versioning has simply been a nonissue in all the projects we have worked on. Of the material in this chapter, customers ask far more questions about how to build modeling languages and models effectively to allow reuse. Most customers have simply zipped up the repository for storage in version control, as version control systems generally require a versionable unit to be presented as a single file.

Where customers have decided to use a multiuser repository, the only issues encountered were technical ones during installation, for example, how to help

customers correct name server misconfigurations within their organization so that client computers could communicate with the repository server. Where customers opted for single-user repositories, a single feature, application, or project was assigned to a single developer and stored in a single repository. In a few cases, an individual repository grew too large to be managed by a single developer, and so those cases simply moved to using multiuser access to that repository when extra developers were added.

15.4 SUMMARY

DSM already offers large amounts of reuse and productivity through the work done in the modeling language, generators, and domain framework. The amount of reuse can be increased still further by reusing models and model elements. Storing models in a multiuser repository allows the greatest scope for reuse, makes it easiest, and avoids many of the problems familiar from reuse by name reference in code-based development. Where necessary or desired, work can also be split over several repositories, each used by a smaller team, or even one or more repositories per developer. These more disjoint approaches inhibit reuse and make the process of using DSM more like traditional programming. While this may not be a good fit, there may be some transitional benefit from the more familiar process.

Attempting to force DSM into the file-based model, with multiple files per developer and a single model per file, appears to be a poor solution, at least with current tools. The main expected benefit, better integration with code-oriented version control systems and practices, seems not to be achieved if any reuse is allowed.

Having large repositories as a unit of versioning was expected by some to lead to problems: previous code-based development needed version control systems with small units of versioning to solve some of its common problem situations. Experience shows that with DSM those problems either do not occur or are solved more effectively here in other ways.

CHAPTER 16

CONCLUSION

Domain-Specific Modeling (DSM) is nothing new. Rather, it gathers together a number of existing techniques into a cohesive whole. Some of the techniques, like component frameworks, are well known and widely used. Others, like creating your own graphical languages, have previously been used by a much smaller number of people. Many of these techniques have been hyped to solve the software crisis; used singly, all have failed. The task has been too large and the problem too complex for any one silver bullet.

The interesting thing about silver bullets is that they are not available off the shelf (Gray, 2005):

Like darning socks, making bullets is a dying art. Used to be just about everyone with a need for ammo poured their own, using iron or even wooden molds. These days only a few diehard hobbyists still do it, and they use aluminum molds. But even fewer people still make silver bullets.

Actually, not many people ever made silver bullets . . . At 1,764°F, molten silver would ruin traditional and modern bullet molds. They could have been fashioned using jewelers' methods, but that would require a new plaster mold for every bullet. Frankly, I think people spent a lot more time talking about silver bullets than they did turning them out.

DSM solutions too will generally require building for your specific situation. There are sound economic reasons for this. Assume for a minute that you are lucky enough

that some kind vendor has already built a DSM tool that is perfect for your case. Unless your exact problem happens to be identical to one shared by a number of companies, the vendor will find itself with a customer base of one. You can be certain that the next version of their tool will generalize to be applicable to a wider market—and hence no longer be such a perfect fit for you. Even if you stick with the first version, you will find your own problem and solution domains evolve, again spoiling the fit. These problems will be familiar to anybody who has bought clothes: “one-size-fits-all” doesn’t, and many clothes that fit 10 years ago no longer seem so fitting in either style or size.

16.1 NO SWEAT SHOPS—BUT NO FRITZ LANG'S METROPOLIS EITHER

When the authors wanted to get some shirts with our company logo, there were of course no shops that already carried such products. Paying seamstresses to embroider each shirt by hand would have been one approach, and indeed the common one a few decades ago. Looking at software written by a given team today, there are a number of similarities with that approach. Building software is labor intensive and uses generic tools. We may have progressed from the needle and thread of Assembler to the sewing machines of 3GLs and IDEs, but the tools are largely similar across all developers. Despite the similar tools, the code written across all projects will show a huge amount of variation. However, the code written in this one project should recognizably belong together. The company coding standards, component libraries, shared architecture, and in-house style guide should lead to the work of each developer all bearing the “mark” of this company and team. If it does not, the situation would be similar to each seamstress embroidering a different version of the company logo.

This shared set of properties, or commonality, is what DSM can leverage to improve productivity. Most organizations will already devote efforts to codifying and sharing this information, and some may automate parts of the process. Further automation has generally been prevented by lack of time or insufficient payback. Similar problems faced the embroiderers: they could have built a machine that would just embroider our logo, but we needed our order by the next week, and the cost of building the machine would be higher than the size of the order.

Fortunately, the shop we went to had a configurable embroidery machine, into which the expert embroiderer could specify our logo, and which was then able to produce that logo automatically and reliably. Entering the logo into the machine required a new set of skills and a start-up cost. Once entered, anyone in the shop could run the machine to sew the logo on different material, in different colors, rotated, scaled, stretched, or duplicated in an endless pattern. These differences are analogous to the variability between different pieces of code from the same team, although clearly in software there is far more variation per piece of code in the same project than there is per sewing of the same logo in embroidery. The basic situation was however the same: for all the stakeholders, this configurable automation approach represented a clear saving over the other choices.

Clearly, the expert embroiderer who could specify a new logo to the machine had an important role. But what of the other workers who would use the machine then and on subsequent occasions to embroider the logo onto shirts, caps, and so on? Have they become de-skilled like the workers in Fritz Lang's Metropolis, little more than automatons pushing the buttons and pulling the levers of a machine? Far from it! Their skills are still very much in demand, working with the customer to choose the right material, color, size, and placement of the logo. If people with no aesthetic skills were let loose with the machine, all we would have would be the ability to produce terrible-looking clothes very quickly.

The important role of design is thus still present: only the tedious, time-consuming, and error-prone task of turning that design into finished product is automated. The same is true for the modelers in DSM: if we let people with no abstract thinking or analytical skills loose with a DSM tool, the code produced may be notionally bug-free thanks to the modeling language rules and the generator, but the resulting system will be unlikely to do anything useful.

The idea of DSM is thus not to replace developers with generators, but to enable those same developers to produce the same systems faster, more reliably and with higher quality. That after all is the problem of the software crisis: our projects are late (or fail) and contain too many bugs. DSM brings together a number of techniques to address that problem in a way that has proved repeatable across a wide variety of problem and solution domains.

16.2 THE ONWARD MARCH OF DSM

Unlike recent technologies such as Java or UML, DSM is not something that a vendor can market and push through to adoption with their tools. With those technologies, a gatekeeper in each organization had to be persuaded to adopt, and the job was done. With DSM, pure persuasion is not enough: the organization also needs to find someone and give them the task and resources to create a new modeling language and generator.

For many readers, you will be in the role of that gatekeeper or DSM solution creator. You are perhaps sufficiently convinced by the logic behind DSM to accept that it could work, but unsure if it will work for you. We hope that the material in this book will go some way to helping you feel confident enough to take a stab at trying out DSM in your organization. The gains are worth it.

For the first time in decades, our industry is presented with a way of development that has been consistently shown to increase productivity several times above the current norm. The people who can make the change happen are not the tool vendors, nor the managers, nor the mass of developers, although all these have a role to play. Instead, the choice is with the people who are best placed to make it, the expert developers. The task ahead of you is challenging but worthwhile.

Of course, as a busy lead developer you don't have time for it. But every time you explain something to another developer, or correct their bug, or remind them to look at the in-house style guide—or even just realize why you don't have time—remember that there is a better way.

APPENDIX A

METAMODELING LANGUAGE

A metamodeling language is used to specify the abstract syntax of modeling languages. In this appendix we describe the main concepts of the metamodeling language used in this book. For some background on describing modeling languages, see Section 4.2.1. To understand the basic principles of metamodeling, see Section 4.2.4. More details and some more advanced topics on metamodeling can be found in Sections 10.3–10.5.

Throughout the book, we have consciously avoided limiting the discussion to a particular metamodeling language or tool, but when discussing concrete cases or details of a particular technique it is important to have a precise, consistent set of concepts. In the example cases in Part III, and occasionally in the discussions in Part IV, we therefore apply a widely used metamodeling language, GOPRR.

Although more familiar languages from other domains can also capture some of the information necessary in metamodels, GOPRR is preferred here because it was specifically designed for describing modeling languages. Using a domain-specific modeling language for the task of metamodeling brings the same benefits as the use of DSM in any domain: simplicity, precision, and automation. Languages like UML, MOF, and ER were intended for other domains, entailing greater effort, loss of precision, and risk of misunderstanding for the reader. These issues are discussed further in Section 14.3.1, which looks at what is needed in a metamodeling language.

Before looking at metamodels, we will start with an example model of a simple order handling system (Fig. A.1).

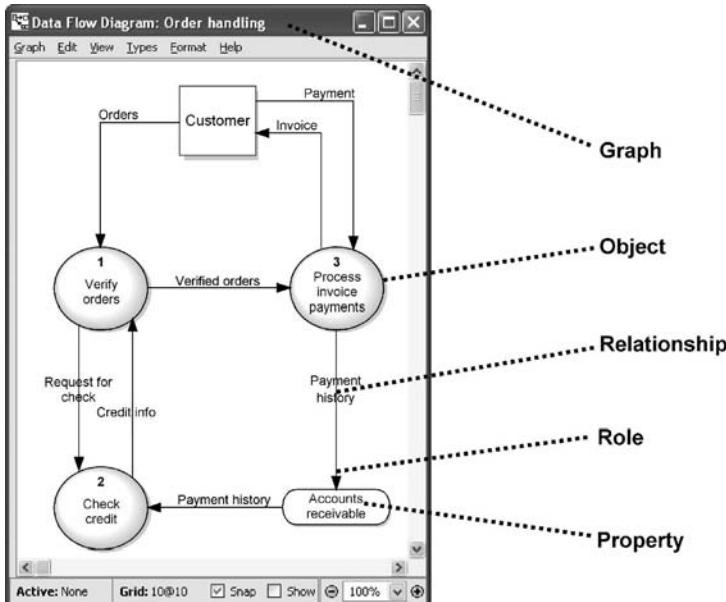


FIGURE A.1 A data flow diagram of order handling system (partial)

This diagram describes the flow of data when handling orders and invoices. On the right of the figure, we have labeled some examples of the basic constructs that can be found in all models, whatever their language.

- A graph is one individual model, often shown as a diagram
- Objects are the main elements of graphs, often shown as boxes or circles
- A relationship connects objects together, often shown as a label over the connection
- A role connects an object into a relationship, shown as a line and often an arrow-head
- A property is an attribute characterizing one of the above, often shown as a label

TABLE A.1 Instances, Types, and Metatypes

Instance	Type	Metatype
Order handling	Data flow diagram	Graph
Process invoice payments, customer	Process, external	Object
Payment history, verified orders	Data flow	Relationship
Arrow-head at accounts receivable	From, To	Role
“Process invoice payments”, “3”	Process name, Process number	Property

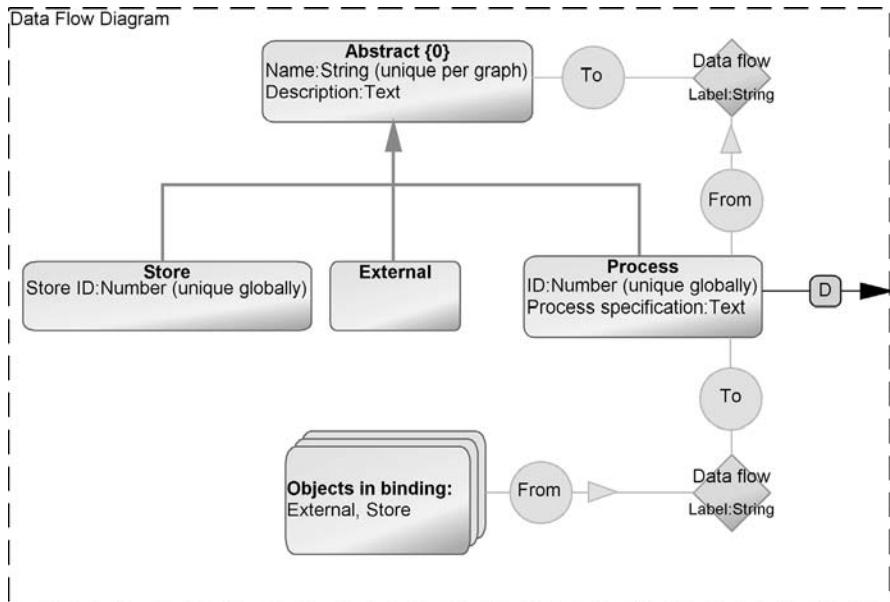


FIGURE A.2 A metamodel of data flow diagrams

We will call these basic constructs metatypes. Table A.1 lists some example instance elements of this model, their types, and their metatypes.

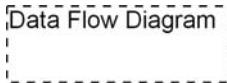
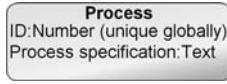
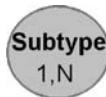
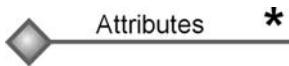
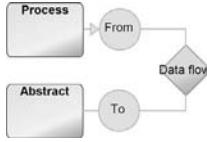
A language usually has additional constructs that are not immediately visible from one example model. In our example, a process may contain subprocesses that are specified in another data flow diagram. A language also has a number of constraints that ensure model correctness and consistency, such as:

- Processes can be connected with data flows to processes, externals, and stores
- Stores and externals can be connected to processes, but not to stores or externals
- Each data flow connects two objects, with one from role and one to role
- Each process must have a process number, unique over all diagrams
- All processes in a diagram must have different process names

A metamodeling language can describe these constraints along with the modeling constructs. Figure A.2 illustrates the metamodel for data flow diagrams.

The metamodel is specified with GOPRR, a domain-specific language for specifying modeling languages. From a metamodel like this, modeling tool support can be created automatically without additional development tasks. Table A.2 shows the concepts and notation of GOPRR; as a shorthand, the names omit the word “type”: an “Object” here is actually an object type.

TABLE A.2 GOPRR Metamodeling Concepts

Metamodeling concept	Notation
Graph specifies one modeling language, such as data flow diagram and use case diagram. A graph contains objects, relationships, and roles.	
Object describes the basic concepts of a modeling language. Objects are the elements that you connect together and often reuse, such as process, state, and actor.	
Relationship defines properties for the objects' connections, such as data flow, inheritance, call, and transition. They are used to form bindings with objects and roles.	 
Role specifies the lines and end points of relationships, like the Subtype part of the inheritance relationship and the From part of the data flow relationship.	
Property defines the attributes which characterize any of the previously mentioned language concepts. Properties can be of different data types (string, text, number, Boolean, collection, etc.) and link to other modeling language concepts or to external sources, such as files, programs, or web services. Examples of properties are process name, multiplicity, and data type.	Simple properties are defined inline in other concepts. Properties whose values are objects are shown like this: 
Binding connects a relationship, two or more roles, and for each role, one or more objects in a graph. Binding is further specified with multiplicity.	
Object Set describes a collection of objects that can play the same role in a binding, for instance, External and Store can both be in the From role in a data flow relationship.	
Inheritance allows creating subtypes of other language concepts, for instance, External is a subtype of Abstract.	
Decomposition allows objects to have subgraphs, for instance a process can decompose to another data flow diagram.	
Explosion allows objects, relationships, or roles to be linked to other graphs, for instance, the detailed structure of a Store in a data flow diagram may be specified in an entity relationship diagram.	

REFERENCES

- Albrecht, A.J., Gaffney, J.E., Jr, Software function, source lines of code, and development effort prediction: a software science validation, IEEE T Software Eng. 9, 6, 1983.
- Alderson, A., Meta-CASE technology, in: Endres, A., Weber, H. (Ed.), Software Development Environments and CASE Technology, Proceedings of European Symposium, Königswinter, June 17–19, No. 509, Springer-Verlag; Berlin, 1991, pp. 81–91.
- Alderson, A., Experience of Bi-Lateral Technology Transfer Projects, 2nd IFIP WG8.6 Working Conference: Diffusion, Transfer and Implementation of Information Technology, Technical Report SOCTR/97/04, Staffordshire University, 1997, <http://www.soc.staffs.ac.uk/reports/soctr9704.html>.
- Alford, M., A Requirements Engineering Methodology for Real Time Processing Requirements, IEEE T Software Eng. 3 (1), 60–69, 1977.
- Amber, S., Agile Modeling Newsletters, Software Development Magazine, March, 2006.
- Arango, G., Domain analysis methods, Software Reusability, Chichester, England, Ellis Horwood, 1994.
- AUTOSAR, AUTOSAR methodology, v. 1.0.1, 2006, http://www.autosar.org/download/AUTOSAR_Methodology.pdf.
- Avison, D.E., Fitzgerald, G., Information systems development current themes and future directions, Inform. Software Technol. 30 (8), 458–466, 1988.
- Babin, S. Developing Software for Symbian OS: An Introduction to Creating Smartphone Applications in C++, John Wiley & Sons Ltd., 2005.
- Batani, C., Lenzerini, M., Navathe, B., Conceptual database design: An entity relationship approach. Benjamin-Cummings Publishing Company, 1992.

- Batory, D., Chen, G., Robertson, E., Wang, T., Design wizards and visual programming environments for GenVoca generators, *IEEE T Software Eng.* 26, 5, 2000.
- Bell, A., Death by UML Fever, *ACM Queue*, Vol 2, no. 1, 2004.
- Bettin, J., Measuring the potential of domain-specific modeling techniques, Proceedings of the Second Domain Specific Modeling Languages Workshop, Helsinki School of Economics, Working Paper series, 2002.
- Bézivin, J., Ploquin, N., Tooling the MDA framework: a new software maintenance and evolution scheme proposal, *Application Development Trends*, 2001.
- Bhanot, V., Paniscotti, D., Roman, A., Trask, B., Using domain-specific, modeling to develop software defined radio components and applications, Proceedings of the 5th OOPSLA Workshop on Domain-Specific Modeling (DSM'05), in: Tolvanen, J.-P., Sprinkle, J., Rossi, M., (Eds.), *Computer Science and Information System Reports*, Technical Reports, TR-36, University of Jyväskylä, Finland, 2005.
- Blackwell, A., Metaphor in diagrams, Ph.D. Thesis, University of Cambridge, September 1998.
- Booch, G., Rumbaugh, J., Unified Method for Object-Oriented Development, Documentation Set, Version 0.8, Rational Software Corporation, 1995.
- Brinkkemper, S., Formalisation of Information Systems Modelling, Thesis Publishers, Amsterdam, 1990.
- Brooks, F., *The Mythical Man-Month*, Addison-Wesley, 1975.
- Bubenko, J.A., A Method Engineering Approach to Information Systems Development, The Proceedings of the IFI P WG8.1 Working Conference on Information Systems Development Process, 1988, pp. 167–186.
- Bubenko, J.A., Lange fors, B., Sølvberg, A., Computer-Aided Information Systems Analysis and Design, Studentlitteratur, Nordforsk, Lund, 1971.
- Buede, D.M., *The Engineering Design of Systems*, Wiley, 1999.
- Caine, S.H., Gordon, E.K., PDL—A tool for software design, Proceedings of the National Computer Conference, AFIPS Press, 1975.
- Chen, P.P., The Entity-Relationship Model: Toward a Unified View of Data, *ACM T Database Sys.* 1 (1), 9–36, 1976.
- Compuware, Proving Productivity—An Independent J2EE Development Study, 2003, <http://www.compuware.no/optimalJ/Middleware1.pdf> accessed January 2007.
- Costagliola, G., De Lucia, A., Orefice, S., Polese, G., A classification framework to support the design of visual languages, *J. Visual Lang. Comput.* 13 (6), 573–600, 2002.
- Czarnecki, K., Generative software development, Invited talk at Seventh International Conference on UML Modeling Languages and Applications, October 10–15, 2004, Lisbon, Portugal, 2004.
- Czarnecki, K., Helsen, S., Classification of Model Transformation Approaches, Online Proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA, Anaheim, 2003.
- Dubinsky, Y., Hartman, A., Keren, M., D5.3 Industrial ROI, Assessment, and Feedback, ModelWare Report No. 511731, 2006.
- Duffy, D., *Domain Architectures*, Wiley, 2004.
- Engstrom, E., Jonathan, K., Building and rapidly evolving domain-specific tools with DOME, IEEE International Symposium on Computer-Aided Control Systems Design, Anchorage,

- AL, USA, 2000, <http://moncs.cs.mcgill.ca/people/mosterman/campam/cca01/cacs00a/index.html/ek.pdf>.
- Fayad, M., Johnson, R., Domain-Specific Application Frameworks: Frameworks Experience by Industry, Wiley, 1999.
- Ferguson, R.I., The beginner's guide to IPSYS TBK, University of Sunderland Occasional Paper 93/3, 1993.
- Fitzgerald, G., Validating new information system techniques, in: H.E. Nissen., H.K. Klein., R. Hirschheim. (Eds.), Information Systems Research: Contemporary Approaches and Emergent Traditions, Elsevier Science Publishers, 1991.
- Gane, C., Sarson, T., Structured Systems Analysis: Tools and Techniques, Prentice Hall, Englewood Cliffs, NJ, 1979.
- Goldkuhl, G., Stefan, C., Customizable CASE Environments: A Framework for Design and Evaluation, Linköping University, Sweden, 1993.
- Gray, T., Van Helsing, C., Popular Science, 07/ 2005.
- Greenfield, J., Short, K., Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley, 2004.
- Griss, M., Wentzel, K., Hybrid Domain-specific kits for a flexible software factory, Proceedings of ACM Software Applications Conference, SAC'94, 1994.
- Gronback, Richard Podcast on GMF, 2006, <http://eclipsezone.com/files/podcasts/1-GMF-Richard.Gronback.html>.
- Haddad, C., Model-Driven Development: Rethinking the Development Process, Burton Group, 2004.
- Haine, P., Second generation CASE: can it be justified?, in: Kathy S.;Paul L. (Ed.), CASE: Current Practice, Future Prospects, Wiley, Chichester, UK, 1992.
- Halstead, M.H., Elements of Software Science, Elsevier, North-Holland, New York, 1977.
- Harel, D., Statecharts: a visual formalism for complex systems, Sci. Comput. Program. 8, 231–274, 1987.
- Heym, M., Österle, H., Computer-aided methodology engineering, Infor. Software Technol. 35(6/7), 345–354, 1993.
- Hietaniemi, J., Perl for Symbian, <http://sourceforge.net/projects/symbianperl/> (accessed January 2006).
- Iivari, J., Relationships aggregations and complex objects, in: Ohsuga, S; Kangassalo, H; Jaakkola, H, Hori, K, Yonezaki, N. (Eds.), Information Modeling and Knowledge Bases III: Foundations, Theory and Applications, IOS Press, pp. 141–159, 1992.
- International Telecommunication Union, Packet-based multimedia communication systems, Recommendation H.323, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, July 2003.
- Isazadeh, H., Architecture analysis of MetaCASE, Ph.D. Thesis, 1997, <http://www.collectionscanada.ca/obj/s4/f2/dsk2/ftp04/mq20654.pdf>.
- ISO ISO-IEC 10027, Information technology—Information Resource Dictionary System (IRDS)—Framework, ISO/IEC International standard, 1990, <http://www.iso.org/>.
- Jackson, M.A., Software requirement & specifications, A Lexicon of Practice, Principles And Prejudices, Addison Wesley, ACM Press, 1995.
- Jarke, M., Pohl, K., Weidenhaupt, K., Lytytinen, K., Marttiin, P., Tolvanen, J.-P., Meta modeling: a formal basis for interoperability and adaptability, in: Krämer, B; Papazoglou,

- M; Schmidt, H. (Eds.), *Information Systems Interoperability*, John Wiley & Sons, Research Studies Press, Somerset, 1998.
- Jarzabek, S., Tok, W.L., Model-based support for business re-engineering, *Inform Software Technol.* 38 (5), 355–374, 1996.
- Jeulin, P., GraphTalk and the Metalogy, 2005, <http://pjeulinmetadone.blogspot.com/2005/06/graphtalk-and-metalogy.html>.
- Johnson, L., A view from the 1960s: how the software industry began, *IEEE Ann. Hist. Comput* 20 (1), 36–42, 1998.
- Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S., Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical report CMU/SEI-90-TR-21, Software Engineering Institute Carnegie Mellon University, 1990.
- Kelly, S., What's in a Relationship: on distinguishing property holding and object binding, in: Hesse, W., Falkenberg, E. (Eds.), *Proceedings of 3rd International Conference on Information Systems Concepts, ISCO 3*, University of Marburg, Lahn, Germany, 1995.
- Kelly, S., CASE tool support for co-operative work in information system design, in: Rolland, C., Chen, Y., Fang, M. (Eds.), *IFIP TC8/WG8.1 Working Conference on Information Systems in the WWW Environment*, Chapman & Hall, 1998, pp. 49–69.
- Kelly, S., Kalle, L., Matti, R., MetaEdit+: a fully configurable multi-user and Multi-tool CASE and CAME environment, in: *Proceedings of the 8th International Conference on Advanced Information Systems Engineering, CAiSE'96*, Heraklion, Crete, Greece May 1996, Constantopoulos et al. (Eds.), *Lecture Notes in Computer Science N:o 1080*, Springer-Verlag, Heidelberg, 1996, pp. 1–21.
- Kelly, S., Tolvanen, J.-P., Visual domain-specific modeling: benefits and experiences of using metaCASE tools, in: Bezivin, J., Ernst, J. (Eds.), *Proceedings of International workshop on Model Engineering, ECOOP 2000*.
- Kieburtz, R., et al. A software engineering experiment in software component generation, *Proceedings of 18th International Conference on Software Engineering*, Berlin, IEEE Computer Society Press, March, 1996.
- Kotteman, J., Konsynski, B., Information systems planning and development: strategic postures and methodologies, *J Manage Inform Syst.* 1, 2, 1984.
- Kyo, C.K., Sholom, G.C., James, A.H., William, E.N., Spencer, A., Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
- Lennox, J., Schulzrinne, H., Call Processing Language Framework and Requirements, May 2000, <ftp://ftp.rfc-editor.org/in-notes/rfc2824.txt>.
- Lennox, J., Wu, X., Schulzrinne, H., Call Processing Language (CPL): A Language for User Control of Internet Telephony Services, October 2004, <http://www.ietf.org/rfc/rfc3880.txt>.
- Long, E., Misra, A., Sztipanovits, J., Increasing productivity at Saturn, *IEEE Comput.* 35–43, 1998.
- Marttiin, P., Rossi, M., Tahvanainen, V.-P., Lyytinen, K., A Comparative review of CASE shells: A preliminary framework and research outcomes, *Inform. Manage.* 25, 11–31, 1993.
- Mathworks, MATLAB Desktop Tools and Development Environment, 2007.
- McCabe, T.J., A complexity measure, *IEEE T Software Eng.* 2, 4, 1976.
- MediaDev, Survey of DSM Attitudes and Tools among Lead Developers. 2006.

- Mellor, S., Balcer, M., Executable UML: A Foundation for Model-Driven Architecture, Preface, Addison Wesley, 2002.
- Mellor, S., Shlaer, S., Object Life Cycles: Modeling the World In States, Yourdon Press, Computing Series, 1991.
- Meta Systems Ltd., QuickSpec. Reference guide, Ann Arbor, Michigan, 1989.
- MetaCase, EADS Case Study, 2006, <http://www.metacase.com/papers/>
- MetaCase, Nokia Mobile Phones Case Study, 2000, <http://www.metacase.com/papers/>
- Moore, M., Monemi, S., Wang, J., Marble, J., Jones, S., Diagnostics and integration in electrical utilities, IEEE Rural Electric Power Conference, Orlando, FL, May 2000.
- Moore, W., Dean, D., Gerber, A., Wagenknecht, G., Vanderheyden, P., Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework, IBM Redbook, 2004, <http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg246302.html?Open>.
- Narraway, D., Designing and generating mobile phone applications, Presentation at MetaEdit+ Method Seminar, 6th Nov, Helsinki, Finland, 1998.
- National Instruments, Labview Fundamentals, User Manual, 2005.
- Nokia, Python for Series 60: API reference, version 1.0, 2004, <http://www.forum.nokia.com/>.
- Nokia, S60 SDK documentation, version 2.0, 2005, <http://www.forum.nokia.com/>.
- Nordstrom, G.G., Metamodeling—Rapid Design and Evolution of Domain-Specific Modeling Environments, Ph.D. Thesis, Vanderbilt University, 1999, http://www.isis.vanderbilt.edu/publications/archive/Nordstrom_GG_3_0_1999_Metamodeli.pdf.
- Olle, T.W., Sol, H.G., Verrijn-Stuart, AA., Proceedings of the IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies, North-Holland, Amsterdam, 1982.
- OMG, Meta Object Facility (MOF) specification, April, 2002, <http://www.omg.org/docs/formal/02-04-03.pdf>.
- OMG, MDA Guide Version 1.0.1, 2003, <http://www.omg.org/docs/omg/03-06-01.pdf>.
- OMG, Meta Object Facility (MOF) 2.0 Core Specification, 2005, <http://www.omg.org/docs/ptc/04-10-15.pdf>.
- OMG, Object Constraint Language, Version 2.0, 2006, <http://www.omg.org/docs/formal/06-05-01.pdf>.
- Pastor, O., Ramos, I., OASIS: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach, UPV Publication Service, SP-UPV, pp 95–788, 1995.
- Pocock, J.N., VSF and its relationship to open systems and standard repositories, in: Endres, A., Weber, H., (eds.), Software Development Environments and CASE Technology, Springer-Verlag, Berlin, 1991.
- Raistrick, C., Francis, P., Wright, J., Carter, C., Wilkie, I., Model Driven Architecture with Executable UML, Cambridge University Press, 2004.
- Raunio, A., Experiences on using DSM at EADS, DSM seminar presentation, in: Koskinen, M., Jauhainen, E., (Eds.), Proceedings of Finnish Computing Science Days, University of Jyväskylä, TU-25, 2007.
- Ripper, P., V-Market: A framework for agent mediated e-commerce systems based on virtual marketplaces, Msc. Dissertation, Computer Science Department, PUC-Rio, 1999.
- Rosen, M., Enterprise Architecture Advisory Service, Cutter Consortium. 2006.

- Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E., SIP: Session Initiation Protocol, RFC 3261, June 2002.
- Rossum van, G., Drake, F.L., Jr. Extending and Embedding the Python Interpreter. Available (accessed Jan 2006), <http://www.python.org/doc>.
- Rozenberg, G., (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 1: Foundations, World Scientific, 1997.
- Rumbaugh, J., Jacobson, I., Booch, G., The Unified Modeling Language Reference Manual, Addison-Wesley, 1999.
- Rust, K., personal communication on Westmount CASE tools, 1994.
- Ruuska, J., Factors of CASE tool usability: An empirical study in a telecom company (in Finnish), University of Tampere, 2001.
- SAE, AADL, Architecture Analysis & Design Language, SAE standard AS5506, 2004.
- Schipper, M., Joosten, S., A validation procedure for information systems modeling techniques workshop on Evaluation of Modeling methods in Systems Analysis and Design, 8th Conference on Advanced Information Systems Engineering, (CAiSE'96) 1996.
- Seeley, R., ADT at Gartner ITxpo: Gates sees more modeling, less coding, Application Development Trends 3/30/2004, 2004, <http://www.adtmag.com/article.aspx?id=9166>.
- Smolander, K., OPRR: a model for modelling systems development methods, in: Lyytinen, K., Tahvanainen, V.-P. (Eds.), Next Generation CASE Tools, IOS Press, Amsterdam, The Netherlands, 1991.
- Smolander, K., Lyytinen, K., Tahvanainen, V.-P., Marttiin, P., MetaEdit—a flexible graphical environment for methodology modelling, in: Andersen, R., Bubenko, J.A., jr; Solvberg, A. A. (Eds.), Advanced Information Systems Engineering, Proceedings of the Third International Conference CAiSE'91, Trondheim, Norway May 1991, Springer-Verlag, Berlin, 1991.
- Software Productivity Research, Programming Languages Table™ (PLT2006b), 2006.
- Stahl, T., Völter, M., Model-Driven Software Development: Technology, Engineering, Management, Wiley, 2006.
- Ströbele, T., EclipseUML—UML und Eclipse, OOP Conference, 24–28 January, Munich, 2005.
- Sutherland, I.E., Sketchpad: a man-machine graphical communication system, Proceedings of the AFIPS Spring Joint Computer Conference, Washington, D.C., 1963, pp. 329–346.
- Sztipanovits, J., Karsai, G., Bapty, T., Self-adaptive software for signal processing, Communications of the ACM May 1998, 66–73.
- Teichroew, D., Ernest, A.H., III, PSL/PSA: a computer-aided technique for structured documentation and analysis of information processing systems, IEEE T Software Eng. 3, (1), 41–48, 1977.
- Teichroew, D., Petar, M., III, Ernest, A.H., Yuzo, Y., Application of the entity-relationship approach to information processing systems modelling, in: Chen, P.P. (Ed.), Entity-Relationship Approach to Systems Analysis and Design, North-Holland, 1980.
- Tolvanen, J.-P., Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence (Ph.D. thesis), Jyväskylä Studies in Computer Science, Economics and Statistics, Jyväskylä: University of Jyväskylä, 1998, <http://www.cs.jyu.fi/~jpt/doc/index.html>.
- Tolvanen, J.-P., Kelly, S., Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. Proceedings of the 9th International Software Product Line Conference, Springer-Verlag, LNCS3714, 2005.

- Turing, A.M., On computable numbers, with an application to the entscheidungsproblem, Proc. London Mathe. Soc., Ser. 2 (42), 230–265, 1937.
- Venable, J., CoCoA: A conceptual Data Modeling Approach for Complex Problem Domains, Dissertation, State University of New York at Binghamton, USA. 1993.
- Voelter, M., MDSD/PLE Conference in Leipzig, <http://voelterblog.blogspot.com/2006/mdsdp-le-conference-in-leipzig.html>.
- Weber, R., Zhang, Y., An Analytical evaluation of NIAM's grammar for conceptual schema design. Information Systems Journal, 6, 1996.
- Weiss, D., Lai, C.T.R., Software Product-line Engineering, Addison Wesley, Longman, 1999.
- Welke, R.J., The CASE Repository: More than another database application, in:Cotterman, W. W., Senn, J.A. (Eds.), Proceedings of 1988 INTEC Symposium Systems Analysis and Design: A Research Strategy, Atlanta, Georgia, Georgia State University, 1988.
- White, S., Software architecture design domain, Proceedings of Second Integrated Design and Process Technology Conf., Austin, TX, December. 1–4, 1996, pp. 283–290.
- Wijers, G., Modeling Support in Information Systems Development, Thesis Publishers, Amsterdam, 1991.
- Wittgenstein, L., Tractatus Logico-Philosophicus, Routledge, and Kegan Paul, London, 1922.
- Yourdon, E., Whatever happened to structured analysis?, Datamation , June 1986, pp. 133–138.
- Zachariadis, S., Mascolo, C. Emmerich, W., The SATIN Component System—A Metamodel for Engineering Adaptable Mobile Systems in IEEE T Software Eng. October 2006.

INDEX

- .NET, 87, 315
- 4-layer architecture, 75
- 4GL, 66
- Abstract syntax, 68
- Abstraction, 16
- Abstraction level, 15, 48
- ADA, 24
- Adding modeling concepts, 264
- Agile, 53, 57, 78, 297
- API
 - generating against, 182–185, 221
 - model access, 270, 287
 - tool integration, 389
- Application developer, 67, 89
- Architecture of DSM, 63–64
- Aspects, 51, 73
- Assembler, 153
- ATL, 395
- Autobuild, 297
- Automotive, 234
- AUTOSAR, 35, 46
- Behavior, 50, 71
- Binary relationships, 369
- Binding, 109, 251
- BNF (Backus–Naur Form), 76
- BPMN (Business Process Modeling Notation), 230
- Branch, 405
- Build, 298
- Business rules, 72
- Business value, 21, 26
- C, 218, 291
- C#, 272, 291, 300
- C++, 81, 186
- Call processing, 97
- CAN, 233
- Cardinality, 250, 370
- CASE, 59, 66
- CDIF, 76
- Checking models
 - with generators, 84, 273
 - with metamodels, 69, 74
- Class diagram, 9
- Code generation, 49, 79–83, 267–310
- CodeWorker, 273

- Component framework, 312, 323
- Concrete syntax, 70, 261
- Connectivity constraints, 251
- Consistency, 245, 250, 262
- Constraint language, 241, 372
- Default values, 242, 249–251
- Deployment of DSM, 91, 347–352
- Developer expertise, 31
- Development productivity, 16
 - cases, 22
 - maintenance phase, 25
- Document generation, 85, 280
- Document Type Definition, 102
- DOM, 283
- Domain
 - candidate selection, 329–333
 - concepts, 3, 48, 228
 - definition, 3
 - economics, 34–39
 - examples, 47
 - experts, 79, 89
 - framework, 311–329
 - knowledge, 33
 - rules, 251
 - size, 46
 - specific language, 63
 - vocabulary, 230
- Domain-specific modeling language, 13, 68
- DOME, 273, 361
- DSL tools, 272, 392
- DSM
 - architecture, 64
 - benefits of, 21–34, 52
 - definition process, 91, 329–356
 - deployment, 91, 347–352
 - evolution, 18, 66, 353
 - examples, 93–96
 - in use, 397–407
 - maintenance, 91
 - testing, 345
 - tool, 60, 357–395
 - use scenario, 343
 - users, 89–91
 - versioning, 346
- DTD, 102
- Eclipse, 38, 394
- Economics of DSM, 34–39
 - cost of waiting, 39
 - development costs, 24, 36
 - maintenance costs, 25, 38
 - ownership, 39
 - payback, 36
 - productivity, 16, 22
- ECU, 233
- EEPROM, 141
- Embedded software, 26, 29, 83, 141, 219, 292
- Ergonomist, 90
- Evaluating languages, 261–264
- Executable UML, 56
- Expert concepts, 236
- Expert knowledge, 33
- Extending UML, 58
- Feature
 - configuration, 235
 - modeling, 235
- Financial products, 120
- Flow machine, 287, 292
- Flow model, 71, 100, 106, 310
- FPA, function point analysis, 84
- Framework, 87
 - code, 88, 184
 - developer, 90
- Function-based generation, 290
- General-purpose language, 6, 55
- Generating
 - Assembler, 153
 - C, 47, 218–221, 274
 - C++, 81, 187
 - code, 320
 - data, 320
 - documentation, 85
 - full code, 49
 - Java, 132, 211–218
 - metrics, 84
 - MIDP, 224, 316
 - Python, 176–184
 - skeleton code, 12
 - XML, 65, 113–117, 283–287
- Generator, 64
 - approaches, 270
 - debugger, 382
 - definition process, 269–310
 - developer, 90
 - crawler, 267, 273
 - escape character, 272

- function-based, 176, 290
- output filtering, 381
- output patterns, 276
- principle, 80
- recursion, 156, 300
- template-based, 272
- testing, 304
- tools, 380–383
- transition-based, 294
- usage, 83
- visitor pattern, 271
- GEF, 375
- GIF, 281
- GME, 373, 391
- GMF, 394
- GOPRR, 76
- GOTO, 289
- Graphical
 - diagram, 50
 - language, 50, 259
- GUI, 59, 161, 234, 260
- Hardware concepts, 232
- HMI, 90, 234
- Home automation, 141
- IDEF, 45
- Image formats, 281
- Industry experiences, 22–24, 93–190
- Information hiding, 51
- Infotainment system, 234
- Instance, 76
- Insurance products, 120, 237
- Integrated languages, 74, 253, 382
- Interaction diagram, 11, 71, 238
- Inversion of control, 325
- IP telephony, 97
- Iterative generator, 287, 290
- J2EE, 87, 120, 132
- Java, 135, 208–211, 272
- Java MIDP, 161, 224, 316
- Javascript, 272, 282
- JET, 272, 395
- JPEG, 281
- Labview, 35, 59, 71
- Language
 - concepts, 228
- definition process, 227–266
- developer, 89
- evolution, 264
- formalization, 67, 76, 247
- identification, 230
- integration 73, 253, 386
- maintenance, 264
- multiple, 72–74, 239, 253
- notation, 70, 259
- patterns, 231, 250
- representation, 50, 70
- representational styles, 78, 259
- reuse, 344
- rules, 250, 342
- semantics, 69
- syntax, 68
- testing, 261
- users, 51
- Learning, 33, 351
- Legacy code, 17, 54, 331
- Library, 86
 - model, 154, 169
- Lisp, 291
- Look and feel, 233
- M2M (Machine to machine communication), 140
- Make tool, 298
- Management support, 334
- Mandatory values, 251
- Manual coding, 7, 12, 312
- Matrix-based representation, 50, 205, 259
- MDA, Model-Driven Architecture, 57
- MERISE, 45
- Message sequence diagram, 11
- MetaEdit, 362
- MetaEdit+, 361, 391
 - metamodeling language, 76
 - generator language, 178, 273, 275
 - use example, 109, 122, 143, 178, 191
- Meta-metamodel, 75, 367
- Metamodel, 59
 - definition, 74
 - example, 249
 - tool architecture, 59
 - tool support, 383–385

- Metamodeling
 - language, 60, 75, 248
 - process, 247
- Metaprogramming, 324
- Metrics, 84
- MIDP, 161, 224, 316
- Military, 23
- Minimizing modeling work, 53, 170, 262
- Mobile phone, 22, 160
- Model, 77–79
 - checking, 84, 251, 279
 - completeness, 250
 - debugging, 52, 78, 224
 - hierarchy, 110, 129, 150, 248, 342, 371, 382
 - keeping up-to-date, 12
 - layering, 108
 - of computation, 70, 237–239
 - refactoring, 30, 74, 245
 - representation, 50
 - reuse, 30, 108, 256, 270, 397
 - to model transformation, 254, 272
 - trace, 224
 - users, 79
 - versioning, 78, 254, 404
 - views, 50
 - visitor, 271
- Modeling
 - behavior, 10
 - code, 54
 - conventions, 246, 250
 - structure, 8, 71
 - training, 32, 351
 - tool, 59
- Modeling concepts
 - definition, 240
 - extensions, 58, 72, 240–242, 263
 - identification, 229–232
 - naming, 240, 246
- Modeling language, see Language
- Model-driven development, 5
- Model-to-model transformation,
 - 254, 272
- MOF, 76, 121
 - metamodel example, 123
- Multiple languages, 73, 253
- Multiple modelers, 391–404
- Multiplicity, 239
- MVC (Model-View-Controller), 196, 260
- N-ary relationships, 128, 251, 369
- Naming conventions, 240, 246, 251
- Notation, 70, 257
 - defining symbols, 259
 - examples, 112, 131, 150–151, 174, 208
 - photorealism, 257
- Notational element, 258
- OAW (openArchitectureWare), 395
- Occurrence constraint, 251
- OCL, 56, 58, 372
- Operational semantics, 70
- Organizational change, 348
- Outsourcing, 26, 33
- Ownership of a DSM solution, 39
- Partial class, 300, 322
- Pattern, 231, 250, 276
- Payback, 36
- PDA, 186, 234
- Phone application, 161
- Phone switch, 24
- Photorealism, 257
- PHP, 272
- Physical product structure, 232
- Pilot, 91, 345–347
- Platform independent, 57, 317
- Platform-specific, 57
- PNG, 281
- Problem domain, 16, 49, 229
- Product family, see Product line
- Product line, 24, 36, 192, 243
- Product quality, 27–30
- Productivity, 16, 21
 - development time, 22, 223
 - in maintenance, 25
- Profile, 58, 252
- Proof-of-concept, 91, 335
- Protected regions, 66, 169, 295
- Prototyping, 84
- Python, 161, 176–184, 291
- Quality
 - improvements, 27
 - of generated code, 54, 82–83
- Recursive generation, 288–291
- Refactoring, 30, 263, 345
- Regular expression, 115, 367

- Removing modeling concepts, 265
- Representational styles, 50, 259
- Return of investment, 36
- Reuse
 - model elements, 30, 256
 - rules, 252
- Reverse engineering, 5
- Roles, 79, 89–91
- Roundtripping, 5, 54, 296
- Ruby 291
- Rules
 - checking, 252, 279
 - domain rules, 251
 - modeling rules, 251
 - for reuse, 252
- RTF, 282
- S60, 160–163, 301
- SDL, 292
- Semantics, 69
- Sequence diagram, 11
- Sequential generation, 287, 290
- Session Initiation Protocol (SIP), 97–100
- Simulink, 35, 59
- Singleton class, 324
- Skeleton code, 12
- Sketching, 54
- Smalltalk, 291, 294
- Software architecture, 49
- Solution domain, 47–49, 229
- SSADM, 45
- Standardized languages, 55, 57
- State machine, 10, 57, 71, 170, 195, 241, 292, 320
- State chart, 292
- Static structures, 50, 71
- Stereotype, 58, 240
- SVG, 281
- Switch-case, 217, 293
- Symbian, 160, 186
- Symbol, 258–261, 375
 - of connection boundary, 378
 - of role, 376
- Syntax, 68
- SysML, 55
- Table-based language, 50, 259
- Tail recursion, 291
- Target environment, 86
- TBK/Toolbuilder, 363
- Test engineer, 85
- Testing DSM
 - generator, 304
 - language, 261
 - with DSM, 29, 85
- Textual language, 50
- Tools, 59, 357–396
- Tool
 - developer, 90
 - generator, 380
 - integration, 388
- Training, 32, 351
- Transition table, 294
- Translator, 369, 381
- Turing, 292
- Type, 76
- UML, Unified Modeling Language, 6, 12, 55
 - based generation, 303
 - executable, 56
 - profile, 58
 - stereotype, 58, 240
- Uniqueness constraints, 251
- Updating models, 264
- Use case diagram, 8
- User interface, 13, 160, 206, 234, 317
- Variability space, 235
- Variation, 242–245
- Version control
 - of code, 308
 - of models, 78, 404
- View, 51
- Voice menu, 141
- Waiting costs, 39
- Wizards, 235, 311
- Word, 282
- XMF-Mosaic, 373, 395
- XML
 - DTD, 102
 - generation, 113–117, 132, 135, 283–287
 - schema, 99, 101, 236
- XSLT, 276