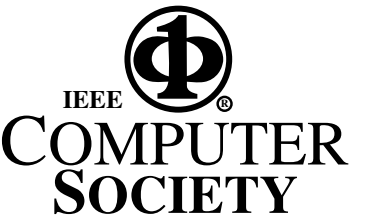
DOMAIN-SPECIFIC MODELING

Enabling Full Code Generation

STEVEN KELLY
JUHA-PEKKA TOLVANEN



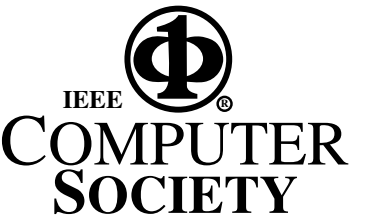
A Wiley-Interscience Publication
JOHN WILEY & SONS, INC.

DOMAIN-SPECIFIC MODELING

DOMAIN-SPECIFIC MODELING

Enabling Full Code Generation

STEVEN KELLY
JUHA-PEKKA TOLVANEN



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

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

Published simultaneously in Canada

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

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

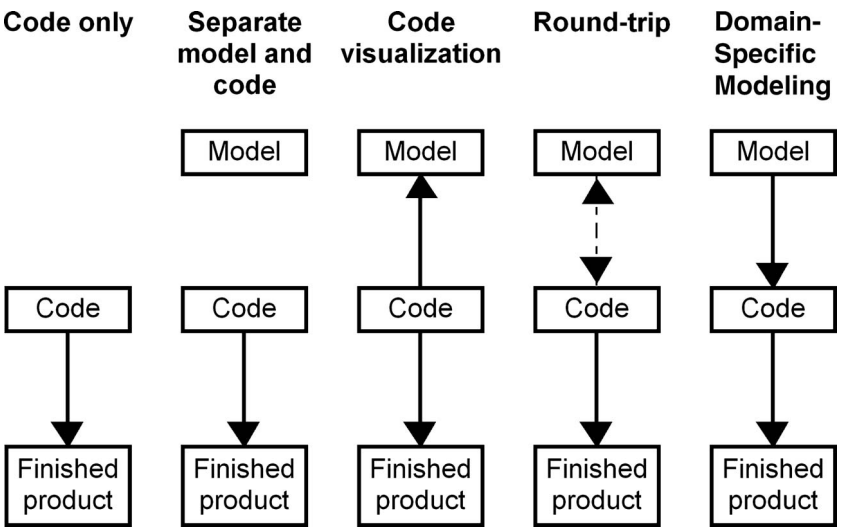


FIGURE 1.1 Aligning code and models

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

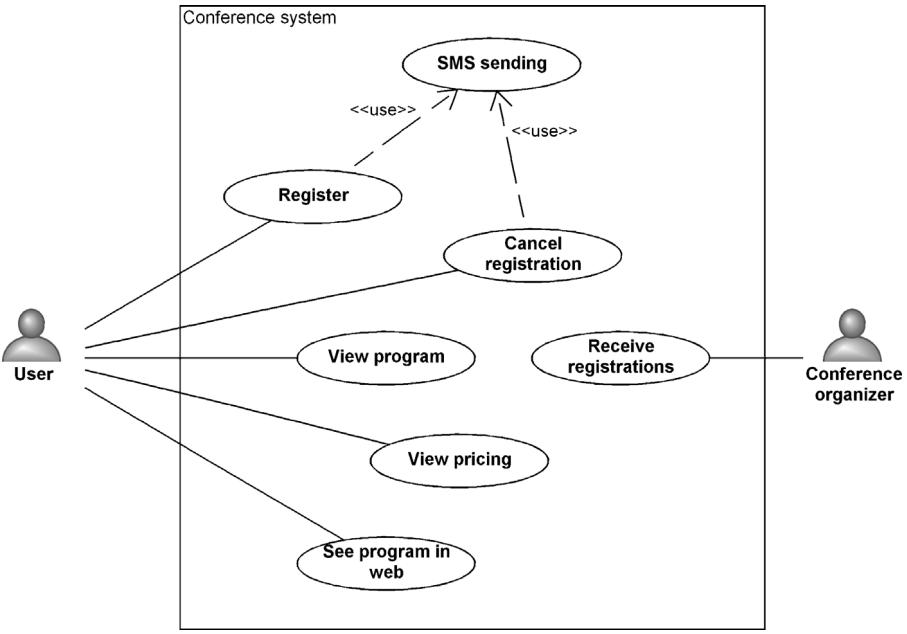


FIGURE 1.2 Use cases of the conference application

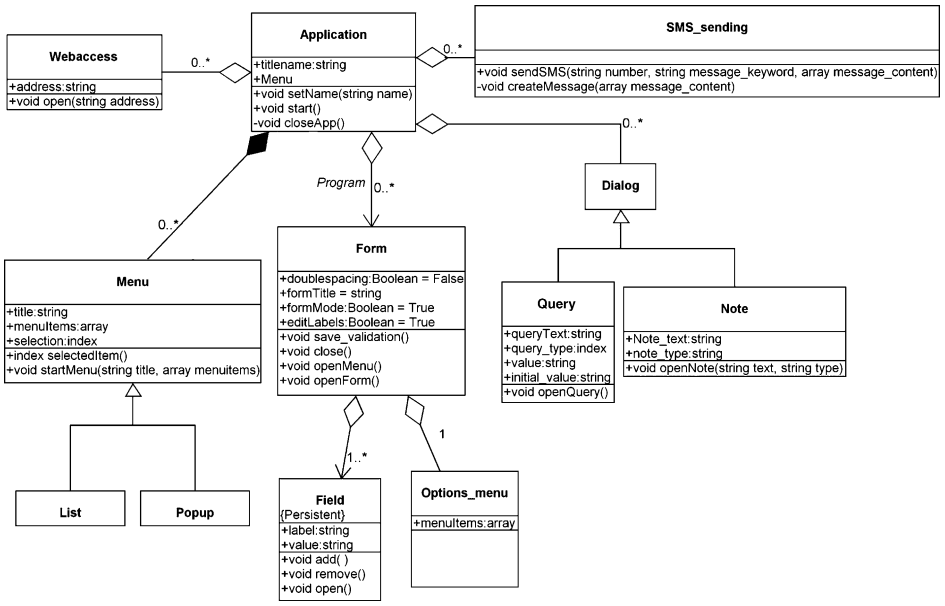


FIGURE 1.3 Class structure of the application

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

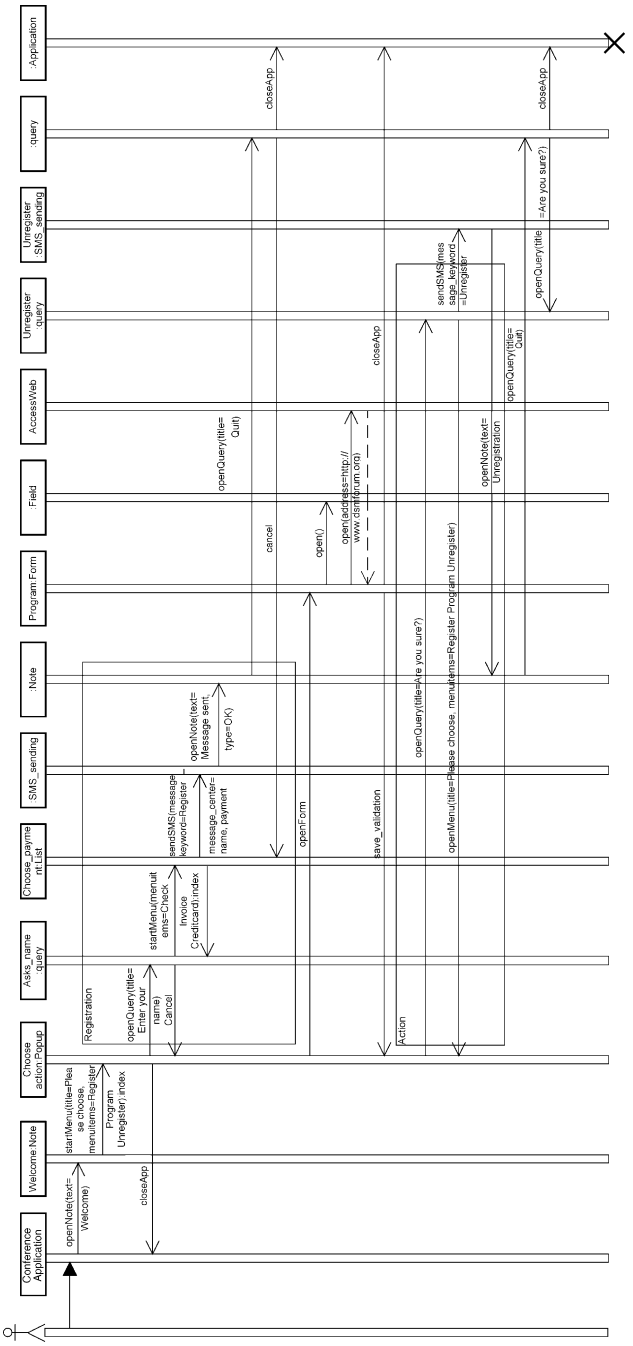


FIGURE 1.4 A view of the application behavior

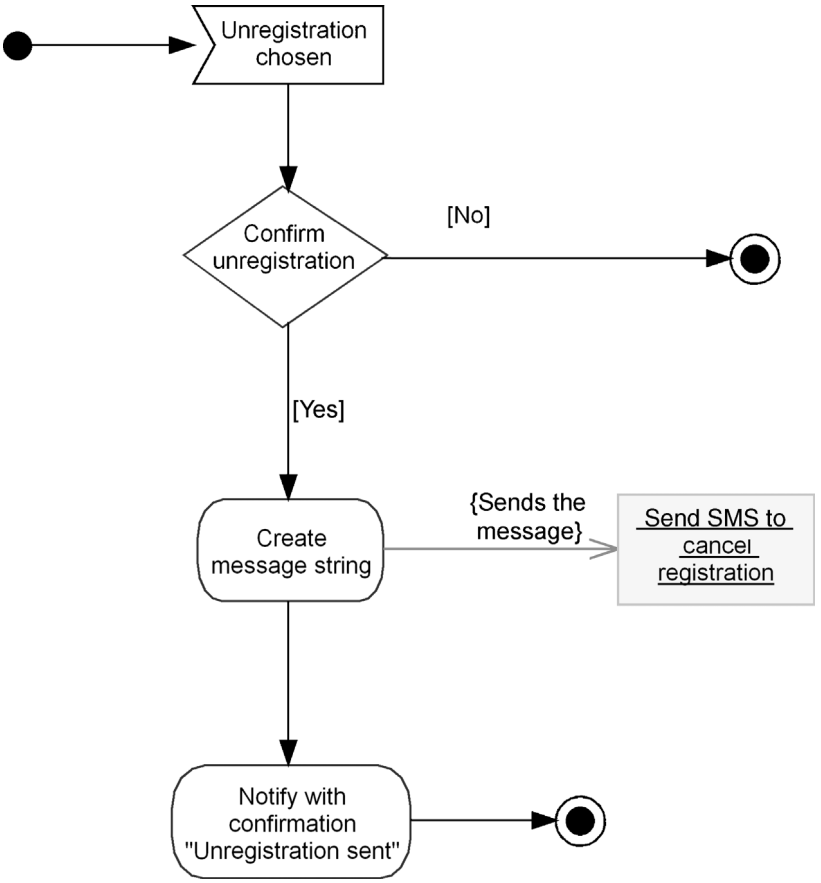


FIGURE 1.5 Activity diagram specifying steps while unregistering from a conference

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 synch 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