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

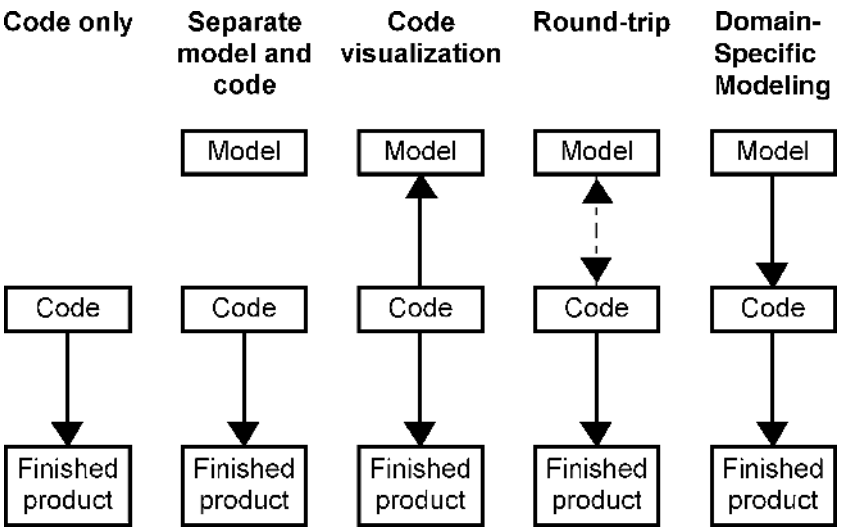

FIGURE 1.1 Aligning code and models

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

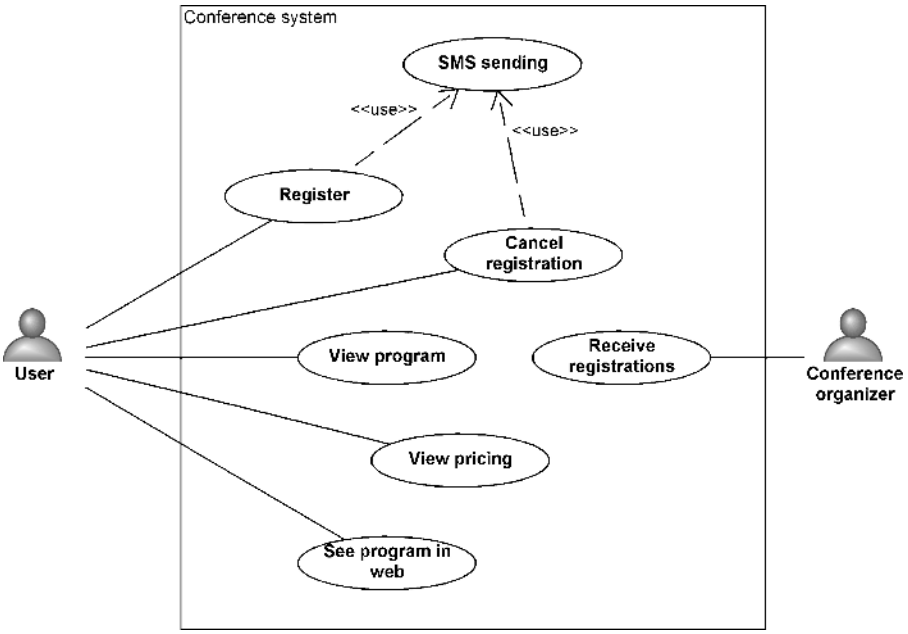

FIGURE 1.2 Use cases of the conference application

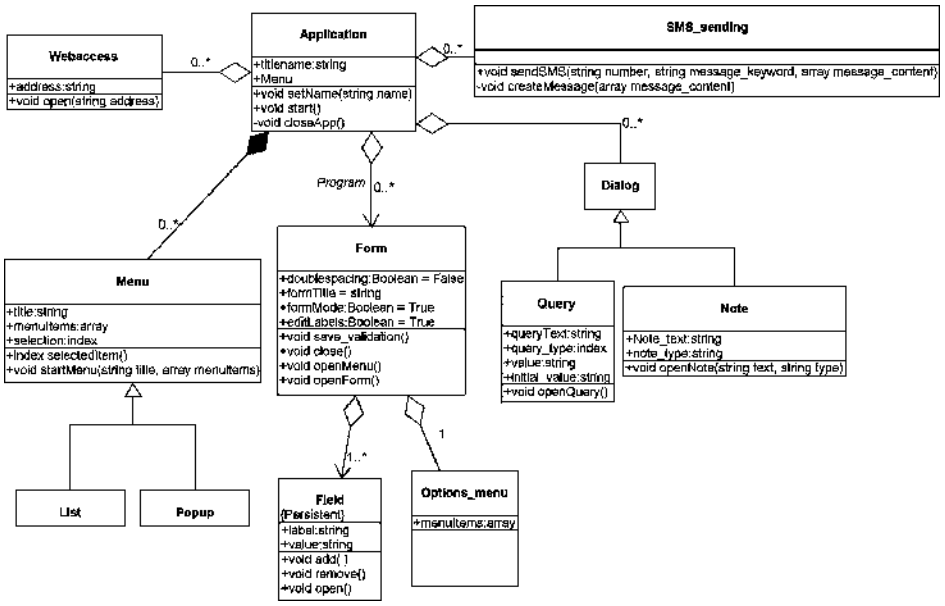

FIGURE 1.3 Class structure of the application

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

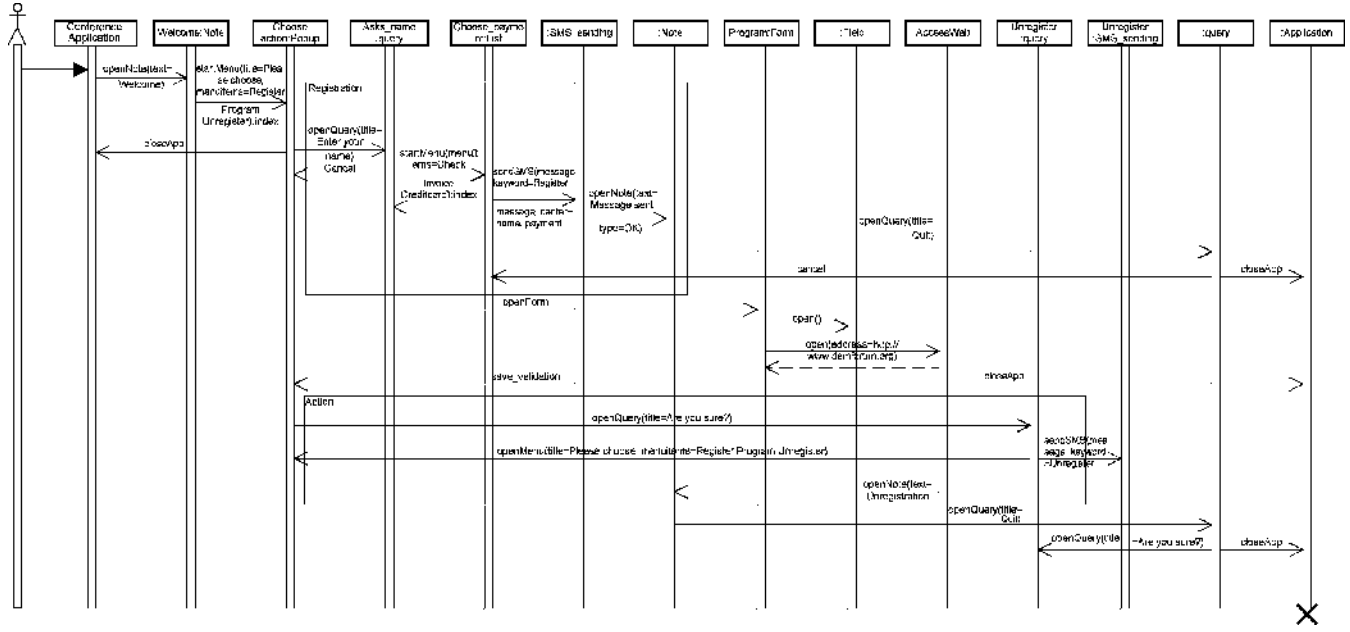

FIGURE 1.4 A view of the application behavior

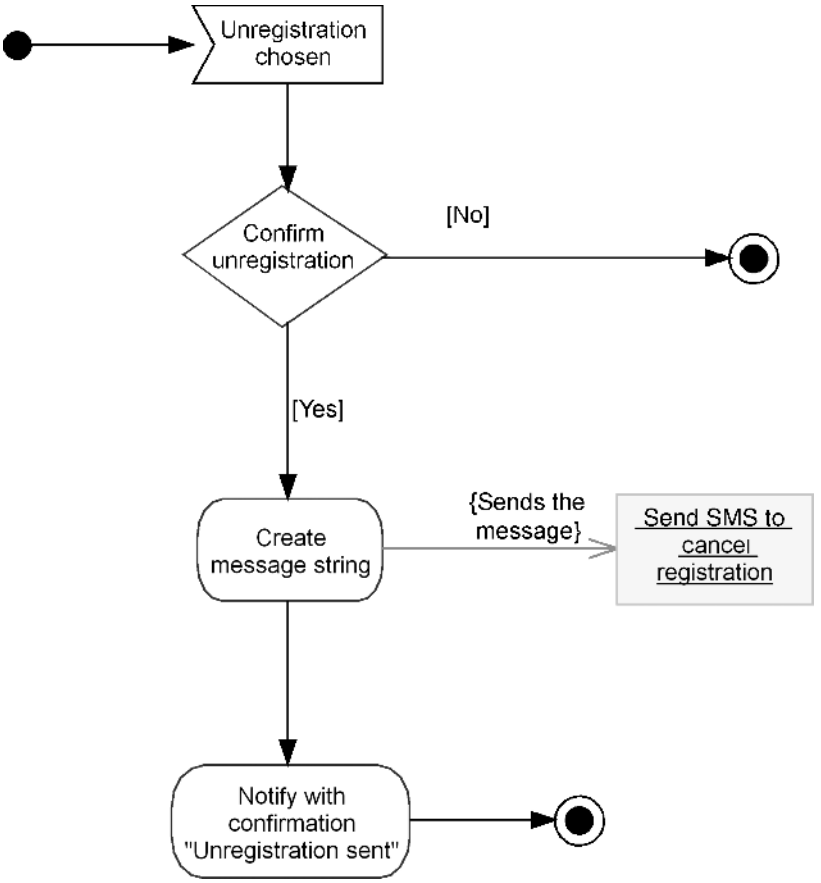

FIGURE 1.5 Activity diagram specifying steps while unregistering from a conference

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

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

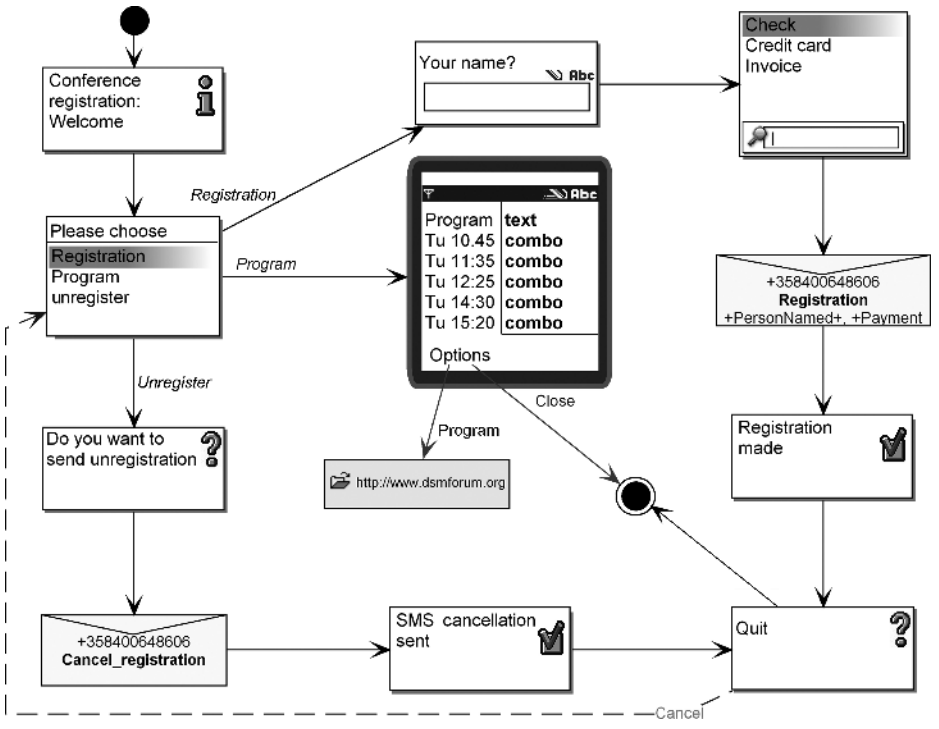

FIGURE 1.6 Conference registration application

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

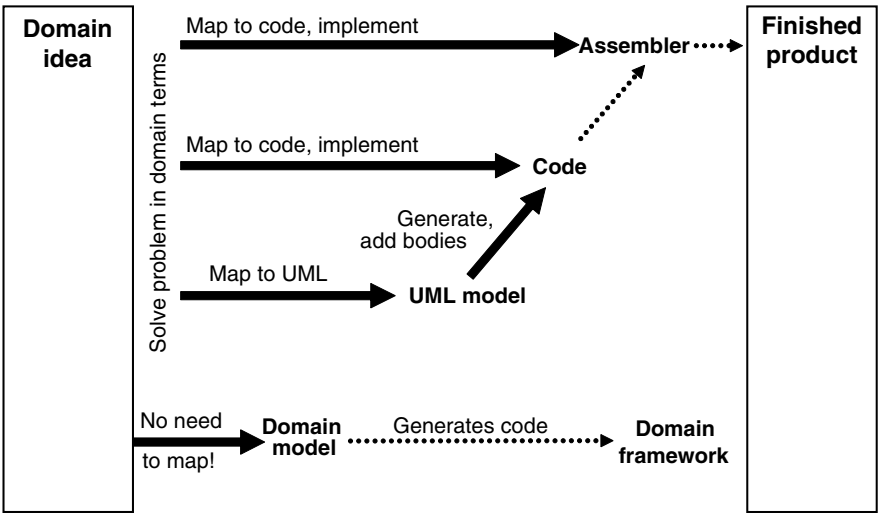

FIGURE 1.7 Bridging the abstraction gap of an idea in domain terms and its implementation

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 in into your organization.