### 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

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

63

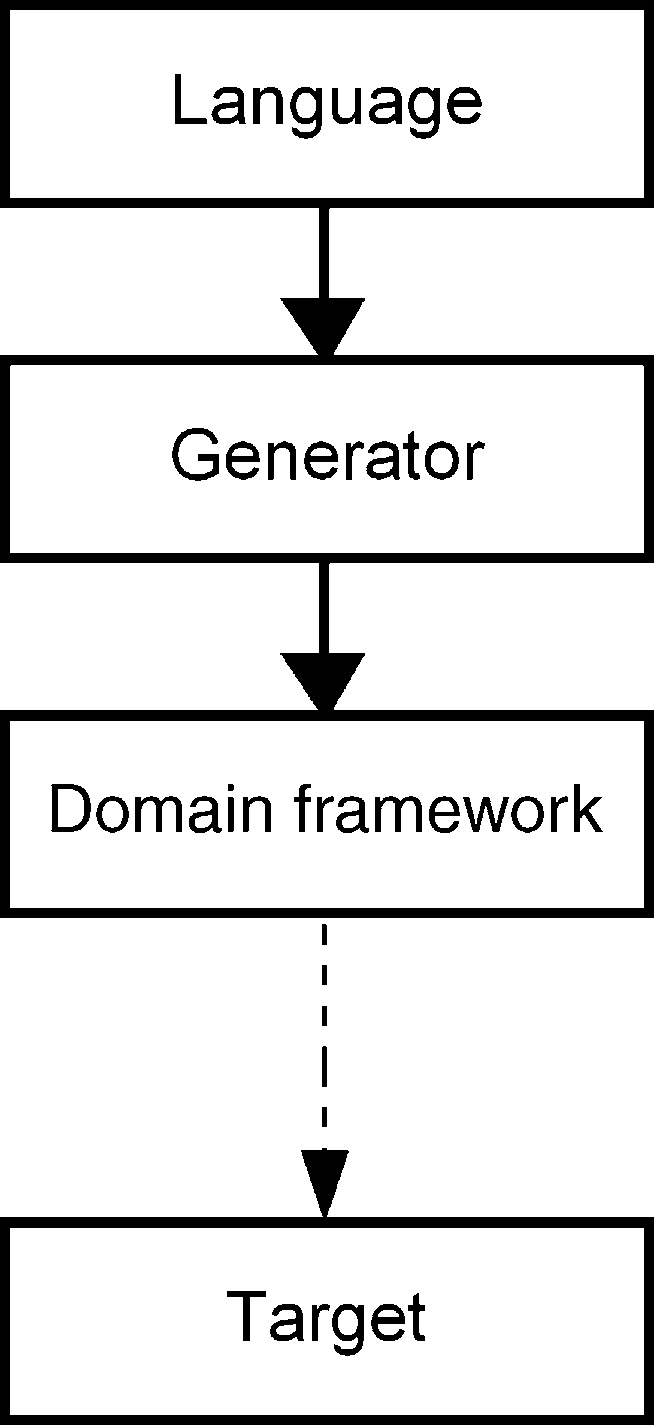


FIGURE 4.1 Basic architecture of DSM

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

### INTRODUCTION

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 workbetween 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

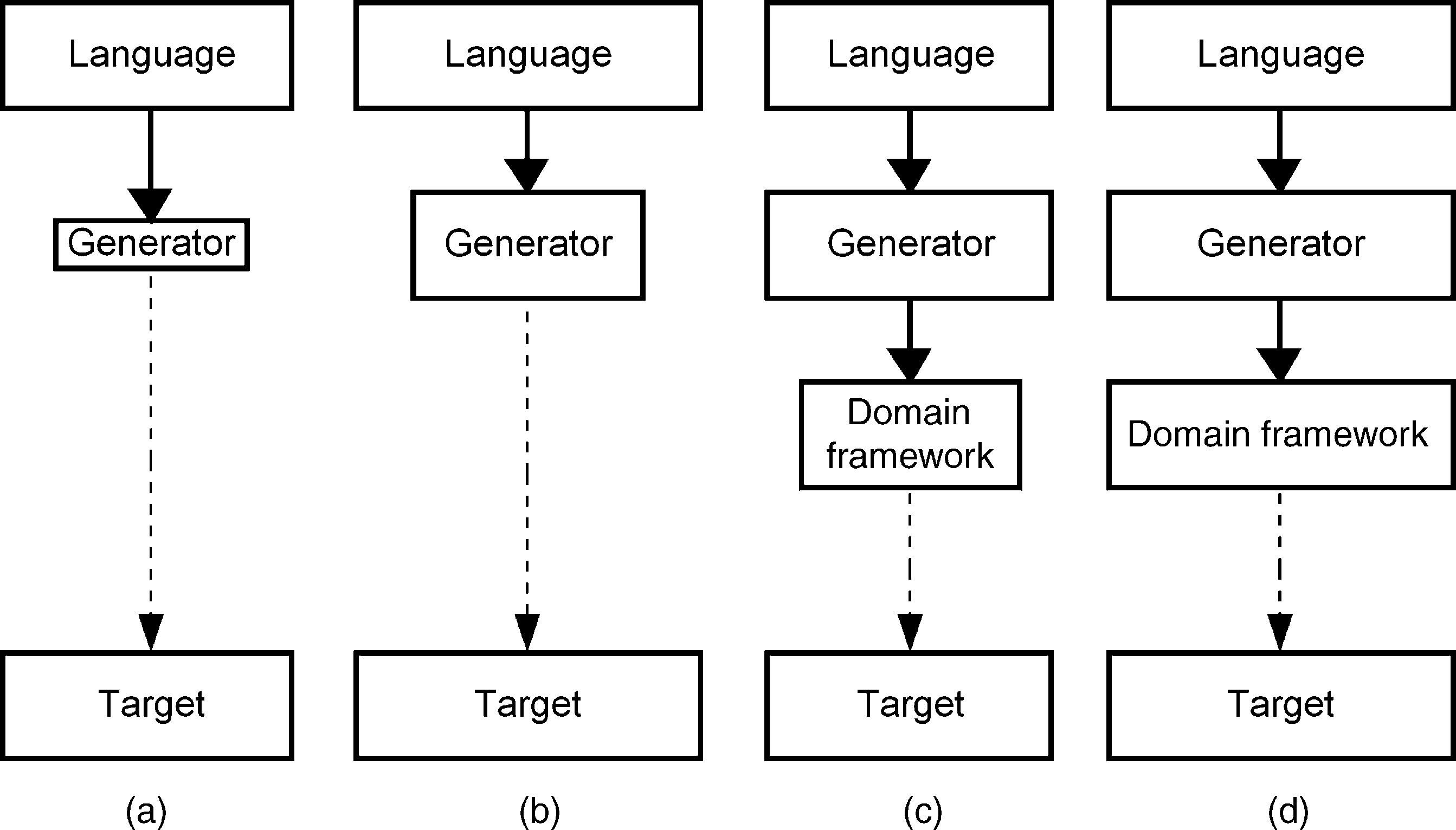


FIGURE 4.2 Dividing the automation work in DSM

(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

### INTRODUCTION

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

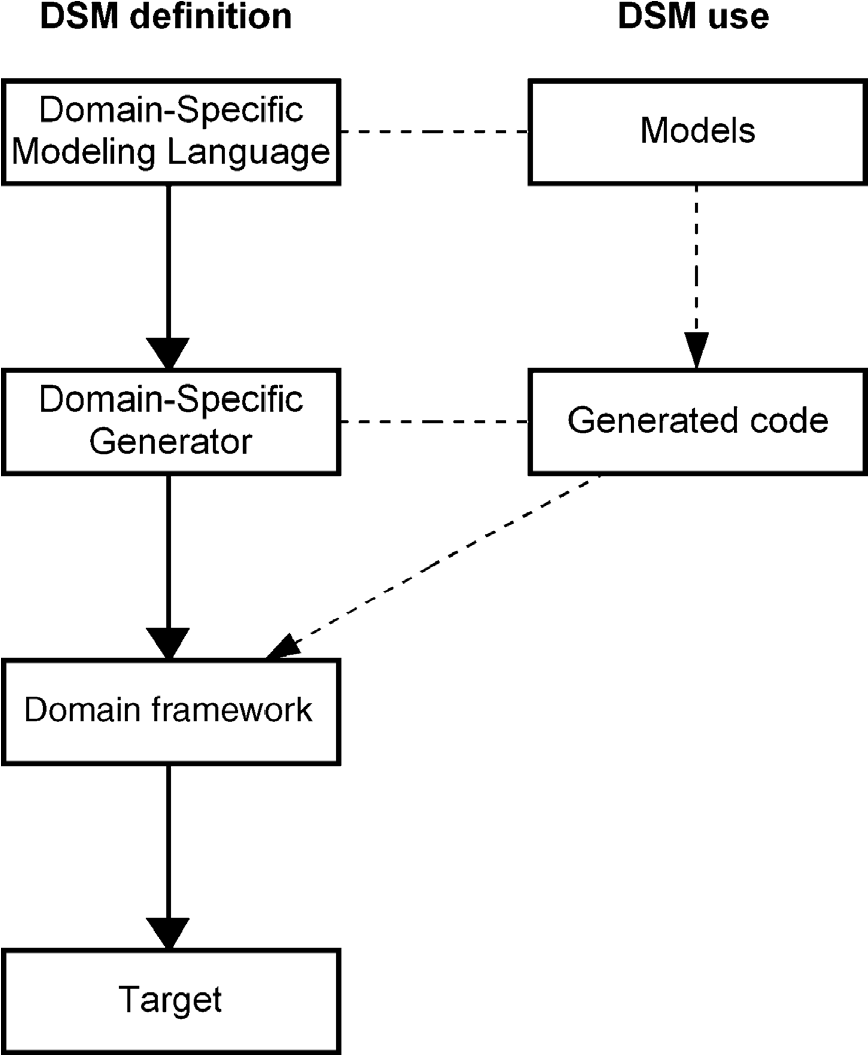


FIGURE 4.3 DSM definition and DSM use

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 wouldtreatassourcecode.Ifthelanguageisformedcorrectly,itshouldapplytermsand 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 havewelldefined 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 oneto-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

Q1 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 nondeterministicly? 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 a 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 Buede (1999).

Modeling Static Structures One class of modeling languages addresses solely, or mostly, the static structures of an application. Perhaps the best known domainspecific 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 offive 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 domainspecific 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

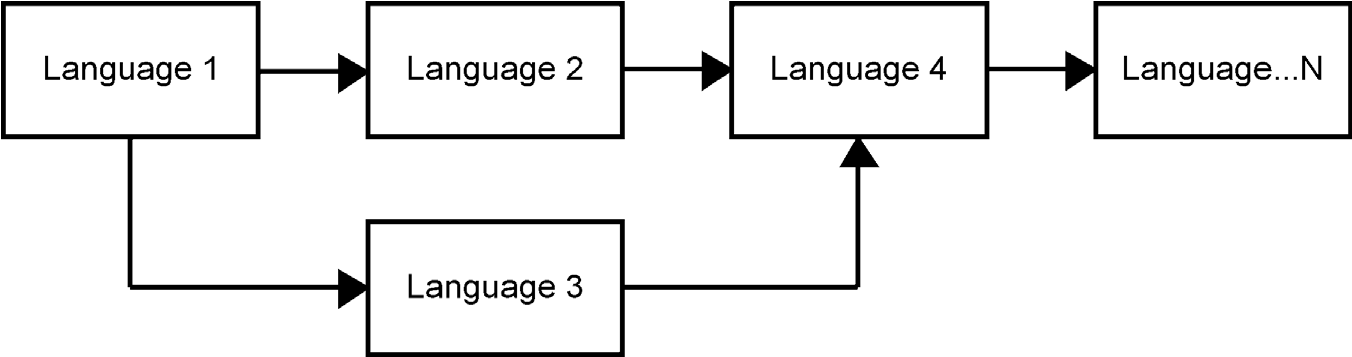


FIGURE 4.4 Integration of languages via transformations

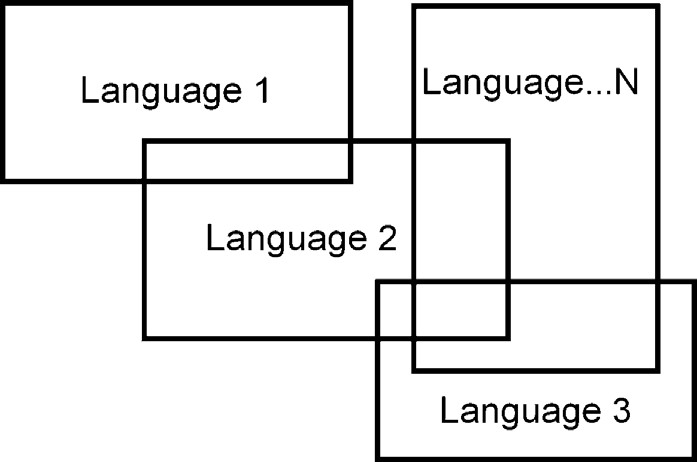


FIGURE 4.5 Integration of languages via common elements

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 Starndards Organization’s IRDS framework (Information Resources Dictionary Standard, (ISO, 1990)) and later in Q2 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.

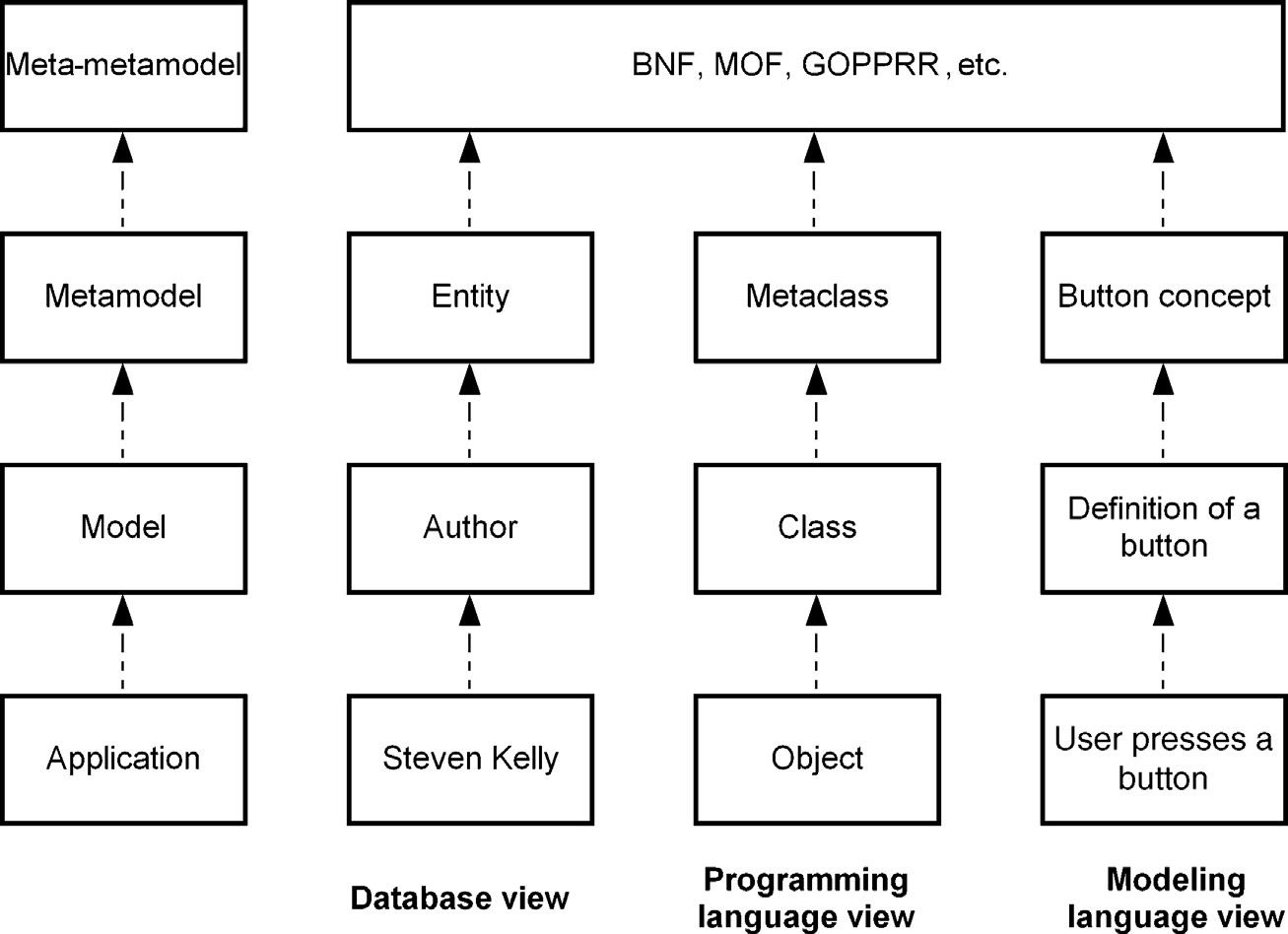


FIGURE 4.6 Four layers of instantiation

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 metametamodel 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

### MODELS

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.

1. // ---------------------------------------------------------
2. // void CAkn**ConferenceRegistration**View::**welcome**()
3. // ---------------------------------------------------------

04

1. void CAkn**ConferenceRegistration**View::**welcome**()
2. {
3. CAkn**Information**Note\* aNote;
4. aNote = new (ELeave) CAkn**Information**Note(ETrue);
5. aNote->ExecuteLD(\_L("**Conference registration: Welcome**"));
6. }

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,totheoutputcode.Thenotificationtype“Information”inlines7and8andthe 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 existingnotificationtypesorbyenteringthevalues.Therestoftheoutputisproducedby 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 notalways 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

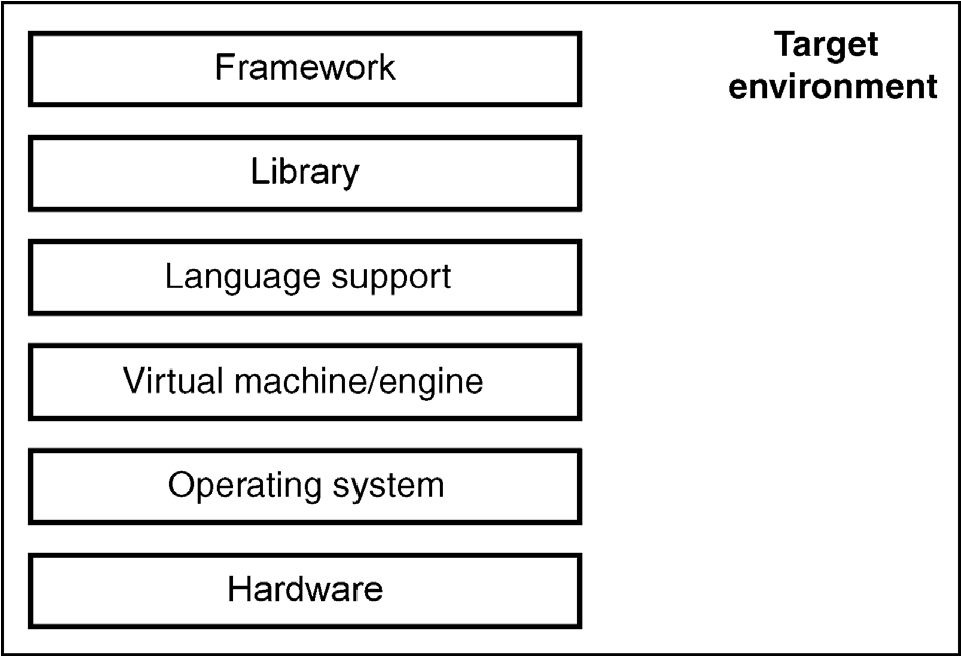


FIGURE 4.7 Layered architecture of a target environment

DOMAIN FRAMEWORK AND 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

### DSM ORGANIZATION AND PROCESS

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

### DSM ORGANIZATION AND PROCESS

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

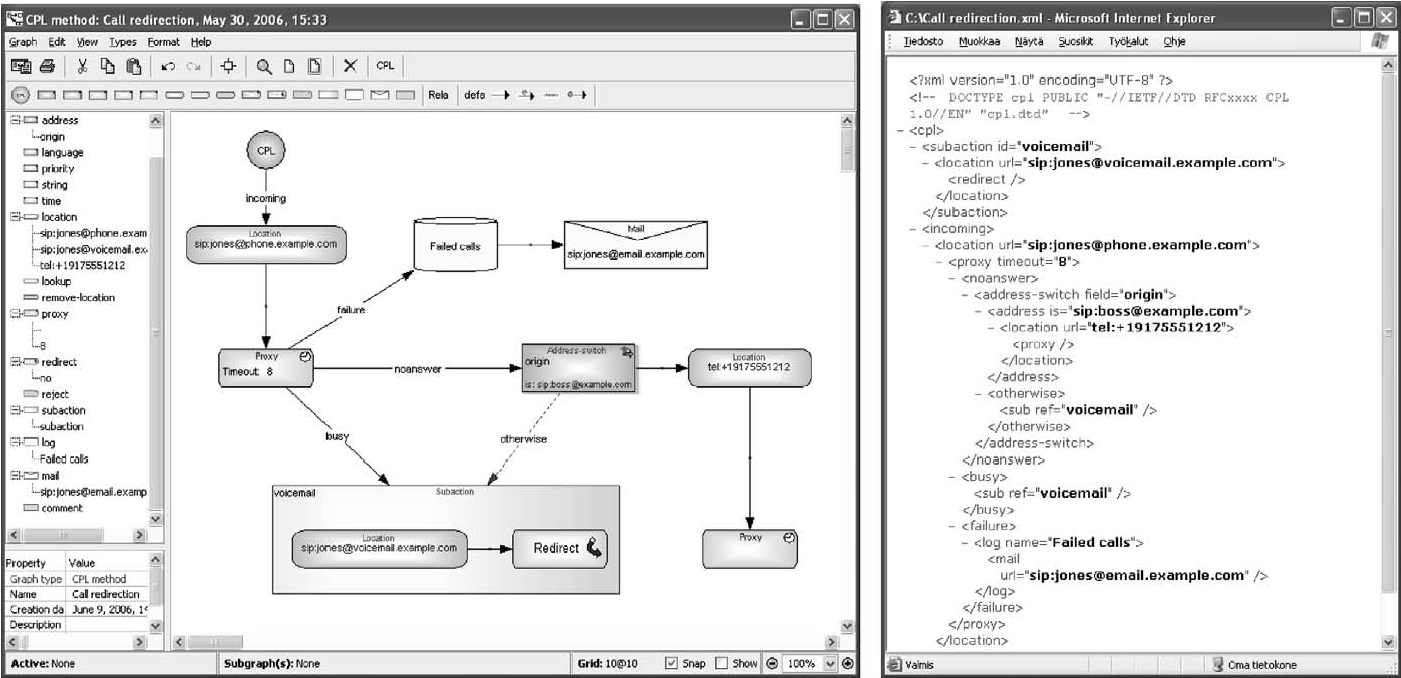
In this part of the book, we illustrate Domain-Specific Modeling (DSM) examples from practice. Following the DSM architecture, we describe for each case the language definitions, how the code generators work, and how the services of the underlying platform and domain framework are used. Later, in Part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

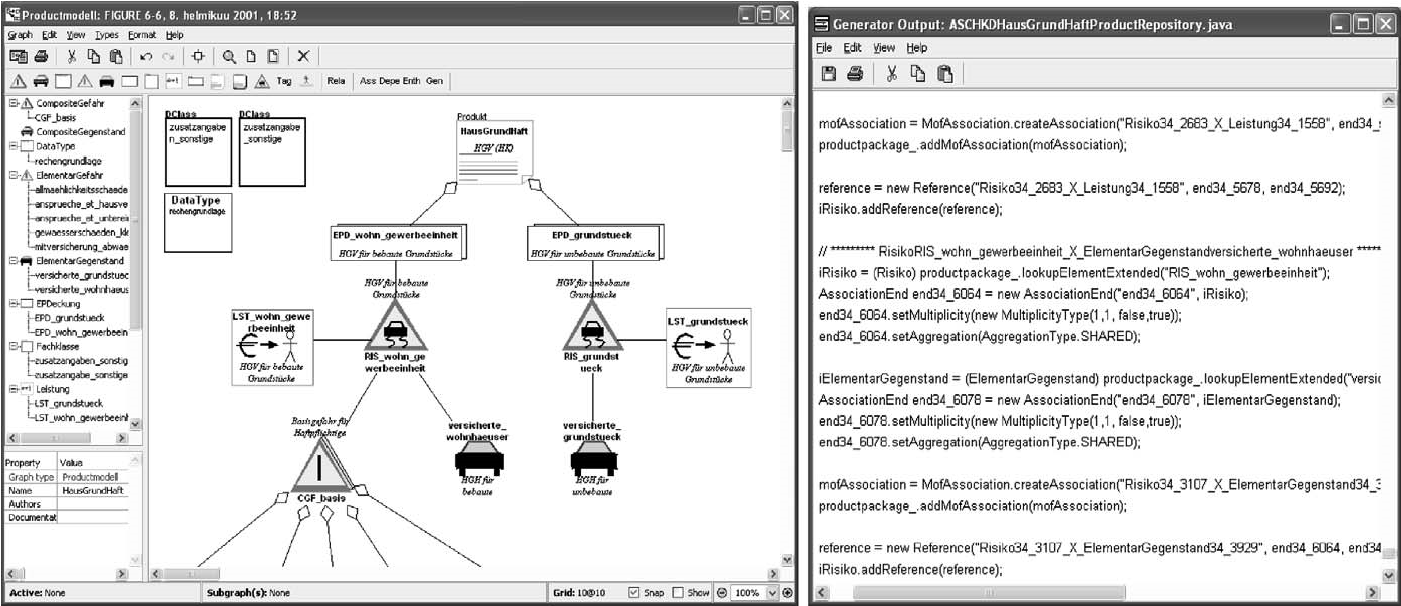
TELEPHONY AND CALL PROCESSING

Chapter 5: IP telephony and call processing



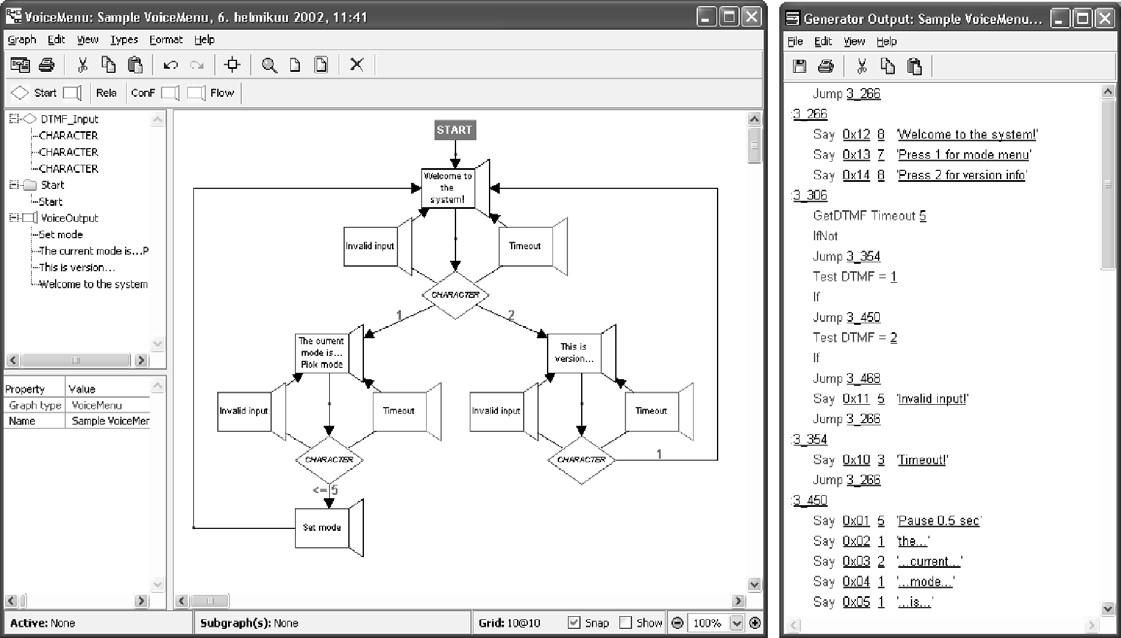
We start with the case of a language based on an XML schema. Chapter 5 illustrates a case for service creation: describing IP telephony services using flow models and generating a service description in XML. A service engineer draws models like the above to define different telephony services, and then 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



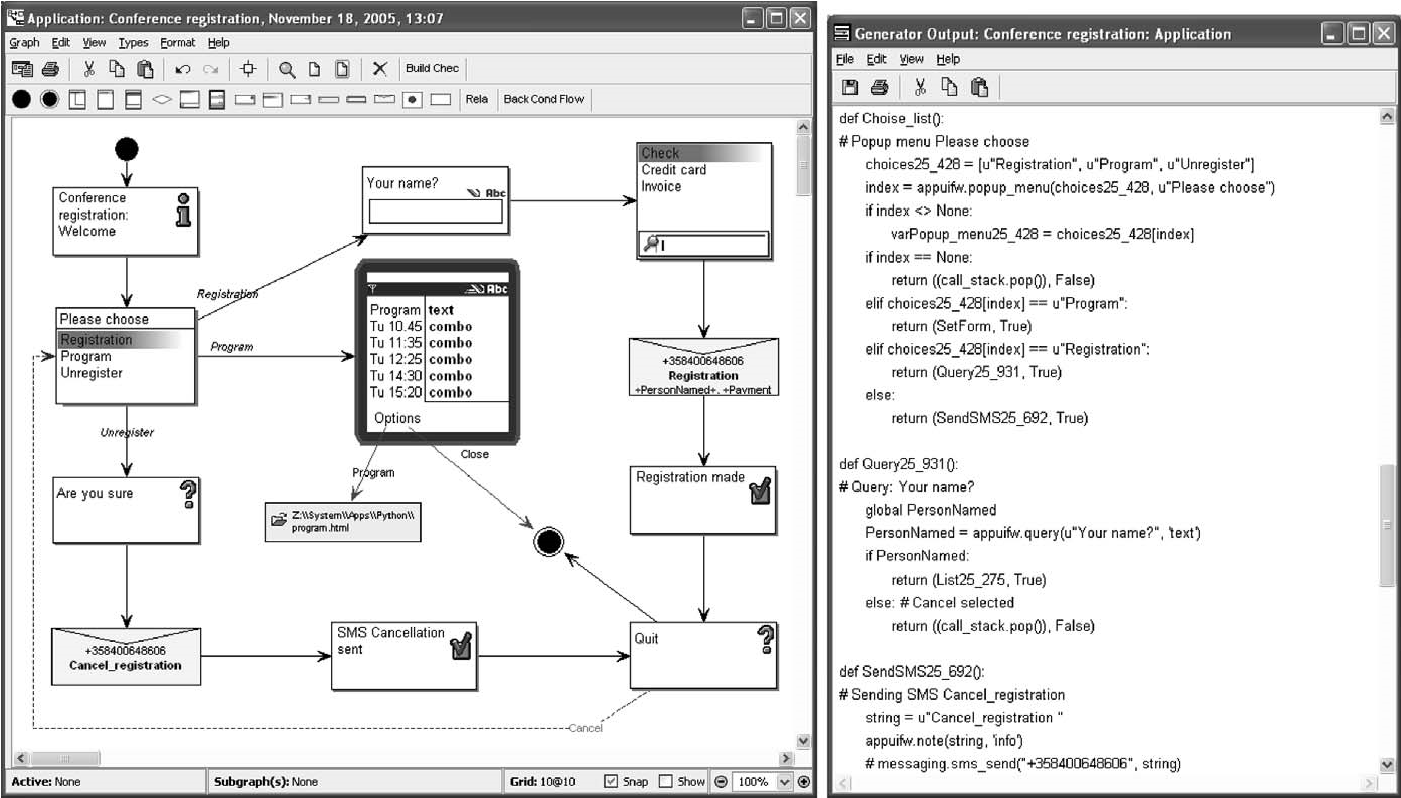
The second example illustrates a case of capturing insurance products using static declarative models. An insurance expert, a nonprogrammer, draws models to define insurance products, and then the generators produce the required insurance data and code for a J2EE web site. As the generated code covers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

TELEPHONY AND CALL PROCESSING 95 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.

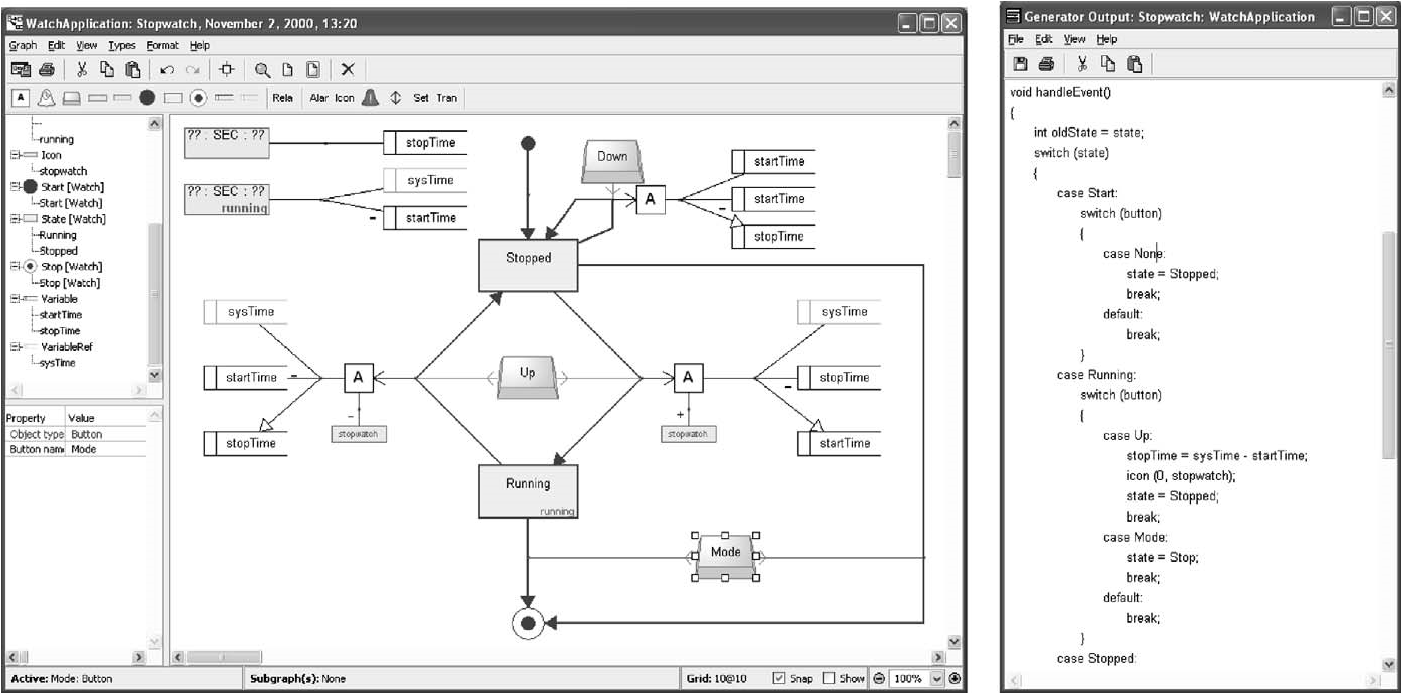
Chapter 8: Mobile phone applications using a Python framework



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.

TELEPHONY AND CALL PROCESSING

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://dsmbook.com along with a free MetaEdit+ demo.