### CHAPTER 3

## DSM DEFINED

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

3.1 DSM CHARACTERISTICS

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

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

45

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

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

3.1.1 Narrow Focus

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

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

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

English.

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

TABLE 3.1 Example Domains for DSM

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

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

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

Narrow Problem Domain For most developers, a narrow focus is set by providing a language that operates on already known concepts relevant to the specific domain and has rules that guide developers in making the specifications. For example, a language can prevent incorrect or poor designs by simply making them impossible to specify. This prevents errors early on—when they are cheapest to correct. Such guidance can be done during every modeling step by checking that models follow the metamodel (language specification) or during a separate model checking process. In the former case, the language could prevent making illegal connections between certain model elements or force a modeler to specify certain data. In the latter case, model checking could report illegal structures or incomplete designs. The narrow focus of the language can also help modelers in following approved design guidelines and in reusing available specifications. By focusing on precise concepts that are already known, the models in DSM become easier to read, remember, check, and validate.

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

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

3.1.2 High Level of Abstraction

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

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

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

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

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

Q2 architectures (Duffy, 2004) already make the work of generators easier by raising the level of abstraction on the code side.

3.1.3 Full Code Generation

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

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

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

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

3.1.4 Representations Other Than Text

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

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

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

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

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

3.1.5 Larger Number of Potential Users

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

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

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

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

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

3.2 IMPLICATIONS OF DSM FOR USERS

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

3.2.1 What Does DSM Offer for Developers?

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

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

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

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

### IMPLICATIONS OF DSM FOR USERS

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

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

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

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

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

3.2.2 What DSM is Not

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

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

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

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

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

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

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

DIFFERENCE FROM OTHER MODELING APPROACHES

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

3.3 DIFFERENCE FROM OTHER MODELING APPROACHES

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

3.3.1 How Does DSM Differ from UML?

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

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

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

3.3.2 How Does DSM Differ from Executable UML?

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

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

DIFFERENCE FROM OTHER MODELING APPROACHES

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

3.3.3 How Does DSM Differ from MDA?

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

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

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

When it comes to standardization, it is obvious that the best DSM solutions will never be standardized. Modeling languages and generators that fundamentally increase productivity and improve quality give competitive advantage and are naturally kept for internal use only. Why would any company that outperforms its competitors publish its DSM solution? Standards that are widely adopted are still good but we should remember that then the leading edge of competition has moved elsewhere. For example, standard libraries are good for automation since they narrow the focus for a DSM solution and often raise the abstraction from the implementation side. For practical DSM, the existence of standard platforms is not that relevant since DSM solutions can be built based on standards or equally well on proprietary and in-house platforms and target environments.

3.3.4 What If We Customize UML?

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

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

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

3.3.5 Other Domain-Specific Approaches

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

### TOOLING FOR DSM

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

3.4 TOOLING FOR DSM

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

3.4.1 Rethinking the Tooling

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

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

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

CASE/UML tool

DSM tool

Modeling

languages

(

hard-coded, fixed

)

Modeling

languages

(

any metamodel

)

Metamodeling

language

(

hard-coded, fixed

)

Models

Models

FIGURE 3.1 CASE or UML tool versus DSM tool

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

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

3.4.2 DSM Tool Capabilities

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

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

### SUMMARY

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

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

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

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

3.5 SUMMARY

DSM copies the principle that made the compiler so successful—raising the level of abstraction by removing the need for developers to write Assembly and letting them work in 3GLs and OOP languages instead. But to make it work for a given company, the principle needs to be applied to that particular setting, tailor-made for the company’s own problem domain and code style. The most feasible way of doing this is to give companies full control over creating and maintaining their own domainspecific modeling languages and generators: Experienced developers in a domain are often better able to define how their products should be developed than external tool vendors.

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

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