### CHAPTER 12

## DOMAIN FRAMEWORK

Domain Specific Modeling (DSM) is only one of a number of ways of improving developer productivity in a given domain. Like many of these ways, it takes advantage of some kind of repetition in applications built for that domain. A common approach is to build a library of reusable components, leaving the application code architecture to the developer. One step further is to build a framework for applications in that domain. As well as providing components, a framework fixes the architecture of the application, with the developer plugging in her own application-specific code at predefined points. Using a framework generally involves a large initial investment of time to learn how it is meant to be used—in the best case by reading extensive and accurate documentation, but too often by trial and error and user forums.

In an attempt to make it easier to get started with a framework, there is sometimes a wizard that will query the developer for necessary information and generate a partially completed skeleton application. This approach is particularly seen where the framework is integrated with an Integrated Development Environment (IDE), for example, early Graphical User Interface (GUI) builders. The problem in many cases with this approach is that it presents the developer with a mass of code that she has not seen, let alonewritten, and yet is expected to maintain. That differs from successful approaches to raising developer productivity, such as compilers, where the developer can remain on a higher level and not have to understand or work with the larger mass of code that is automatically generated on a lower level.

With DSM, the code that is generated remains firmly “under the hood.” The developer keeps working on the model level throughout development. There’s

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

311

no working on an unholy interlinear mix of autogenerated and handwritten code. What DSM does is raise the level of abstraction to a level where things are simple for the developer. Of course, underneath there is code, and underneath that is assembler, machine code, microcode, logic gates, electrons, and so on. The trick is how successfully we can hide the next level down and keep it hidden. DSM does it, most wizards do not.

If wizards represent something of a dead end in the evolution of reuse, how can we best use components as part of a DSM solution? In particular, given we quite probably already have a library of components for our domain, what extra do we need to take best advantage of them with DSM? In our experience, the best approach is to create a separate layer, the domain framework, between the generator and the components.

Fig. 12.1 compares manual coding, wizards, and DSM. The darker blocks represent the artifacts that the developer must work with. In the first approach, manual coding, all code is written and maintained by hand on top of an existing set of components and platform. In the second approach, the developer gives input to a wizard that generates some of the code. The developer has to add code and is also responsible for maintaining the code generated by the wizard. The input to the wizard is often lost, or the wizard cannot be repeated. Even in the best case, there are a bunch of gotchas, documented and undocumented, awaiting the developer who tries to repeat the wizard after significant manual coding. In the third case, the developer works solely with models in a DSM language. The code is generated from those models and takes advantage of a domain framework.

Unlike the component framework it rests upon, which is generic for all problem domains and/or all companies, the domain framework is specific to the problem domain and the particular company that is using the DSM solution. This allows the generated code to be smaller than the generated or handwritten code in previous approaches. Although the generated code and domain framework together will be

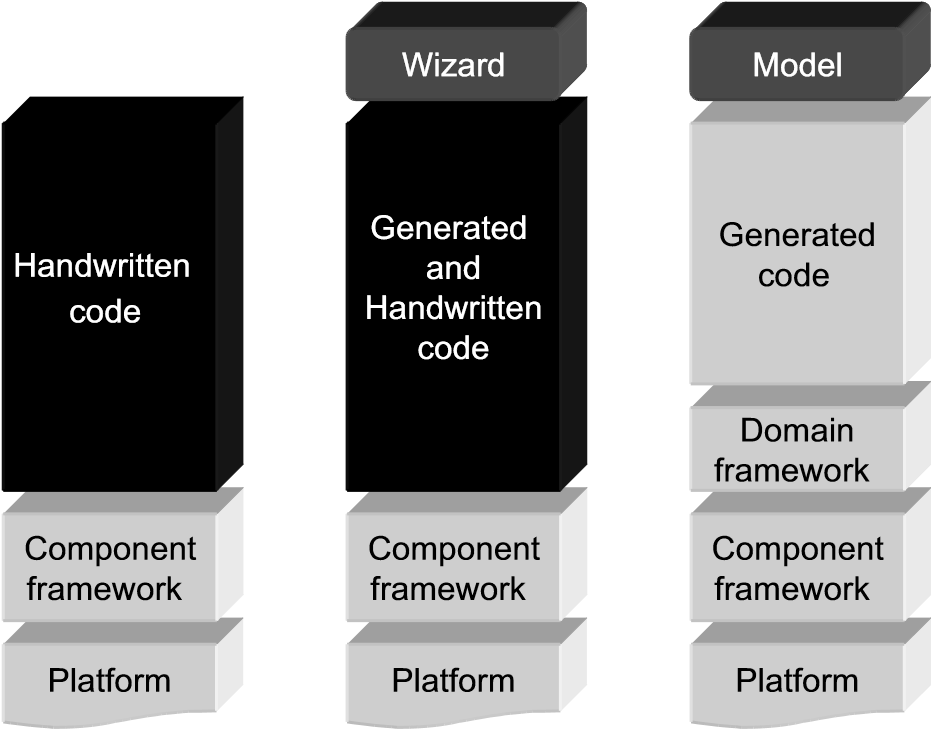


FIGURE 12.1 Manual coding versus Wizards versus DSM

REMOVING DUPLICATION FROM GENERATED CODE

around the same size as previous approaches, the amount of code across the range of products will be less since the domain framework is only counted once.

In addition to making the generated code smaller, benefiting the developer, the domain framework also helps the metamodeler. It insulates the generator from changes in the components, and at the same time allows us to make concessions to ease of generation, without changing the components themselves. In many cases, the components have arisen from spotting repeated patterns in the code, rather than being systematically architected for the problem domain. The domain framework thus also offers us a chance to provide a more domain-oriented, less implementation-oriented, interface to existing proven functionality.

Before this starts sounding like too ambitious a project, let’s reassure ourselves. The domain framework does not need to raise code to a rarefied level of abstraction, easily digestible by any third grader. We have two other layers above it, the generator and the modeling language, which bear the bulk of that burden: the domain framework will mostly be invisible to developers. Second, building a domain framework on top of existing components is not really a new kind of task, simply an extension of the already familiar task of condensing repeated code into reusable functions and components. Finally, as layers go, the domain framework is generally thin. Initially it can be completely empty, and you add to it when necessary, prompted by needs arising from building the generator.

12.1 REMOVING DUPLICATION FROM GENERATED CODE

In many ways, DSM is all about removing unnecessary duplication of effort, and in software development this generally translates into removing unnecessary duplication of manually written code. The generator plays a central role in this process, replacing many instances of copy–paste duplication with one copy of the relevant code in the generator. Even without DSM, component libraries have played an important part, allowing a repeated piece of code to be collected into a component and invoked each time via a significantly shorter piece of code. A domain framework makes it possible and cost effective to go further than normal in this area. Let us look first at a concrete example of how the economy of reuse changes in DSM.

The Windows call to open a “Save As” dialog effectively takes 23 parameters, one of which is any combination of a set of 27 flags (Fig. 12.2). This has two consequences: the number of different possible dialogs is astronomical and the code to produce any one of them is substantial. Any given application will however generally contain only one or a few such dialogs, so the code for the dialog is generally written by hand—or more likely copied and pasted from last year’s project. For many programmers, the number of calls to such a piece of code is simply not high enough to persuade them to create their own higher-level dialog function.

The code for a “Save As” dialog would thus be likely to be found as a fairly long boilerplate section in a DSM generator. Some features of the dialog may be picked up from the model and used to affect the generated code, either being output to become

BOOL GetSaveFileName(

LPOPENFILENAME *lpofn* **Flags**

lStructSize;

DWORD

hwndOwner;

HWND

hInstance;

HINSTANCE

LPCTST

lpstrCustomFilter;

LPTSTR

nMaxCustFilter;

DWORD

nFilterIndex;

DWORD

LPTSTR

nMaxFile;

DWORD

lpstrFileTitle;

LPTSTR

nMaxFileTitle;

DWORD

lpstrInitialDir;

LPCTSTR

lpstrTitle;

LPCTSTR

**Flags;**

nFileOffset;

WORD

nFileExtension;

WORD

lpstrDefExt;

LPCTSTR

lCustData;

LPARAM

lpfnHook;

LPOFNHOOKPROC

lpTemplateName;

LPCTSTR

pvReserved;

void \*

dwReserved;

DWORD

FlagsEx;

DWORD

}

OPENFILENAME, \*LPOPENFILENAME;

From MSDN Library, Win32 and COM development,

©2006 Microsoft Cor

p

oration.

typedefstructtagOFN{

DWORD

This member can be a combination of the

following flags.

OFN\_ALLOWMULTISELECT

OFN\_CREATEPROMPT

OFN\_DONTADDTORECENT

OFN\_ENABLEHOOK

OFN\_ENABLEINCLUDENOTIFY

OFN\_ENABLESIZING

OFN\_ENABLETEMPLATE

OFN\_ENABLETEMPLATEHANDLE

OFN\_EXPLORER

OFN\_EXTENSIONDIFFERENT

OFN\_FILEMUSTEXIST

OFN\_FORCESHOWHIDDEN

OFN\_HIDEREADONLY

OFN\_LONGNAMES

OFN\_NOCHANGEDIR

OFN\_NODEREFERENCELINKS

OFN\_NOLONGNAMES

OFN\_NONETWORKBUTTON

OFN\_NOREADONLYRETURN

OFN\_NOTESTFILECREATE

OFN\_NOVALIDATE

OFN\_OVERWRITEPROMPT

OFN\_PATHMUSTEXIST

OFN\_READONLY

OFN\_SHAREAWARE

OFN\_SHOWHELP

OFN\_USESHELLITEM

lpstrFile;

lpstrFilter;

FIGURE 12.2 Parameters to configure a Windows “Save As” dialog

parameters to the call, or being used as conditions in the generator to cause different fixed text parameters to be output.

Within any given product built with the DSM language and generator, this solution will look all right: the code will probably appear only once or twice. Across the whole product family, however, the code will appear many times. While this duplication also occurred in handwritten code, it was spread over so many developers and so much time that developers rarely felt a pressing need to refactor it.

In a way, the generator already provides a good refactoring: the boilerplate code appears only once in the generator definition, and all occurrences in the code are produced automatically from that. However, this still leaves many occurrences in the code. The time-honored principle of avoiding duplication is thus not yet fully applied. The principle is so well proven, and such an integral part of DSM, that a deep costbenefit analysis is probably unnecessary: a few examples should suffice.

In an embedded product, the extra code size may be an issue, particularly where a whole product is built up of many smaller DSM models. In a distributed project, coherence of code generated with different versions of the generator may be a sticking point. In a multiplatform project—even just over several versions of Windows—the code to open the dialog would vary on each platform, so moving the variability from the generator to the framework would simplify adding new platforms. In all cases, Occam’s razor applies: source code explains to the computer what it should do, and the fewer words we can do that in, the better.

### HIDING PLATFORM DETAILS

We would thus want to look at refactoring that repeated block of code out into its own function in the domain framework. The function would set up some desired defaults for the majority of parameters, and have its own parameters for a few features of the dialog that vary. The generated code will thus be much shorter in each case, easier to read, and easier to relate to the data in the model. At the same time this approach will also harmonize the use of file dialogs, removing unnecessary variation—necessary variation will of course be captured in the parameters and specified in the models.

12.2 HIDING PLATFORM DETAILS

Modern software development is based in a large part on existing general-purpose components, often supplied as part of a development platform such as Java JDK or Windows .NET. After a platform’s early days, there is an implicit assumption that these components will be accurate, bug-free, and offer the current interface for the foreseeable future. There is also often the implicit assumption by the developers that the code they are working on will continue to use the same platform. Sadly, neither of these assumptions seems particularly well founded.

If the operating system, platform library, and SDK we rely on cannot, in the end, be relied on, how can we reduce our dependence on it? One way is to create a new framework: cross-platform, open source and hence bug-free (or at least if there is a bug you can quickly correct it), and switch to using that. Irony aside, this is indeed a popular approach, as can be seen from examples such as GTKþ or the lower-level Cairo Graphics library. However, to be successful, such projects need a large number of participants with different platforms and needs, and require a number of years to complete. While most of them never make it, the ones that do are a great addition to the tools available to software developers.

Another approach is to attack the other end of the problem: rather than trying to make a new implementation framework that is more generic, make a new framework layer that is more specific to your problem domain. The interface of this domain framework upwards to your generated code can remain the same, and only the interface down to the components needs change with the platform.

12.2.1 Bypassing Bugs and Platform Evolution

We saw an application of this approach in the Watch example in Chapter 9. At the start of the implementation, we were using JDK 1.0 for maximum compatibility. We quickly discovered a number of bugs and significant missing functionality in the Java Date class. Looking at JDK 1.1, some of these bugs were corrected, but new ones had been introduced. Despite the addition of extra Calendar, TimeZone and DateFormat classes, java.sql.Date and java.sql.Timestamp, there were still large areas of functionality missing. In particular, we needed the ability to add and subtract times as both points in time and periods of time—a fairly standard requirement. A more domain-specific requirement was to increment a single unit of a time with rollover, as watches do when editing the time. For instance, 59 seconds would be followed by 00 seconds, without incrementing the minutes.

Normally we would have implemented our own calls over the top of the existing Java Data class. In this case, its poor quality, lack of stability, and focus shifting away from simple time toward esoteric calendars and time zones led us to create our own METime class. This class interfaced only with low-level platform primitives like the system millisecond clock and modular integer arithmetic. While as novices in both Java and the time domain, writing this class required more effort than it should have, it was still completed in around half the time we had spent fighting with the Java Date class.

The main value of METime is that it has remained unaffected by the significant changes to time and date handling in Java over the subsequent JDK versions. It also helped us to cope when the domain revealed surprising complexity. For instance, a World Time application works so that it shows the current local time plus a userdefined offset. When editing the World Time, the time displayed is the local time plus the offset, but the changes made while editing only affect the offset. Time units thus roll over when the local time plus the offset reaches the maximum value. If the offset is þ6 hours, the local hours value equals 17, and the user increments the hours, the displayed figure must decrease from 23 to 0, but the offset must increase to þ7; not for instance to 17, which has the same value modulo 24, but a very different effect on any existing timer alarms that might be counting down.

12.2.2 Extending a Framework from One Platform to Many

As we saw in Chapter 9, the Watch example was designed and built purely to run as a Java applet in a standard desktop OS. When we decided to extend it to work as a Java application in a cell phone, many of the initial assumptions were no longer true. We could no longer use text input fields to show the time, nor buttons that the user could click with a mouse. The main application class would no longer be an Applet, but a Midlet, which worked rather differently. Compilation would no longer be as simple as “javac \*.java,” but would require a number of new phases and configuration files.

Perhaps most challenging, we wanted the same models to be able to generate an application for either platform. Or rather even more: we did not just want to be able to get to a state where one set of models could generate both, we wanted the existing models, built before any idea of MIDP was considered, to work on both platforms. In a real commercial case, this would have saved the modelers (of whom there would be many) having to rework each of their models. While achieving this may require more work in the DSM solution (the metamodel, generators, and framework), the benefit would be multiplied over many modelers, each with many models.

Of course, if the claims of DSM are true, and if the Watch DSM solution had been made according to good DSM principles, this requirement should be feasible. The watch models would contain only information about what the applications should do, and no implementation details of how they would do it or on what kind of platform. Happily, it turns out this was the case, and the results are seen in Fig. 12.3. You’ll just



FIGURE 12.3 Watch in Nokia, Motorola, and Sun MIDP emulators, and browser applet

have to take our word for it that the Watch example really was our first shot at making a DSM example to accompany MetaEdit+, and not for instance the only one out of 27 such examples that we could actually get to work!

You may recall that the authors of the existing framework, while experienced programmers, were Java novices. The original codewas thus unsurprisingly in need of refactoring, and the new platform provided a good reason. First, we refactored out the mass of user-interface, control, and state machine behavior into their own classes. From this, it was easier to see what had to be done.

The majority of classes were platform independent, requiring only basic Java functionality. The user interface and control Application Program Interfaces (APIs) are different for MIDP, with only very basic widget support. The widgets for displaying the time and icons were thus replaced with a lower-level set of text drawing operations in a new WatchCanvas class. As the text and icons had to adapt to different MIDP devices’ fonts and screen sizes, it was soon noticed that a similar WatchCanvas class could also replace the old widgets in the applet version. This resulted in smoother updating in the applet, as well as keeping the applet and MIDP versions more visually similar.

12.3 PROVIDING AN INTERFACE FOR THE GENERATOR

The generator takes input in a format specified by the modeling language, and produces output that must run under the domain framework. Since both modeling

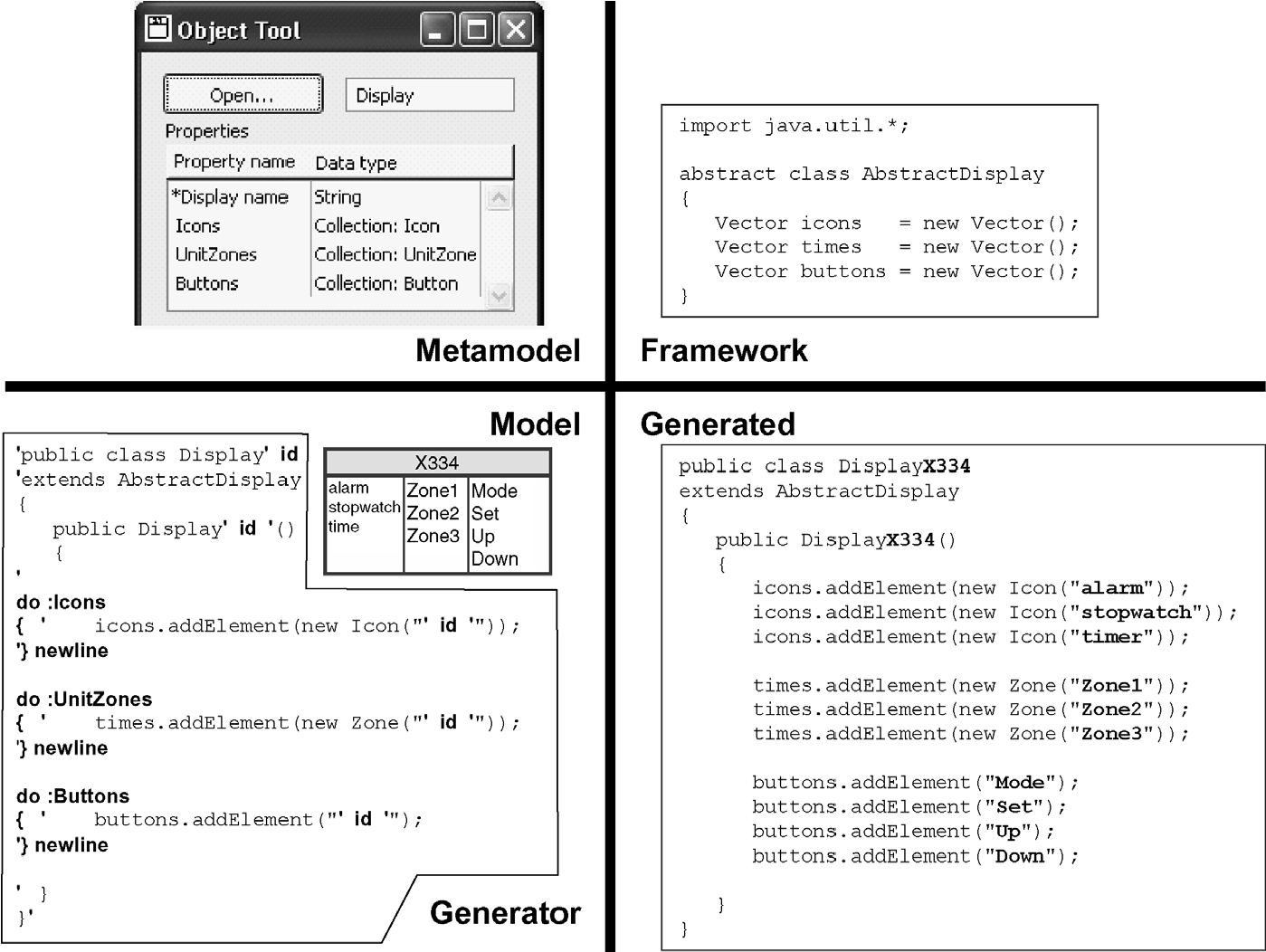


FIGURE 12.4 Parts of a DSM solution

language and framework are created specifically for the same problem domain, there is often a good correspondence between them. The better the correspondence is, the easier it is for the generator to map from its input to its output. In Fig. 12.4, there is a good correspondence between the Display modeling concept in the metamodel and the AbstractDisplay class in the framework. The three lists from the model can thus map one-to-one with the lists in the generated code: the only change is that the modeling language calls the middle list “UnitZones,” whereas the Java class calls it “times.”

An exact one-to-one correspondence may not however be desirable in all cases: It leads to a domain framework that has to be completely driven by data structures that mimic the model. While completely data-driven code will work, it requires a large amount of work in the domain framework, can be harder to follow, and is almost always slower than more traditional code. Parts of the generated code may well be data-driven, for instance, the basic structures of a state machine. The Watch example in Chapter 9 used a data-driven approach for the Java state machine, but more procedural code for the C state machine. As a general guideline, the lower the level of the target language you are generating, the less likely it is that you will choose a data-driven approach.

How can the domain framework offer an interface for the generator to use? Largely in the same way as any component framework: it can provide utility functions, data structures to be instantiated, ways to attach behavioral code generated from the models, customizable and extensible components, and an architecture that works with all these. Utility functions will be familiar from other frameworks, and were covered in Section 12.1, so we will concentrate on the other ways.

12.3.1 Data Structures to be Instantiated

In many cases at least some of the modeling language concepts will appear in the domain framework as data structures with similar contents. This is obviously the case for a generator for a modeling language that offers a graphical front end for an XML schema. There each object type will tend to map to an element with attributes or subelements for each property, as we saw in Section 11.3.4. Generation of other textual DSLs will work similarly. Even where the generator produces mostly executable code rather than data, some parts of the model will normally be represented directly in data structures.

In an object-oriented target language, these data structures will be instances of classes defined by the domain framework. The class structure will mirror the structure of the corresponding type in the modeling language. Properties will be mapped into member variables of the class, and some of the relationships and similar links to other model elements may also become member variables. Relationship types themselves may sometimes map to their own class, with the main member variables holding the objects they connect, and extra variables for the properties of the relationship and roles.

Whatever the type of the target language, such data structures will have associated behavior. Perhaps more commonly than in traditional frameworks, an important part of that behavior will be the accessors and constructors. The model contains data in a format that is easy for a human to work with, but the code that will use these data structures will want to access it in a machine-friendly format. That transformation can be accomplished either by the generator, the constructor and/or setter accessor, or the getter accessor.

In all but the simplest cases, the generator would be a poor place to carry out this transformation: it is better to keep the code that produces the internal format in the same place where that format is defined. If performance is not a major concern for the data in question, it may be easiest to keep the internal format similar to the human-readable format and provide machine-friendly getters. This will keep the data structure definition in close correspondence with the modeling language, and keep instances in a format recognizable to the modeler (e.g., if he has to resort to source-level debugging). Should performance constraints arise later, the variable in question can be changed to a machine-friendly format. If there is a strong need to maintain the human-readable format, an extra variable can be added to cache the machine-friendly format.

Where performance is at all a concern, or when that area of the DSM solution has become stable and is no longer being changed, the domain framework can offer a constructor or setters that take the human-readable format as arguments, and fill the machine-readable format into the data structure. In most cases, such data structures are read-only after construction. If they will be modified later by the framework—for example, if the model is simply specifying an initial set of values—there should also be direct setters, which would take an argument in the machine-readable format.

12.3.2 Integrating Code Specified by the Model

Only in simple cases can a framework be turned into an application purely by configuring it with data from models. More often, there is a need to specify new dynamic behavior per application. We have seen a number of ways of building modeling languages to capture such information, and in Chapter 11 we saw several approaches for turning that information into code. Now we must consider how the domain framework can best integrate such code.

First though we must address the myth that information can be divided into data and code. All too often we hear skeptics say: “sure, you can generate that part, but that’s just data: what about some real code?” The truth is that there is a sliding scale of increasing complexity, with stereotypical data and stereotypical code at opposite ends. Good developers will generally work to push things toward the simpler, datalike end of the spectrum, and thus reduce complexity. In many cases we have seen in industry, the “need” for real code has in fact simply been evidence of a failure to take a step back and look at the wider picture.

To take a more iconoclastic view of things, all code is in fact data, used to configure some framework or engine. C code is the fodder for the compiler: one long string, or after parsing a simple tree of a few keywords with primitive string or integer values as leaves. Database guys can snicker at developers as they point out that even an online pet food store might have more complicated data structures. Going further down in search of Code For Real Men works no better: machine code is even simpler, and the processor just chews that stuff up like a 1960s BASIC interpreter. In a last attempt to salvage some pride, the Real Developer pulls out self-modifying code; the database guy looks on with pity: “so you finally have some data that’s not read only. Of course, we’ve never seen anything like that...”

Clearly, the search for a dividing line between code and data is fruitless—and not particularly flattering to our egos. Let’s abandon any attempt to find a higher truth here, and settle for a pragmatic definition. If a piece of generated text simply instantiates and provides values for a data structure, it’s data; otherwise, it’s code. How can a framework provide ways to integrate such code? Obviously, if the code calls the framework, the problem is trivial, so we shall concentrate on cases where we want the framework to call the code.

First, the framework can be made to call a specific function, and the generator produces that function. This will normally of course be extended to a number of such functions. An example can be seen from the “perform” function used in the Watch applications from Chapter 9, as seen in Fig. 12.5. The

AbstractWatchApplication framework class implements a state machine, calling the “perform” method for the actions specified in transitions between states. It defines “perform” only as an abstract method. The code generated from the Stopwatch model is a subclass of that, and provides a concrete implementation of the “perform” method

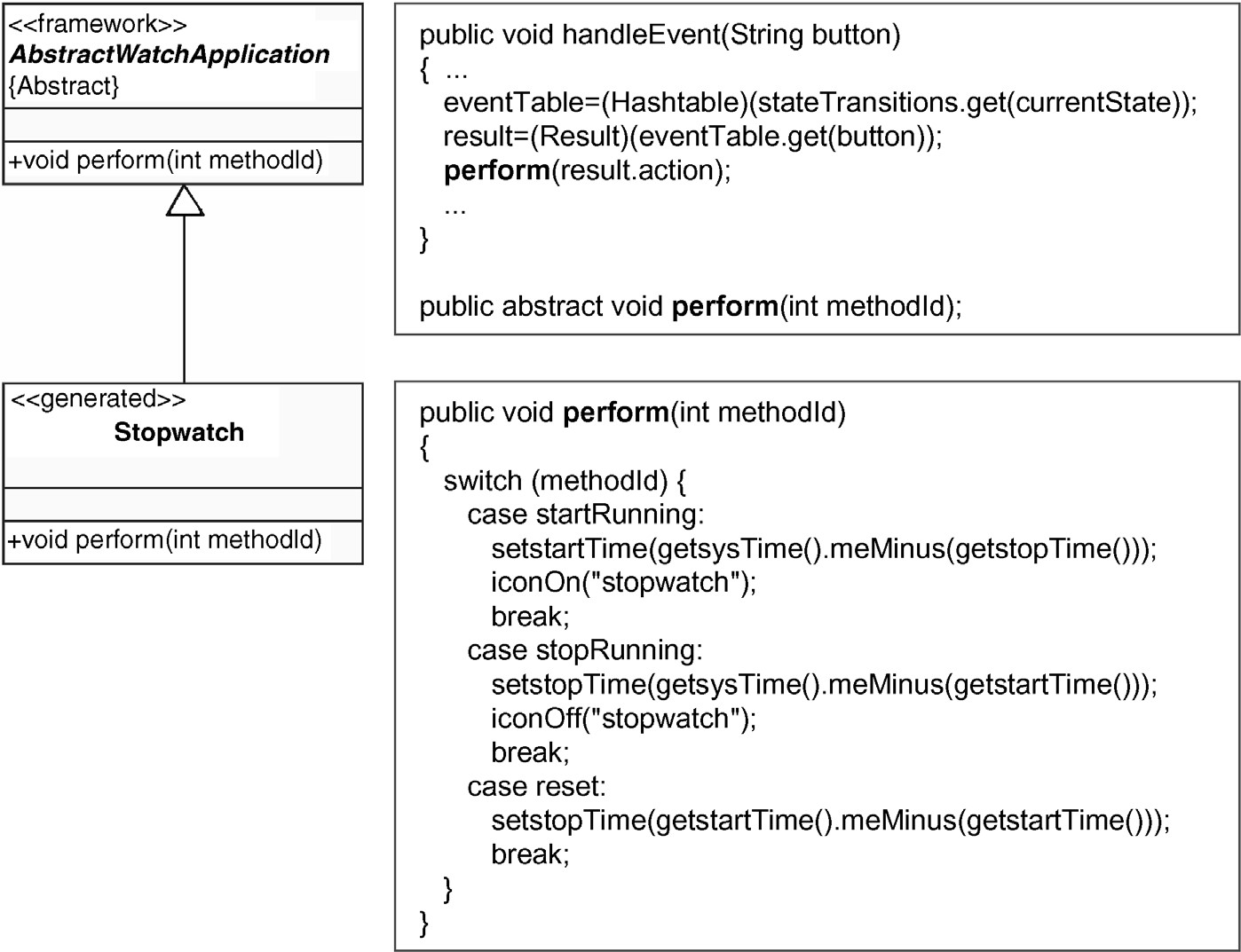


FIGURE 12.5 Framework calling a generated function

to specify its own actions: setting various time variables and turning the stopwatch icon on or off.

Second, the framework can provide hooks, events or callbacks, and the generated code can declare its generated functions as handlers for these. This allows a large degree of freedom and variation in how the application takes advantage of the framework. In third party component frameworks, such an approach demands good judgment and architecture from the framework provider: all too often the point at which you would need to step in and have your code executed is not offered by the framework. In a domain framework, that is easily remedied; a good architecture is however still needed.

Where the implementation language permits it, the second way can also be extended far enough that it could be considered a new, third way. The framework can offer data structures, some of whose members are pieces of code. Some languages, for example, C, allow named function pointers to be used in this way; others such as Smalltalk allow arbitrary anonymous blocks of code. Java can use inner classes, although their syntax and the multiplicity of compiled files leave something to be desired. Fig. 12.6 shows an example of inner classes being used to accomplish the same behavior as the overriding of the “perform” function above.

A fourth way to integrate generated code into the framework is via direct injection: changing the framework code itself. This is only viable where only one piece of the

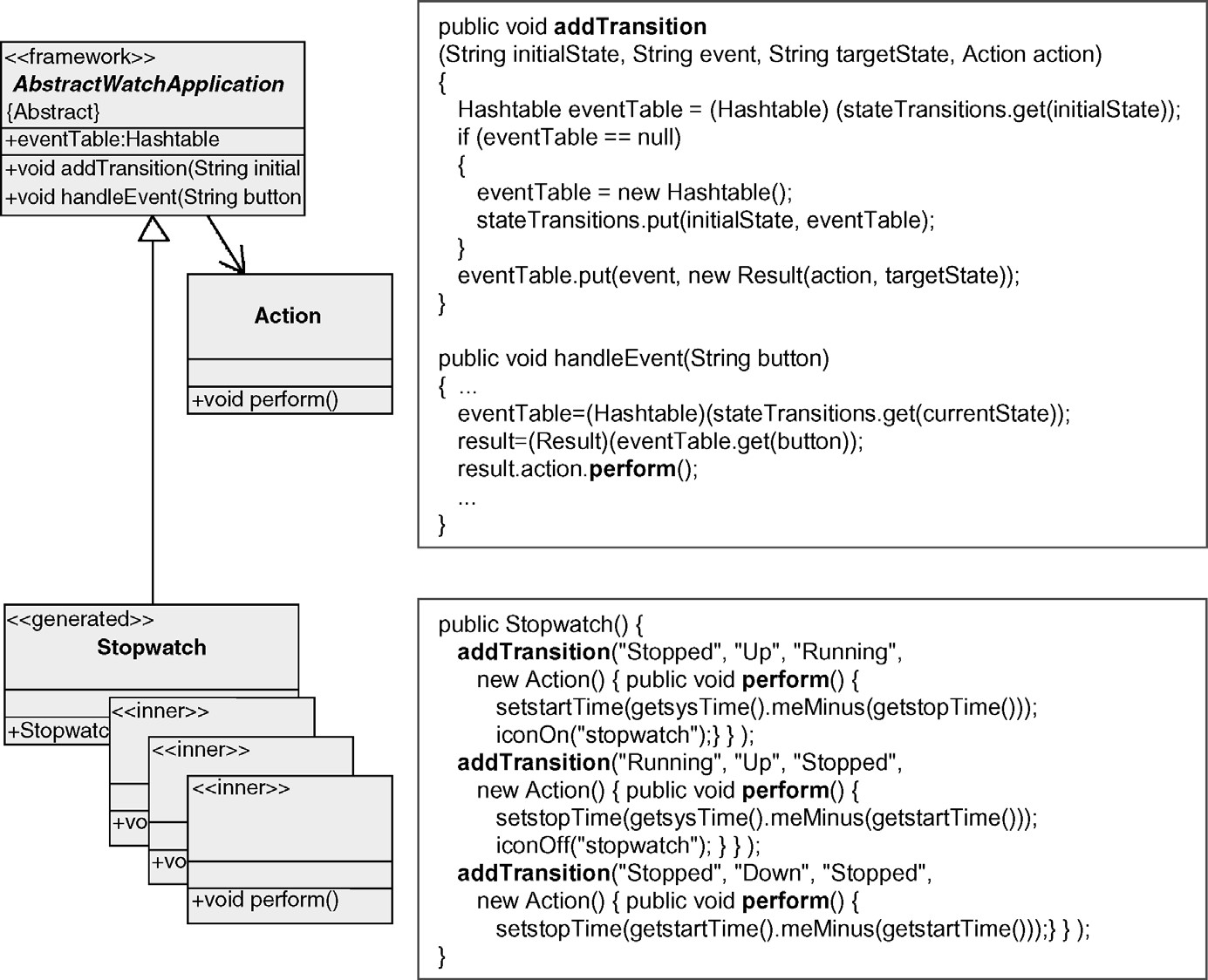


FIGURE 12.6 Generated function hooking itself into framework

application wants to change any given piece of the framework. If multiple conflicting changes were needed to the same piece of the framework, there would be no way to reconcile them to satisfy all the requirements. While the framework could in theory be duplicated for each case, making the application contain multiple edited copies of it, in that case it could hardly be considered a framework.

We have seen cases where a framework is literally edited via a script or macros, but this is best saved as a last resort (or for those who find themselves unaccountably short of war stories). Some languages, for instance C, offer a preprocessor which allows the definition of macros. The framework can then contain references to the macro, and the generator defines its code into the macro definition in a separate file. Aspect-Oriented Programming systems allow the use of separate code files which are then woven into the framework at specified points. Java is particularly strong in this field, in part because of the language’s tendency to forbid many tricks commonly used to influence already compiled code. C# allows the use of partial class files: the framework could contain the main class file, and the generator could produce an extra partial class file for that class, adding new behavior.

These approaches are best used to provide reasonably limited amounts of code to the framework. Should you find the need to add significant amounts of code, or add code that also defines new or extended data structures, you will probably want to move on to the next section.

12.3.3 Customizable and Extensible Components

In many object-oriented component frameworks, a large share of the workload is shouldered by abstract classes in the framework. The developer building on the framework uses and extends these classes by subclassing them. This allows the full range of behavior customization, extension, and modification:

. A subclass can fill in data structures defined in the superclass.

. A subclass can provide concrete implementations of methods defined as abstract in the superclass.

. A subclass can add new data members for its code to use.

. A subclass can add new methods for its code to use. . A subclass can override the accessors defined by the superclass, to retrieve data from elsewhere or manipulate the values returned.

. A subclass can override other methods defined by the superclass, to change its behavior.

This can all of course be carried out on a number of levels:

. An abstract class in the platform or component framework can be subclassed by generated code.

. An abstract class in the platform or component framework can be subclassed in the domain framework to provide the commonalities required by generated applications. The result could be a concrete subclass that is instantiated and used by generated code, or then itself an abstract class to be further subclassed by generated code.

. If parts of the model reuse other parts of the model, this may correspond to generating subclasses of generated subclasses.

As can be seen, this allows a vast array of different approaches and tactics, allowing you to produce applications in almost any way you want. This is of course good news for you the reader: you can generate full code without it becoming bulky (move common parts to the framework) or ugly (replace hacks with a good architecture in the domain framework). It is however bad news for us as authors: with such a broad vista of possibilities, it is hard to cover the ground or give concrete advice. Fortunately, at this level building a domain framework is similar to building any other kind of components, so your existing experience will be valid and useful, as will any good books on object-oriented programming.

If this is your first foray into DSM, you may well have found many other previous beliefs about standards, modeling, and code generation being challenged. Hearing that there is an area of DSM where things are pretty much “business as usual” will no doubt come as something of a relief! Without wishing to disturb you from your reverie, we do want to provide a few signposts to small ways in which the domain framework and generated code may be a little different from what you are used to.

GeneratedCode Contains More SingletonClasses When you look at code generated to run on a fair-sized domain framework, you may well notice a larger than average number of singleton classes. They will be subclasses of a domain framework class, often in quite a wide inheritance tree, and each may add only a small amount of code. This may appear disconcerting: if you saw the same in handwritten code, you would look suspiciously at it and try to apply various best practice strategies. What could be moved up into the superclass to allow that single class to be used for all of the instances? Or which of the subclasses was really the same as another? Or perhaps the subclasses could be arranged into a deeper hierarchy, with extra abstract classes containing the features shared by several subclasses?

In fact, assuming the framework and generator have been made well, you are actually seeing the end result of thosevery same strategies—just applied far more than normally happens with handwritten code. The subclasses look small because every last drop of commonality has been squeezed out of them into the superclass. None of them will be the same, because in that case the modeler would have reused the corresponding model element rather than creating a duplicate. Finally, the amount of common content in groups of subclasses would be so small that the weight of an extra class would normally not be justified.

The last two points need a little clarification: at any given state of the models, there may well actually exist complete duplicates or sets of classes with common elements. However, the duplication will be purely coincidental, and could change at any time. The duplication is more like that found when two Person objects have the same height and weight: it does not mean we need to separate that “commonality” into a new HeightAndWeight class, with both Person objects pointing to the same instance. Of course, there will be cases where you missed something in the domain analysis, and you really should make the granularity of reuse finer in the models. In our Person analogy, that might correspond to noticing that groups of people tended to share the same surname and address fields, leading you to posit a Family class.

Finally, there may be cases where the singleton classes only differ in name and the values they give to static variables. If that is the case, and the classes really are singletons, it will probably be an implementation choice whether to simply move the name and the static, class-side variables to be instance variables. Examples of this can be seen from the Watch Java code in Fig. 12.4: do we really need a new class DisplayX334, or could we simply change its superclass, AbstractDisplay, so it could be concrete?

Domain Frameworks Contain More Metaprogramming Since most bugs are found in code not data, good programmers generally look for chances to turn code into data. In simple cases this can be easy and natural, but in less obvious cases the code needed to read and enact the data becomes complicated. The use of a programming language with good support for metaprogramming can help a lot here, but still the code requires an above-average programmer. Unfortunately, the vast majority of programmers consider themselves “above average,” just as most automobile users consider themselves better than average drivers ... To put it another way, in the right hands metaprogramming is a powerful tool, but wielded inexpertly it can wreak havoc with the readability and quality of your code. For this reason, frameworks are a particularly good place for metaprogramming: it is fair to expect the developer of a framework to be a better programmer than the users of that framework.

DSM offers particularly fertile ground for metaprogramming. First, special cases are particularly irksome in metaprogramming. In normal code, an extra IF statement is neither here nor there, but in the compact yet convoluted coils of a piece of metaprogramming, any extra complication can easily render the whole hard to comprehend. Since DSM has already forced a more thorough domain analysis than normal, such cases can be taken into account in the modeling language or elsewhere, or at least will be known from the start rather than hacked in as an afterthought. Second, metaprogramming—like any data-driven code—is sensitive to illegal or wayward input in its data. The metaprogrammer must either write extensive code to check the data or then crank up their optimism a notch or two. In DSM, the structures and constraints of the modeling language can ensure that only well-formed and valid data can be entered.

12.3.4 Inversion of Control

A common pattern when using frameworks is to switch the basic execution architecture: rather than the application code running the show and calling framework code as needed, the framework itself runs and calls the application code provided to it. This is known as inversion of control.

The way the inversion works can perhaps be explained by analogy with sports team coaching. In standard component development, the job of the framework developer is just to provide a set of components for the developers to use as they see fit. Picture a naı¨ve—or progressive—coach offering his team a selection of balls, goalposts, and a goodly supply of the apparently indispensable traffic cones, and letting them get on with things. When a new, more experienced—or authoritarian—coach appears, she runs things very differently: she leads the players through a series of drills, setting out the equipment to be used for each. In other words, she is saying to them, “I don’t just have the stuff you need; I really know how you should use it to get the best results”.

Obviously, this approach demands less intelligence of the team members, but more obedience. This perhaps explains why it is not a universally popular approach among framework users: we developers are not exactly renowned for our subservience, or for taking lightly any perceived slights on our ability. It is, in fact, an approach better suited for use by factory workers, drones, machines—or generators. A good generator might at best be described as smart, but it could never be accused of intelligence: the ability to come up with good, new, solutions to new classes of problems. It is however totally obedient and predictable, and copes remarkably well with the lack of intellectual stimulus in its working day.

Building a domain framework with inverted control can start right from the beginning of the code-related phase of DSM. Back in Section 11.1 at the start of the Generators chapter, we saw how writing a small example application in the domain was a useful first step. The simple approach shown there reduced this application into a generator that produced most of it, and a model that supplied the values that varied between applications. Later, we saw how frequently encountered or long blocks of code in the generator could be separated out from the generator into components. However, this clearly leads to a situation where the generated code is calling the shots, invoking components as needed.

To invert control, we can look at the example application from a different point of view. What parts of it show the flow of control and execution architecture that will be common to all applications in the domain, or to an important subset of such applications? If a working generator has already been built, it may be easier to spot these parts by comparing several generated applications. You can also look at the generator itself, in particular for any pieces of boilerplate code longer than a line or two, and which are generated only once or a few times for each application. Such parts can also appear in the generator as a larger structure of code that remains the same for all applications, with short one or two line segments of boilerplate interspersed with the details filled in by the model.

For instance, in the switch–case pattern described in Section 11.3.6 there is a clear pattern of two nested layers of switch–case statements to implement a state machine. The structure always remains the same, with the models simply providing different states, events, actions, and transitions. This was also the approach used in the C generator in the Watch example in Chapter 9. To invert control, the underlying structure of the switch statements—see what state we are in and what event occurred, then do the requested action and follow the transition to a new state—is moved out of the generator into the domain framework. The generator simply supplies the definitions of the states, events, and transitions as data, and probably implements the actions as new functions. The framework runs its generic state machine over the data supplied by the generator, calling the generated action functions where necessary. This is the approach seen in the Java generator in the Watch example.

12.3.5 Engines Reading Models as Data

We can also take inversion of control further, to the point where the framework becomes an engine. In this approach, all possible behavior of applications in the domain is covered in the engine. Any given application specifies its subset of this behavior space as data, which is read and acted on dynamically by the engine. In a way, the engine is an interpreter for the data supplied by the models, as opposed to the earlier cases where the generator acted more like a compiler to turn the models into code.

Theoretically, the engine could read the models directly from the memory structures in the running modeling tool. Since this requires the modeling tool itself to be present at runtime of the applications, it is hardly ever used for deployment. It may however be useful in some cases for simulation: the engine runs the models directly, and interacts with the modeling tool to visually trace the execution in the models. This kind of visual trace is however also possible with more traditional generation.

The engine could also read the models directly from the files they are stored in on disk. In practice, the model files will normally contain a substantial amount of information that is redundant for the execution, for example, the layout coordinates of each model element and various documentation fields.

### SUMMARY

More normally, the models are exported by generators into code that instantiates framework data structures to recreate the relevant model structures, or into a simple file format that is easily read by the engine.

12.4 SUMMARY

The domain framework is a layer of code that sits between the generated code and the existing generic components and platform. Its main function is to avoid repetition and complexity in the generated code and also in the generator.

The generated code thus comes entirely from models, and the framework is entirely handwritten. This separation distinguishes DSM from code generated by wizards, where the application developer is expected to work with and around the generated code, and code that is common to all applications in that domain is duplicated into each application. In DSM, the application developer can ignore the generated code, and the domain framework is referred to by that code rather than being copied piecemeal throughout it. The separation thus naturally follows the separation of the different developer roles.

The domain framework also separates along the lines of generalizability. Below it are generic components, used by a wide range of applications on that platform. In the domain framework itself are the elements common to all applications in this narrow domain: the envisaged application space for this DSM solution. Above it is the information specific to individual applications, that is, what distinguishes each of those applications from each other. Building a domain framework is probably the part of DSM that will be most familiar to developers from their previous experience. It is similar to building components and more general frameworks: by no means an easy task, but certainly a skill that can be learned. The main difference between building a domain framework and more traditional frameworks is the target audience. The “users” of a domain framework will be the generator and its generated code, rather than developers and their handwritten code. This solves a problem commonly experienced by builders of traditional frameworks: they make a wonderful framework, but people often misuse it—or more often fail to take advantage of it or parts of it.

As the use of the domain framework is automated in the generator, domain frameworks are used more consistently than traditional frameworks. This makes it possible for them to go further than normal in removing duplication. For human developers and traditional frameworks the cost of one developer learning a framework feature must be recouped over the number of times that one developer uses that feature. In a domain framework, only one developer need learn that feature, applying it in the generator, yet the cost will be recouped over all developers using the framework. Often, the framework developer will also be the generator developer, so even that small cost becomes almost zero.

The domain framework also separates thegenerated code from the platform. It hides the platform details, and thus insulates the rest of the DSM solution from changes in a platform. The insulation works equally well whether the change is to a new version of the same platform or to a new platform altogether. This can also be applied when there is a desire to support several platforms in parallel: the same modeling language, models, and generators can be used for all platforms, with just a new version of the framework being written for each platform. As the size of the framework will almost always be smaller than the amount of code that would be needed for a single application in that domain without DSM, the cost of supporting an additional platform will be comparable to the pre-DSM cost of building a single application. Furthermore, whereas without DSM building a single application on the new platform would offer precisely one application, with DSM every single application built for the previous platform will now also be available for the new platform.

While it is good to be aware of such possibilities, the main focus for the domain framework developer will be elsewhere. As in any framework, the domain framework should provide the algorithms and functions required in that domain. In addition to those, the domain framework should focus on matching well with the modeling language where possible, and on providing a good interface for the generator. The interface will consist of data structures to be filled in, places to add code generated from models, and components to be extended by generated code. The aim is to support the largest possible extent of the desired application space, for the smallest amount of work for the modeller—and to a lesser extent, the metamodeler.

Domain frameworks tend to differ somewhat from traditional frameworks, for instance, in greater use of inversion of control and metaprogramming. There is also often a tendency toward small singleton classes in the generated code using the domain framework. Depending on the domain, there may also be other considerations such as the framework size, generated code size, or total executable size or speed. These will influence the strategies and programming style used in the framework, but in a way already familiar to any experienced developer in that domain.

Building the domain framework can happen at any time in the overall DSM process: as the first step, after the modeling language, with the generator, or after the generator. We have seen and worked on successful cases in all of these different ways. Until you have the experience of a couple of successful DSM solutions under your belt, however, we would recommend that you leave the framework until after the first version of the generator. The main difficulty for people in stepping up to DSM is to leave the code world behind when building the modeling language. Since the benefits of DSM are largely due to the raise in the level of abstraction in the modeling language, it is best to avoid the risk of a familiar area such as the framework dragging the level back down toward the code.

Similarly for the generator: take the time to learn the new mindset required for using a generator language, rather than trying to take a “shortcut” by doing everything in the familiar language of the framework. After all, the generator language itself is (hopefully) domain-specific, honed for the task of generation. So, do not be in a hurry to return to the familiar world of code: DSM involves learning some new things, but a little abstinence from coding will whet your appetite for it. At the proper time, building the framework will more than satisfy that appetite, as you get to use all the skills of your trade. Best of all, as you update the generator to take advantage of the framework, your framework will for once have the discerning user you have always longed for.