### CHAPTER 13

## DSM DEFINITION PROCESS

In this chapter, we will look at the process of creating, introducing, and evolving the whole Domain-Specific Modeling (DSM) solution. Earlier, Chapters 10–12 have focused on individual parts of the solution, and the case studies in Chapters 5–9 have described what happened in specific instances. Here, we will give an overview of the process as a whole and also look at the various groups and roles involved.

The first steps are to identify a good area in which to apply DSM (Section 13.1) and assemble a team to carry out the work (Section 13.2). The choice of domain can be tested through a proof of concept implementation of part of the DSM solution (Section 13.3). If all looks good, a full DSM solution can be built for that domain (Section 13.4) and tried out in a real-life pilot project (Section 13.5). You can then polish and update the DSM solution based on feedback from the pilot project and deploy it to your organization (Section 13.6). The deployment will already have involved planning for the evolution of the DSM solution (Section 13.6.3), and this evolution will continue throughout the usage of the DSM solution (Section 13.4).

13.1 CHOOSING AMONG POSSIBLE CANDIDATE DOMAINS

Deciding whether to apply DSM in a given domain should take into account a number of factors. The domain itself has surprisingly little effect: the whole idea of DSM means it is highly adaptable to different domains. Similarly, the target platform is

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

329

largely irrelevant: anything that takes text files as input is fair game, whether it be a C compiler, a Python script, a DBMS, or an application reading a configuration file. In the areas of the domain and the target platform, DSM offers the leverage necessary to overcome any challenges. The broader contingencies of the situation, however, are areas over which DSM has little control. At least in your first DSM project, trying to radically change your market, business, or organization is generally biting off more than you can chew.

For many readers, there will be only one domain over which they currently have control. In this situation, there is limited choice available: as Yoda would put it, “Do, or do not.” The latter part of the quote—“There is no ‘try’”—is equally true in our experience: those who decide to make a real go at DSM, rather than just dabbling, are the ones who also make it succeed.

Other readers may have several domains within their purview. This generally implies a position in a larger organization, either in charge of a number of areas or as part of a research center or advanced development group. In such cases, the chances are that you will first want to try DSM out in one project—both to confirm it works and to be able to apply it subsequently with greater experience in other projects.

In both cases, it is important to understand the wider forces at work in your organization that bear on the introduction of DSM in a given domain. We will look at the forces through the form of a questionnaire, Table 13.1, together with rough scoring and commentary for different answers that will help you rank your candidate domains and the organizational units responsible for them. While by no means as accurate as similar tests for marital compatibility in women’s magazines, this should at least help you recognize the relevant issues and take their possible impact into account. We would like to thank Laurent Safa of Matsushita Electrical Works, Ltd., Japan, for the initial suggestion and draft of such a questionnaire.

13.2 ORGANIZING FOR DSM

DSM does not require a large amount of resources: the focus is on quality not quantity. When the initial DSM solution is in place, all developers’ work will be bounded by, guided by, and reliant on the work done by the initial team. The team must, therefore, be composed of people who really know their work. This is clearly an area where Brooks’ law applies more than most. Adding lower quality resources to the team will have a large negative effect: probably on the initial creation project and most certainly on the use of DSM.

While the quality of the team members must be high, any innate perfectionism must be reigned in, at least in the early stages. It is vital to try the modeling language out in practice, and for this only the core concepts and support for simple examples are needed. Elegance should certainly be aimed for, but if the domain is messy in parts then the best working solution will have to suffice.

The team will need a mix of domain knowledge and programming knowledge. By far the best mix is found in expert developers who have been working in that domain for some time. They have built up their own mental map of the domain concepts, a set of

### ORGANIZING FOR DSM

Choosing a Domain for DSM

How mature is the target business area in your company?

|  |  |  |  |
| --- | --- | --- | --- |
| \* | An established business | þ3 | Clear business concepts, needs, stakeholders |
| \* | Turning an existing customizable product into a platform for use by others | þ1 | Existing demand for product and its customisation; DSM makes this fast and palatable for those who are not in-house experts |
| \* | Development of a new product | 0 | Risk of the unknown balanced by agility stemming from lack of legacy code and processes |
| \* | Creating a new platform for use by others | 1 | Desire to offer familiarity and lowest common denominator works against DSM. Uncertain commitment, risk of project failure |
| \* | A research project | 3 | High risk of failure from other factors, and no commitment to project if it  hits difficulties |

How much are development and core business processes in the target area coupled to external organizations?

|  |  |  |  |
| --- | --- | --- | --- |
| \* | Joint venture | 3 | Serious concerns over equality of commitment and ownership, ability to agree |
| \* | Internal except for third-party components and tools | þ1 | Clear control over destiny, components give stability |
| \* | Totally internal, build all own components and tools | 1 | Possibly indicative of fear of commitment, can succeed only if DSM is truly seen as “theirs” |

How closely is this domain related to other candidate domains?

|  |  |  |  |
| --- | --- | --- | --- |
| \* | Simplest or smallest of a group of several similar domains | þ2 | Good possibilities for extension to other domains later, or switching if this turns out to be a bad choice |
| \* | One of several similar domains | þ1 | Indicative of an important business area |
| \* | Different domains work independently of each other | 1 | Sometimes indicates lack of desire to codify and build on what is known |

How much do you customize or configure your software for each customer?

Extensively þ3 DSM will allow you to capture just the

variable aspects, and code generation will improve delivery times and the quality of the finished product

(continued)

(Continued)

|  |  |  |  |
| --- | --- | --- | --- |
| \* | Only for large customers | þ2 | DSM can integrate many of these bespoke cases back into the mainstream software artifacts, benefiting both large and small customers |
| \* | Often but only in limited ways | þ1 | Possible small language to describe configuration |
| \* | Little or not at all | 0 | Useful only if DSM will be used for the main application |

Do you have good source code examples available, for example for teaching new staff?

|  |  |  |  |
| --- | --- | --- | --- |
| \* | Yes | þ3 | Offers a basis for the generator and is indicative of a mature domain where development is managed well |
| \* | No, but we are always saying we should make some | 0 | Basis exists, DSM can help bring about wanted change, but process will be harder |
| \* | Some available, but out of date and from different times | 1 | Often indicative of a lack of stability and of reworking upon reworking, and hack upon hack. DSM may be an effective way out if there is a strong expert developer |
| \* | No, our staff are all expert developers |  | 3 if said by manager þ1 if said by developer |

Do you have an in-house application framework?

|  |  |  |  |
| --- | --- | --- | --- |
| \* | Yes, with established guidelines, best practices and sample code | þ3 | Clear existing framework, expected code generator output, domain concepts |
| \* | Yes, the previous version is the guideline | þ2 | Domain concepts clear, framework quite easy to build from existing code |
| \* | No, but we try to reuse legacy components | 2 | Would have to define the framework, domain concepts probably also unclear |
| \* | No, we develop everything from scratch | Abort | “First of a kind” development, not worth targeting |

### ORGANIZING FOR DSM

(Continued)

How would you describe the maturity of your software development process?

|  |  |  |  |
| --- | --- | --- | --- |
| \* | Precisely defined and developers must follow it | 0 | May be too caught up in CMM and processes to accept DSM or its subsequent evolution |
| \* | The majority is documented and we generally follow it | þ2 | Shows a desire to take the effort to codify what is known and follow it as long as it is useful: fertile ground for DSM |
| \* | We have a good idea where we are at any given time | þ1 | Developer-led |
| \* | No clear process | 2 | Hard to bring about organizational |

change without organization

Can you assign the following kinds of people to the DSM project?

|  |  |  |  |
| --- | --- | --- | --- |
| \* | One of the top three who built the framework/first product | þ5 |  |
| \* | An experienced developer | þ3 | Can build code generator |
| \* | A small team of normal  developers | þ2 | Vital for piloting, can build example apps to be the input for creating generators |
| \* | No, but you can have a summer intern or two | 3 | May have brains and enthusiasm, but lack domain knowledge and software development experience in large teams and long projects |
| \* | No | Abort | DSM cannot be successful without the developers’ experience |

patterns for mapping combinations of those concepts into code, and a good knowledge of the existing framework and platform. There is also a certain mindset that says it is not enough to be able to churn out good working application code in the domain time after time: there must be a higher level way to express applications and automate their creation. People with this mindset are clearly well suited to the tasks of the DSM creation team. They are the kind of people who build text editor macros, word processing templates, and batch files or shell scripts. Most useful are those who have built such things for use by the whole team, as opposed to only for their own use.

Initial trials with the modeling language can happen within the DSM creation team, using lightweight formats such as pencil and paper or whiteboards. If there is a good existing framework and body of example code, and you have a fast DSM environment that supports modeling language evolution, you can move at an early stage to testing your ideas in practice. One way is to do a short proof of concept project, as described in the next section, and another way is to do an early version of the DSM pilot project

with a DSM use team. Even early pilot users require modeling tool support and normally also full code generation (although manual tweaks will be acceptable at this stage). If building tool support to such a stage would take longer than a few days, leave it for now and concentrate on the modeling language. Alternatively, consider using a faster DSM environment for now, even if later you plan to use one that requires more hand coding. This will allow you to verify your modeling language ideas in practice, before investing the time in coding them.

Take every opportunity to find people to try the DSM solution out in practice as you build it. Creation team members should draw models of small yet realistic situations, and you should press-gang other developers into at least looking at the models and thinking about them. While they may not yet have a formal part in the project, keeping the communication lines open is an important factor in later piloting and deployment. Making it clear within the team and to other developers that you are initially aiming only for the simple, common cases will help keep expectations realistic. It will also help to keep the creation team motivated and focused on delivering real value, so it can be used soon in a real pilot project to benefit developers.

As the modeling language and the code generator come close to being able to make simple real applications, start looking for a pilot project and team. The pilot project should be a real application that is important to the business, in the sense that it would have to be built anyway and will form part of a real product. It should not, however, be on the critical path or be strongly coupled with other projects. Wherever possible, the pilot project should be one that is already well understood: the kind of job that you might give to a new developer.

The key criteria for the pilot team are motivation and discipline. While the team need not be gung ho DSM enthusiasts, they must be willing to give it a go, and able to cope with the setbacks and delays that accompany any new venture. Since the modeling language and generators will be unfinished and changing rapidly, the pilot team must be disciplined in following instructions, updating to newer versions of the language and generators, and reporting the issues they encounter.

13.2.1 Management Support

The first steps with DSM can often be performed within the bounds of existing resource allocations, but full implementation will normally require management support. Since there is usually little chance for a team to select or instruct its own management, wewill restrict ourselves to looking at what can be requested from and expected of the existing management. In the broadest terms, we are looking for support from the management for the process of evaluating and implementing DSM, for as long as DSM appears to be a valid solution for this area. Management support can have different forms: . They propose candidate domains that they feel most fruitful to support with DSM.

. They agree to the implementation of a proof of concept and commit to following through if it proves successful.

. They allocate appropriate resources to the DSM process.

. They want to own the consequences of the findings.

### PROOF OF CONCEPT

The last point is particularly important, if hard to define: the more that management feel personally invested in the success of the DSM project, the more likely they are to allocate the resources needed to make it succeed.

13.3 PROOF OF CONCEPT

Moving an organization from coding to model-based development is rarely so fast that you can accomplish it before anyone notices and begins questioning the wisdom of the project. Many people have preconceived prejudices based on older approaches with fixed modeling languages: modeling is a waste of time, you should only use “standard” modeling languages, generated code is bloated, slow, and incomplete, and so on. You will almost certainly need to face such objections at some point, so you may as well get them out of the way early on. Aside from waving books like this and glossy vendor literature at people, it is also useful to show concrete results right from the start.

A good approach, assuming a sufficiently fast DSM creation environment, is to do a short proof of concept project within your DSM creation team. If you can get someone experienced in DSM, either an external consultant or someone from elsewhere in your organization, this need only take a couple of days. If it is your first DSM project and you have no such mentor, a week would be a reasonable time.

The goal of the proof of concept project is to define and implement a small, but still significant, part of your DSM solution. At the end of the project, you will have a meeting where you demonstrate the results and benefits with a concrete example to your executives and product developers. The result of the workshop is a partial modeling and code generation environment for your organization’s own domainspecific modeling language.

13.3.1 Preparation

Whether you are working on your own or with a mentor, the first phase is to outline the scope of your DSM solution and gather the materials that form its requirements. Table 13.2 shows a suggested template for collecting this information. Providing the answers is a good first task for your DSM creation team, and an effective way of checking that you have the necessary expertise on the team. If you find yourself having to ask others for some of the answers, you should consider whether those people or others with their knowledge should actually be a part of the team.

Set a time for the workshop and arrange a place where your team members can work uninterrupted by normal business matters. Most importantly, arrange a meeting with management and senior developers for right after the end of the workshop. When you need to produce concrete results rather than interminable wrangling, there’s nothing better than knowing that you and your work will shortly be on public display. The aim of the workshop is not to get everything right, but to get something working.

Before the workshop, you should make sure that all members of your team understand the ideas of DSM, that all have seen some real examples of it, and that the majority have at least basic experience with the tools and languages of your DSM creation environment.

TABLE 13.2 Proof of Concept Template

Proof of Concept Workshop

The aim of this document is to collect the key elements of your domain for the purposes of the domain-specific modeling language implementation workshop.

The goal of the workshop is to define and implement a small, but still significant, part of your domain-specific modeling language to demonstrate the benefits with concrete examples for executives and product developers. The result of the workshop is a partial modeling and code generation environment for your organization’s own domain-specific modeling language.

As domains differ widely, not all questions may be strictly applicable in your domain: feel free to change the questions or omit irrelevant parts.

1. Introduction

Please give a short introduction to your domain area. What parts of the work in the domain do you want to include in your modeling language, and why are the benefits of domainspecific modeling important in these parts?

1. Usage

Describe briefly how you intend to use the modeling language: what is the input on which the models are based, who makes the models, what is generated, and what other tools are involved. Where possible, give approximate indications of how many users, models, files, and so on there are, and how much various parts are reused between products, features, or models.

1. Sample material

Pick a small but representative existing example whose implementation would take about one week for one developer. A good example is often the kind of feature you would give as a first task for a new programmer. The example should include the major, most central elements of the domain.

Describe the example briefly here, along with the approximate time to implement it, any notes about special features of this example, things that have changed since it was made, and how it fits in with other related applications. Then continue to fill in Sections 3.1–3.4 with more specific information (attach separate documents where necessary, making links to them from this document).

* 1. Sample requirements

Give the requirements for the example functionality, including how it interfaces with other parts of the system.

* 1. Sample design

Give the design documents, including graphical models and text where available. As far as possible, the design should be in step with the code below.

* 1. Sample code

Give the code for this example. If there are several files then also provide an overview of whateachfiledoes.Wherepossible,commentthecodewithreferencestothedesigndocuments.Thesecommentswillhelpinanalyzingwhichpartsofmodelscouldproducewhich parts of code.

### PROOF OF CONCEPT

TABLE 13.2 (Continued)

3.4 Sample user’s manual

If the code will have a human user present when run, either using or monitoring the application, give user instructions for that person using this code.

1. Rough modeling language sketches

Draw the diagrams below in whatever is the easiest format: hand drawn is fine.

* 1. Product contents

Often a product (or range of products) consists of several parts (features, modules, etc.). Sketch a (partial) typical product along with its parts and subparts. Mark which parts stay largely the same between products (e.g., existing code libraries, components) and which parts are to be modeled in the other diagrams below.

* 1. Product behavior

Sketch a diagram of a typical feature (possibly the sample feature above) as might be drawn by designers on a whiteboard at an initial design session. Try to avoid “standard” design languages such as UML, which concentrate on the resulting code components, and use informal concepts commonly used and understood by designers. Sometimes there may need to be more than one diagram at this level, for example, one for the user interface and one for the behavior.

1. Other relevant material

13.3.2 During the Workshop

The activities during the workshop will unsurprisingly be an application of the material in Chapters 10 and 11, and to a lesser extent in Chapter 12, following a miniature version of the process described below in Section 13.4. The key component is the modeling language, and the key way of judging it is that it captures all the information needed for the sample application, and that by changing some of that information you would be able to describe a reasonable range of different but similar applications. You must resist the temptation to work on the more familiar areas of the framework or code generation, or to spent too much time on the graphical representation of the modeling language.

On the basis of our experience over several dozen such workshops, generally over two days, we can give some rough indications of the timetable. On the first morning, we generally spend about 45 minutes getting to know people and making sure they understand the main principles of DSM. The rest of the morning is spent sketching out various decompositions of the information of the domain in general, and the sample application in particular. This gives us some possible starting points for the modeling language—or, if the domain seems to require it, modeling languages. We look at the code of the sample application to recognize parts that are repeated, common to many applications, or variable depending on the application. At this stage, we are not thinking about code generation but about making sure that the modeling language will be able to capture the information we need.

After lunch, we decide on a sketch of the modeling language and draw a sketch model for the sample application. As we get close to something that seems to work, we start defining the concepts of the modeling language: their names and the information that each has to store. So far, even with the fastest DSM tools, the work has generally been so likely to change in major ways that the most effective representation medium has been a whiteboard or paper. The whiteboard has the advantage that you can erase minor mistakes to keep things readable; flip charts are a little messier but give you effectively unlimited space, as you can tear off pages and stick them to walls.

When the concepts and connections of the modeling language have taken shape, we turn them into a metamodel by inputting them into the DSM environment. For now, the graphical representations of the objects will be minimal: colored geometrical shapes with text fields within them to show the values of their properties. While part of the team is inputting the metamodel, another part often fleshes out the sketch model of the sample application on a whiteboard.

When the metamodel is ready, it is passed off to the model group, who create this first model with the new modeling language in the tool. As they are working on that, the metamodel group start looking at code generation. By the end of the first day, there is always a metamodel and a model, and generally a code generator that produces a significant proportion of the code. Parts of the code produced will still be fixed, rather than varying properly based on the information in the model.

Since as visitors we are generally staying in a hotel for the night, we normally take the chance to work in the evening on things that are objectively less important but, in practice, give a disproportionate benefit: pretty symbols for the modeling language, and slides for the presentation in the meeting the next day. We sometimes do a little refactoring of the modeling language or generators if they need it, but nothing major.

The next morning the whole group gets to ooh and aah over the pretty symbols, but the mood is generally tense. The meeting is scheduled for shortly after lunch, and so far the generators seem far from producing full working code. Looking at the code produced and imagining the time it would take to write the rest by hand and get it working, the team is feeling understandably nervous. Quite often, on the second day some team members have to go off to do other urgent tasks, but the scope for parallel or group work at this point is more limited anyway.

One developer generally works on the generators, running them on the model and comparing the output with the known good application code. A second developer may work on a thin domain framework layer, writing functions to reduce duplication in the generated code. If there are other developers, they may be set to work normalizing the known good application code into the more consistent format the generator will produce, debugging problems in the current code produced by the generators, or writing batch or make files that will build the code into an executable.

If all goes according to plan, lunch will be a well-earned respite, safe in the knowledge that the code compiles and runs perfectly, and all that remains is to polish up the demo for the presentation. More often, there will be some problem that necessitates a short or nonexistent lunch break for at least some of the team (normally the leader, either internal or external), and panicked debugging to find out why the darned thing doesn’t work. It’s all part of the fun, and just goes to prove that DSM is part of the real world, not some utopian fantasy.

By the time of the meeting, at least some of the team are rapidly becoming DSM experts. The idea has clicked, and they are churning out new bits of generators or extensions to the modeling language at an astonishing rate. Nothing is more guaranteed to bring sweat to the brow of a DSM consultant giving his introductory spiel to senior management than to see a team member on fire, making changes to the modeling language and generators that had just been made to compile successfully five minutes earlier. And nothing is more gratifying than to be able to say “and you saw that when we pressed that new button, the whole user interface changed to Swedish—and that’s functionality that Fred there added from scratch while we’ve been sitting in this meeting.”

13.4 DEFINING THE DSM SOLUTION

After the proof of concept, the team building the solution can see that DSM is indeed applicable in their domain. Often this may come as something of a shock to some team members, who perhaps have turned a blind eye to the inefficiencies of previous practice and were certain that nothing major could be achieved. Management will also have seen the results and have hopefully allocated sufficient high-quality resources to make it possible to build the full DSM solution. The company is on the brink of making what is probably the largest increase in productivity in its history.

13.4.1 Danger: Pitfalls Ahead

...there is nothing more difficult to carry out, nor more doubtful of success, nor more dangerous to handle than to initiate a new order of things.

Machiavelli

All is not dancing on beds of roses, however—or at least the roses may be thornier at this point than you might imagine. The main dangers are ones of excess, of the team or management “losing their heads” in various ways. Let’s first deal with the easiest (and rarest): an excess of riches. This can be seen with anything new, where management goes overboard on the idea: the whole business must be turned around to follow this wonderful new approach. The best developers are press-ganged into a group tasked with creating a single all-embracing modeling language for every piece of development the company will ever do. This is simply a bad idea. Accept management’s enthusiasm, but try to curb their excess and focus instead on getting the best developers working on the best candidate domain. Once that is in production, there will be a lot more knowledge and experience to apply to subsequent domains. Since maneuvering managers is rather like herding cats, you may want to take a sneak peek ahead to the hints on organizational change in Section 13.6.1.

The second danger is to think that you are already home and dry: that the DSM solution from the proof of concept is the right one. Even in the best case, where the domain is well understood and you have had the assistance of someone experienced in DSM, there will be much work left to do. You may well be best considering the proof of concept as “build one to throw away.” After all, it only took you a few days to build, so there is no great loss. Even if you try building a new language on a totally different tack, discover that it does not work, and return to something more like the proof of concept, you will have gained far more than you lost. Understanding how the DSM solution meshes with the domain, and the various forces that pull it and constrain it, will be invaluable during its further development and evolution. If, however, this understanding is not forthcoming and you feel you are thrashing around, you may want to consider getting someone experienced in DSM to come in and help you get off to a good start.

For the other dangers we could give you a dry bullet list, but we hope you will forgive us if we try something a little more colorful, based on a story by one of history’s most famous public speakers.

3 4

Listen! A farmer went out to sow his seed. As he was scattering the seed, some fell along the path, and the birds came and ate it up. 5Some fell on rocky places, where it did not have much soil. It sprang up quickly, because the soil was shallow. 6But when the sun came up, the plants were scorched, and they withered because they had no root.

7

Other seed fell among thorns, which grew up and choked the plants, so that they did not bear grain. 8Still other seed fell on good soil. It came up, grew and produced a crop, multiplying thirty, sixty, or even a hundred times.

Mark 4:3–8, Holy Bible (NIV). 1973–1984 IBS. Used by permission of Zondervan.

The proof of concept has sown the seed of DSM, but in many cases that seed is just left to fall on the ground where people stand. To make seed grow, it has to be planted and tended—just leaving it and hoping it will happen on its own does not work. Our attention flits from one topic to the next, and tomorrow will bring us back to earth with all the normal demands of our current project. These demands will eat up our attention like the birds ate up the seed. There must thus be a clear commitment from managers, accepted by the DSM development team, to actually make a serious project out of DSM. There may be other more pressing concerns, so the project need not start instantly. It is, however, certain that there will never be a time when there are no more immediately pressing issues. While the issues may be more pressing in the short term, it is unlikely that any of them could bring the return offered by DSM.

A second form of not taking DSM sufficiently seriously is to start a project, but only allocate a student, summer intern, or new employee to work on it. For all their brain power and enthusiasm, these people lack the necessary experience in the domain, the local code base, and the realities of team development. Because of this “shallow soil”, and the small size of the team, they may be able to build a modeling language quickly: there will be many things that simply will not occur to them as issues they need to take into account. Thus while the language may appear quickly, in the hard light of day it will not stand up to the test. If management or developers are particularly lacking in perception, this failure may even be perceived to be a failure of DSM in general, stopping further attempts with a more sensible team from getting off the ground for years to come.

What then do thorns correspond to? They are the things that are already growing there, which do not produce much in the way of fruit. In DSM, this generally comes down to existing coding or modeling practices. If the development of the DSM solution starts to look too much like these, they will certainly choke its effectiveness. Sometimes the modeling language is built based on the concepts of the code world, either creating a new set of concepts or reusing those from UML. On other occasions, the generator becomes entangled in requirements that its output must be character for character identical with a given piece of existing code—as if all code in the organization were already that standardized. To be successful, DSM must build on existing code and knowledge, but it must also tear itself free from these constraining thorns where necessary. With suitable ingenuity you might be able to dress a thorn up to look like wheat, but you will never get it to produce a crop.

Thorns can also be things whose seeds are always floating around in the air, and which will take root quickly if someone has prepared some good soil. In many larger companies, there may well be a person or group with a pet theme such as “quality,” “process,” or “architecture.” While these are all good things in themselves, they can choke useful work if they become an end in themselves. As DSM includes elements of all these keywords, it can appear a ripe target for hijacking to further these ends. Enthusiasts for these topics can thus be welcomed on board, but not allowed to take over.

Finally, we have the case where the seed falls on good soil, and things are carried out in a reasonably sensible way. Then we see one of the minor miracles that DSM shares with farming: there can always be disasters, but in most cases, with reasonable effort, care, and attention, you can obtain remarkable results. Maybe your results will not be as glaringly brilliant as the county prize winner—we only know of a few cases where people have claimed a hundred-fold increase in productivity, and only one from our own experience—but anything that gives an improvement of over 100% has to be considered a great success.

13.4.2 First Things First: The Modeling Language

A DSM solution will consist of several elements, as we have already seen: a modeling language, code generators, and domain framework, and the processes and tools that support them. Chapters 10–12, 14, and 15 cover these in depth, including to some degree the process involved in creating them, and how they relate to other elements. Here we shall thus focus on how these elements work and how they are constructed as a whole.

The key element is the modeling language. This is the medium through which development work will take place: the code generators and framework are largely invisible to the modeler. The modeling language is also the schema in which products will be represented. A change to that schema will have larger implications for the existing models and the accumulated experience of the modelers than a change to the generators or domain framework. To put this another way: if we mess up when building the generator or domain framework, we will probably be able to correct the problem with little cost and with an instant and global effect; if we mess up in the modeling language, the cost of updating models will be significant, and that update may not be automatic.

As in the proof of concept, it is thus worth spending more time at the start on the modeling language. We can divide the modeling language into three areas: its concepts, its rules, and its visual notation.

Concepts Of the three areas, the main one to concentrate on is the concepts and their interrelations: the abstract syntax of the modeling language. The concepts define what information models in this language can store from the domain. The interrelations specify how the information is interlinked, for example, via relationships between objects or an object being described in more detail in a lower-level diagram.

Start by defining the concepts. Try to identify the main things that form part of each product or feature in the domain. Add the clearest and most central properties for each concept: what information needs to be stored each time that concept occurs. Look how occurrences of the concepts can be linked to form a whole product or feature, and classify the different kinds of linkages to create the relationships.

As the number of concepts and relationships increases, think a little about how you may want to reuse elements of a model in other models, and which clusters of concepts and relationships have high cohesion and low coupling with other clusters. This will give you hints as to whether you need more than one kind of graph, each with its own set of concepts and relationships (possibly with some overlap). Try also to think whether a model for a largish product or feature would likely become too large to be manageable. If so, consider adding a concept that simply points to a submodel, to allow model layering or a decomposition of a large model into a hierarchy of smaller models.

Finally—for the moment—consider the variability between products, features, and instances of concepts. What is it that needs to be recorded about these differences? What extra information is needed to generate code, or to help the generator decide which of several similar variant ways of doing something it should follow? Add support to the modeling language to capture this information, for example in the form of extra properties on the concepts, relationships, and models.

Rules Rules are distinct from the interrelations in that they do not increase what information can be stored, but instead they prevent certain kinds of information content from being stored, since it is considered illegal or undesirable in the domain. For many metamodelers, there is a considerable temptation to build many rules to protect the modelers. Now is not the time for that. Indeed, experience has shown that the number of rules in a modeling language tends to decrease over time. Modelers find that the tool prevents them from creating a certain structure, but they have a clear idea of what that structure would mean. The rule is thus removed from the modeling language, and handling for that situation is added to the generator, making the DSM solution simultaneously both simpler and more powerful.

Notation Some notation will be necessary to allow the DSM team and modelers to work sensibly with the developing modeling language. For now, the notation should concentrate on presenting the most relevant data content of each concept in a sensible and balanced fashion. The symbols for different concepts should be easily distinguishable, with central concepts or relationships being more visible. With a good DSM environment, it should be quick and easy to achieve this with simple variations in shape, line and fill colors, and line width. Fortunately, it is rare for metamodelers to go overboard with wonderful gradient fills and flashy bitmaps at this stage: clearly, there is no sense in polishing the symbols until the set of concepts is stable. Basic symbols will suffice for now.

There are, however, two things to avoid even at this early stage. Do not make all concepts’ symbols differ solely by color: the brain perceives such elements as representing minor variations of the same thing. Similarly, there is hardly ever a need to include the concept name as part of the symbol: DSM leverages the brain’s ability to recognize things as in the real world. Real-world tables and chairs do not come with such labels, nor would the brain recognize them from the labels if they did (try switching the labels if you don’t believe us!).

13.4.3 Example Models and DSM Use Scenarios

A language is nothing until you use it to say something. Even invented languages like Esperanto or Klingon stand or fall on whether there is anyone willing to speak them. The textual programming languages we use today give good examples of languages that have been designed well and had the rough corners knocked off them through use—and of languages where one or other of those processes has failed. As we saw in Section 10.7, DSM languages are no exception: we need to test the language by building example models with it.

If the example models seem to beworking well, with the language providing a good way to model individual applications or features in the domain, we can consider the next step. If the domain is relatively new, the number of people that will use the language is relatively small, and your DSM experience relatively thin, our suggestion would be to move on to the generators and domain framework. In more challenging cases, and with a little more experience, there is another area it would pay to look at first: DSM use scenarios.

A DSM use scenario looks further than the issue of how the modeling language can be used to build a single application or feature. First, it includes the broader picture of the usage of the whole DSM solution: the modeler, the tool, the models, their representation in model files, the generated code, and the finished products. Second, it expands the scope to look at multiple instances of each of these: how modelers, models, and generated files for different features relate to each other.

Looking at the broader picture of the whole DSM solution lets us spot possible sticking points for the introduction of DSM to the organization. Comparing the various phases, entities, and artifacts to current code-based development also reveals what organizational change will be necessary to adopt DSM. Envisaging the development process of a single product with DSM helps us identify gaps in our thinking: things that need doing, but for which we have not yet assigned resources. It may also suggest ideas for further automation of development: phases that are currently carried out manually by rote, but which the greater precision of DSM would allow us to automate. Sometimes such improvements may be possible with the information captured by the existing modeling language, but more often we will be able to accommodate them with minor additional properties in graphs.

Perhaps more important is looking at the issues that appear when we have multiple models. At its heart, this is a question of reuse. By its very nature, DSM has already provided a massive improvement in reuse and, in fact, has automated this so modelers do not even need to remember to reuse something. Putting an object in a diagram is reuse: all the work of the metamodeler in the modeling language, generator, and domain framework is being reused. Once we have multiple models, however, we open up the possibility for reuse of models or model elements.

Reuse of models or model elements is similar to reuse in normal codebased development. Developers need to be aware of what exists that can be reused, know how to reuse it, and how to make their own components available for reuse. Good modeling tools can, however, bring many improvements compared to code-based reuse. Perhaps the main improvement is that now components can be reused by direct reference, rather than by indirect reference through typing the same name. This goes beyond what even the best refactoring environments offer: nobody need ever think about the name change and what to do, everything just continues to work.

Reuse is discussed in more detail in Chapter 15: for now, the important thing is to make sure that if there are parts of a set of models that will be reused when building another product, these parts can be clearly separated. Reusing “every other object” from a diagram is about as possible as reusing “every other line” from a piece of code. This is thus another place to apply the ideas of commonality and variability that you have already used in defining your modeling language. This will allow your modelers to reuse a model by linking to it, rather than by duplicating it with copy–paste and changing parts of it.

13.4.4 A Balancing Act: Generators and Domain Framework

As we discussed in Chapter 11, the temptation is often to start work on the generators too early. Generators can safely be left until the modeling language and its metamodel are stable. There can still be minor changes, for example, adding a property or maybe a less important object type, but there should not be any notable refactoring of the names of concepts or the division of responsibility between the concepts. Similarly, wait until your example models prove you are happy with the central concepts and how they work in practice. If possible, wait until the use scenarios show that your language, its models, and use process will scale to fit your envisaged needs.

Starting on the generators too early will at best mean significant wasted work for you. At worst, the desire not to redo work on the generators will prevent you from making changes that need to be made to the modeling language. This would mean significant lost productivity over all modelers, over all products, and over the whole life span of the modeling language. Of course, if you find yourself in this situation, you must simply bite the bullet, make the changes to the modeling language, and update the generators—but holding off on the generators until the metamodel is stable is clearly the better strategy.

### PILOT PROJECT

While the definition of the generators and the domain framework is divided into Chapters 11 and 12, respectively, in practice they are often developed in parallel or even in reverse order. The most common pattern, and the one we would recommend for first timers, is that after successfully writing a part of the generator, you notice that it produces similar blocks of code many times in the output. You abstract that block into its own function in the domain framework and replace the block in the generator with a call to that function. In other words, as you work on the generator, you effectively refactor the generated code, extracting a repeated block into the domain framework. Note that this is different from the refactoring you will do on the generator itself, where repeated blocks of generator commands will be refactored into subgenerators.

Since there are no hard and fast rules for ordering the creation of the generators and domain framework, you should treat them as a balancing act. If you find things tilting toward a heavy generator or heavy generated code, push back by transferring some of the burden to the domain framework. If you find the domain framework becoming unwieldy or in danger of turning into an end in itself, apply pressure in the other direction by simply focusing on getting working code out. If you find the generated code becoming obscure and unrecognizable to developers, and at least at this stage there may be a need for modelers to look at and debug the generated code, take a step backward toward generating the same kind of code as good developers previously wrote by hand.

Remember that you can always refactor the balance later: existing models will continue to generate code that works fine, even if you move blocks between the generator and the framework.

As we saw in Section 11.3, generators can be used for things other than code or other product source files such as XML. In most cases, generators for things such as model checking or documentation are not essential at this stage, and can be left to bubble away on the back burner during the pilot project. An autobuild generator, as discussed in 11.4.1, may however be an important element even in the pilot project.

13.5 PILOT PROJECT

While you will have drawn sketches of models while creating the modeling language, and built working example models before creating the generators, there is nothing like a real life project for testing out a DSM solution. In the pilot project, a separate team will be testing the prototype DSM solution to ensure that it works and is a good medium for building applications in the domain. The DSM solution team will be actively present throughout the project, helping the pilot team learn and use the DSM solution, and also fixing and improving the various parts of the solution as necessary.

You need to find a pilot team with a real product or piece of the product to build. As we mentioned above, the target should be something fairly standard in terms of the domain, and the project should not be on the critical path. The team itself should be motivated and disciplined.

The modeling language will have remained relatively stable while building the generators, but now it is time to open it up to change again. Of course, it will break your heart to change what in all likelihood will have become your “baby” by now. In addition, of course, there is a higher cost associated with changes at this point, since you will need to update not only the modeling language but also the generators and existing models. However, this heartache and cost are nothing compared to the heartache you will cause modelers if you do not make the changes, and the cost to the business in terms of lost productivity.

All too often at this stage we have seen people make mistakes that proved fatal to the whole introduction of DSM. Most often these mistakes are some kind of shortcut or ducking out in terms of time, resources, or responsibility. In a way, this is perfectly natural: defining the DSM solution was a challenging learning process, but in a fairly safe environment. The only people involved on a day-to-day basis were the team themselves, and the activity was largely technical. Now a new group of normal users are going to see and use what the team has built. This will certainly result in criticism, constructive or otherwise.

This is also the first phase where there is the start of the organizational changes that DSM entails, and many technical people are uninterested in doing the people-related work to make changes like that run smoothly. If management are skeptical or uncommitted, they may well have temporarily pulled key resources off the team during the earlier phases to fight fires elsewhere. The project will thus have taken longer, increasing the pressure to try to cut corners. This is not the stage to take your best people off the DSM team. You will need all your skills and resources to climb this next learning curve and become adept at managing the evolution of a DSM solution while it is in active use. This, after all, is the task of the DSM team in the future, and it is a lesson best learned now while things are still on a small scale.

At the start of the pilot project, you will need to provide the pilot team with training and documentation of the DSM solution, as well as the tool itself. One good training method is to walk through an existing example model, explaining what it means, and then generate and show the running application. You can then start with an empty model and rebuild the same example from scratch, so the team learn the tool and also get a feel for how long things take. If the team is used to coding in the domain, you can also show them the generated code, but remind them that the idea is to move above that level to think in terms of the domain rather than the code.

As the pilot team use the DSM solution, you will find bugs that need fixing straightaway, areas that need improvement, and things that could be better. It is by far the best if these changes can be made quickly, with minimal interruption to the work of the pilot team. As with later DSM use, this is an area where you will see real benefits to having a DSM environment that makes such evolution possible and easy. Useful features include being able to metamodel and model in the same environment, as opposed to different environments, and automatic updates of existing models as the metamodel changes.

While versioning in general has a smaller role in DSM than in code-based development (see Section 15.3), it is at its most important during the pilot project. At no other stage will the parts of the DSM solution be evolving so rapidly while there are models that rely on them. You must know at all stages which version of the metamodel the models correspond to, and which versions of the metamodel, generators, and domain framework go together. In each of these elements, there are two kinds of changes: changes that require a change in other elements, and changes that do not (e.g., simply changing symbols or fixing a bug in a domain framework function). Use major and minor version numbers to separate the two kinds of change, and have elements’ version comments refer to the versions of the other elements they depend on. In particular, the generator will depend on both the metamodel and the domain framework. As always, version comments can also provide a vital record of the design rationale behind decisions.

At the end of the pilot project, you will have your first working application built with DSM. You will also have a fully working and proven DSM solution, and a fair idea of what kind of use process works well with it. The pilot project also has a major role to play in the process of organizational change. It acts as both a marketing vehicle, showing management and other developers that your DSM solution works, and as an educational program, producing the first small crop of users who are familiar with building applications using DSM.

It is thus important to wrap up the pilot project properly. If there were any changes to the modeling language that you had to postpone, make them now. You should also apply these changes to the pilot project models, both for testing and also since you want them to be at the same version level as the models, both built during the next phase of wider deployment. Write up and present the experiences of the pilot team as a project retrospective, and also record the experiences of the DSM team: what kinds of evolution worked well, and what kinds gave problems for either the tools or their users.

While the special nature of the pilot project makes it less productive than later use, you can still gather some statistics on the time taken to create the product compared to code-based development. Although these figures will not be accurate or statistically significant, management is unlikely to fund the 25 pilot projects that would be necessary for statistical significance! A clear increase in productivity in the pilot project, accompanied by an analysis of problems faced and suggested solutions, should be enough for most pragmatic managers to give the go-ahead for wider deployment of DSM.

13.6 DSM DEPLOYMENT

After the pilot project, the structure of the modeling language and the behavior of the generators will be in good shape. Indeed, in these aspects the DSM solution should be able to make a good claim at production quality. There are, however, other aspects to the solution that are not yet at the same level of quality: it has not made sense to work on these until the core of the solution is stable. In this section, we will look at these areas and at what extra collateral is necessary to turn the DSM solution into what Geoffrey Moore (1999) would call a “whole product.” First though we will take a brief detour to look at the issues of organizational change involved in DSM adoption.

13.6.1 Learning about Organizational Change

For all our love of trying out new technology, we developers are not exactly renowned for our ability to cope with change, particular if it feels it is by imposition rather than choice. Fortunately, there exists good work elsewhere on the topic of introducing change, and that can be applied equally successfully to DSM introduction.

When thinking about deploying your DSM solution, take the time to sit down with two people. First, you need someone from your IT department, who has been responsible for rolling out new technology. If you look how much time and energy goes into something as simple in theory as updating Windows, you should get an understanding that deploying DSM is more than just putting a tool license on everyone’s desk and the metamodel on a file server. Of course, Windows is an unfair example: most of the work there is checking all possible applications for compatibility with the new version. This is not a problem in DSM: the results of DSM—the source code—will look the same as previous source code and run on exactly the same platform. Instead, the change is more like moving from writing documents in LaTeX to writing them in a word processor, or from writing HTML in a text editor to creating it with FrontPage. You are moving from a textual language to a new tool with its own graphical user interface “language.” The IT department will understand the issues of tool deployment and training related to the new tool, language, and process, and you can learn a lot from them.

Second, talk to an old-timer from your development staff: someone who has lived through a major change in development tools and languages, for example, the move from assembler to third generation languages (if you are lucky!), or the move to object-oriented languages, or at least the introduction of UML tools. Only the first of these really corresponds well to introducing DSM, but you take what you can get. In addition to the external view of change introduction given by the IT department, this will give you the internal view of how the change actually took place in the minds— and hearts—of the developers.

Rather than turning aside from the topic of this book to the general psychology and techniques of change introduction, we will simply point you to the excellent set of patterns by Mary Lynn Manns and Linda Rising. They are now available as a book, Fearless Change: Patterns for Introducing New Ideas, and earlier versions are available online, for example, http://www.cs.unca.edu/manns/PC.DOC.

13.6.2 Polishing the DSM Solution

The concepts and main rules of your modeling language are ready, and the generators produce good working code that runs on top of the nascent domain framework. Practically speaking everything works, and if you took it into production now, you would see the large productivity increases you expect from DSM. There does, however, remain some significant work to do. This work will not noticeably improve the productivity of the solution in use, but it will make a big difference to how well your DSM solution is accepted. Without that acceptance, getting the DSM solution into use may take much longer or even fail to happen.

Fancy symbols are, of course, completely unimportant to productivity. Provided the symbols let you easily distinguish the concepts, and present the main properties of their concepts in an easily readable format, you already have a good visual language. We developers, however, are finicky creatures, and the human race as a whole tends to judge by first impressions. A sloppy, ugly set of symbols will have a large impact on how your language is perceived. Even a perfectly decent set of symbols may look rather simplistic to eyes used to Vista or OS X. With access to the right tools and skills, you can turn a good language into a great one. For many of us, this is probably an area best left to (semi)professionals—providing that you can communicate to them the real needs of the situation and that in this case function must always come before form.

The symbols represent the static part of the language, how the users will see the models. On top of this is the issue of how the users will create, edit, and navigate around the models. This user interface will be highly reliant on the tool support facilities provided by the DSM framework or environment you use. We will look at this in more detail in the next chapter, but for now the main issue is that if there are things you can improve in the UI, now is the time to do that.

Next you should look to the other visible parts of the DSM solution. Check that the generated code and domain framework follow in-house standards for coding style, formatting, and so on. Although developers should not have to look at these, they will certainly be turning a sharp eye on them at this stage as they form their opinions about the DSM solution. The other side of the coin is process automation: making an autobuild script or generator so that developers can go straight from models to running the finished product, without having to look at the code or run commands on it.

If the preceding topics were about making things aesthetically pleasing and making visible “seams” in the process invisible, we now want to consider making things visible that are currently hard to see or invisible. For instance, there will be model structures that are not acceptable to the generator, but which the modeling language allows you to create. During earlier stages, we have advised you against adding too many rules to the modeling language. Now is the time to make sure that the necessary rules are in place. You are not trying to stop the developer from doing things, more to prevent her from making models that the code generator does not support. There will still be a large number of models that can be made which make little sense. Trying to prevent all of these is impossible and also undesirable. Some of them will turn out to be perfectly good ideas: creative ways of using the language that had not occurred to you yet.

You should also make sure that your rules do not make the language hard to use. There may be certain states of a model that are illegal, but would occur frequently on the shortest path for editing a model from one legal state to another. It is better to implement such rules as checking reports, which can be run by the user on demand or automatically when generating code.

If your organization has a strong requirement or tradition of documentation in Word files or similar, you will probably need a way to generate the models, their elements, and properties as a document. Good tools will offer you this already, but even then you may want to edit the styles and layout to match existing standards. If the DSM environment provider has not done this work for you, and you need Word files rather than plain text or HTML, be aware that this can be a significant amount of work.

Finally, now is a good time to complete any refactoring on the generator definitions. This will obviously help you later in maintaining the generators, but will also make it easier to evolve the modeling language.

It will hopefully go without saying that you should also have whatever documentation is appropriate for the internals of the DSM solution itself. The project may be your baby now, but what about when you get that big promotion on the basis of your results here? Spare a thought for the poor soul who is given the task of taking over the DSM solution—and for yourself, if the project crashes and burns behind you. Do you really think you can spin it so it looks like you were 100% a hero and the new guy 100% a villain?

13.6.3 Introduction and Evolution Process

One of the more frustrating things for developers is the sheer amount of work that still remains to do once the product is “ready.” Polishing the product is one part, but even after this there is still the installer, download or other distribution system, user documentation, and training material. Introducing the DSM solution to your organization follows this pattern, with a few minor differences.

Because of the importance of evolution in DSM, we want to plan the distribution mechanism for the DSM solution before we finalize the user documentation. At the same time, we can also cover the initial distribution and introduction process. These need to be planned and tested well: while it will be a pain to have to install and uninstall the various components on a few machines, this pain will be nothing compared to the alternative. Imagine what would happen if you tried to push DSM out to all developers with a half-baked distribution system with poor instructions.

We have divided the DSM solution into several components in this book: the modeling language, generators, framework, and tool support. These components may not, however, match the actual files or other elements you need to install for users. For instance, in some DSM environments the language and generators may normally be delivered as a single file, whereas in others they may be delivered in separate files—or even a file for each diagram type and each generator.

In less mature environments, there may not be a separation between the generic modeling tool support, which is the same for all organizations using the tool, and the domain-specific language and generators of each company. Instead, the language and tool support, and possibly the generators, will be lumped together in one large executable or bytecode package. This is not such a problem for the initial introduction, but it makes evolution a harder process. In particular, looking at older versions of models in this situation is problematic: you will need to install the complete, correct old version of the tool. If the tool does not allow multiple versions to be installed simultaneously, this is something of a disaster. The situation becomes even worse if the tool is tightly coupled with your IDE or operating system version: reinstalling the required compatible version of those is unlikely to be anyone’s favorite pastime.

Whatever the situation for your particular case, you must identify the components you need to distribute to developers initially and when upgrading. For upgrades, consider which parts of the DSM solution may change, and whether you want to allow an upgrade just for that part or for a set of parts together. Use your experience from upgrades during the pilot project to come up with a scheme that makes sense for your situation. Remember that at the start the rate of evolution will still be fairly high, although not at the extremes seen in the pilot project: the DSM solution is now more mature, and the larger number of developers adds some inertia.

As you build a mechanism and process for distributing updates to developers, do not forget the importance of a communication channel in the other direction. Developers need a way to submit feedback about the DSM solution. There will probably still be minor bugs in the domain framework, generated code, and generators, since these are the most code-like parts of the solution. If the tool support includes handwritten code, or is produced by a DSM framework that is still under development, it may well contain outright bugs—in addition to the certainty in all tools that individual users may wish some things had been done differently. Finally, the modeling language itself will have areas for improvement and extension, both at the start and later as the domain evolves. Developers need to be able to report all of these, and ideally be informed when the problem is corrected.

At this early stage before the initial deployment of the DSM solution, it is of course impossible to create the perfect process for evolution. What is important now is that there is a tested and documented process that works technically: all developers can easily upgrade to a new version of the DSM solution, and all models update sensibly to work with the new version. This process itself will evolve over time as you gather more experience of what works well with your tools, language, and organization.

13.6.4 Learning Material and Training

The developers who will use the DSM solution are faced with a significant learning curve. The task is made easier since the domain will be familiar, and the modeling language concepts will normally map directly to the domain. On the other hand, the developers will probably not be accustomed to using a graphical modeling language to actually build products, as opposed to simply designing or documenting them. Part of the function of the learning material and training will thus be to help ease this shift in mindset.

Clearly, the most important element to teach developers about is the DSM language. Good DSM environments will allow you to document the individual language concepts inside the tool itself. This will serve as a good online reference to the modeling language, as well as providing context-sensitive help about each of the concepts. Additionally, it is often useful to have the language described in a more familiar format, either on paper or in some electronic document format.

Since the developers will not be metamodelers, it is probably not so useful to provide an actual metamodel of the modeling language. Instead, try to build an example model or models that illustrate all the concepts and the way they are used. You can then describe the model and the way it works in the text. This tends to be more useful than a list of the concepts and their descriptions. Such a list can be added at the end as a reference, although keeping this documentation in the tool itself puts it closer to the developers and also reduces the risk that it will become out of date.

For the tool itself, first look what the tool provider offers in terms of user documentation. All but the most primitive DSM environments and frameworks should have abstracted tool behavior from issues relating to individual modeling languages, so the tool’s functions and behavior can be described once for all modeling languages. In addition, it is also useful if you can provide some initial documentation that describes things in terms of your modeling language. This can help make things more concrete and reduce the danger of new users feeling unsure about which metalevel they are on. Something that has worked well in our experience is to provide step-by-step instructions to building a simple model, just a little more than “Hello world.” Outlining a second task, without the step-by-step instructions, is a good way to nudge users onward from rote repetition to actually learning and understanding how to do things themselves.

Simply providing people with a tool, a language, and documentation goes a long way to helping most of them become productive in the new environment. Different people, however, learn in different ways, and for some the best way is a more formal training session. A disproportionately large number of software developers learn best by reading, but DSM also opens the doors to people who have not traditionally been perceived as developers. A good example of this was seen in the insurance domain experts from Chapter 6, many of whom had never programmed before. Rather than studying alone by reading, these people may prefer to learn by hearing, by seeing, or by doing, often as part of a group. Group training also has benefits for all kinds of learners: increased out-of-band communication, a sense of group identity, and learning to learn from one another and help one another.

Initially, any training will probably be carried out by the DSM solution team. It can also be a valuable process for them, as they see first hand any problems that new users have in getting to grips with the language and its tool support. Since there is no reason to expect the DSM solution team to be particularly gifted or comfortable as teachers, it may later be kindest to all to let them pass the baton to people with more experience in this role. While the odd fine piece of detail may be lost in this transition, overall it is the clear communication of the basics that is more important. More esoteric information and tricks of the trade can be passed on later in various creative ways, for example, by a “Tip of the Week” email.

13.7 DSM AS A CONTINUOUS PROCESS IN THE REAL WORLD

By now it should be abundantly clear that DSM is no silver bullet—nothing is—and certainly not a one-shot, fire-and-forget weapon. A large part of its power comes from its ability to evolve, and this in turn stems from its being close to the users

DSM AS A CONTINUOUS PROCESS IN THE REAL WORLD

and needing to support only a bounded domain and set of users. As the domain evolves and the group of users grows, you will face again the questions of where to draw the boundary, what to include, and what to leave out. You will also face the question of evolution in your organization: fitting the needs of the organization with the desires of the people who make up that organization. Finally, there is the question of the fate of an individual DSM project: what happens at the far end of the life cycle.

13.7.1 Evolutionary Forces

Evolution in human organizations works best when it can find the right balance between change and continuity. At one extreme, the DSM solution could change so fast that nobody could keep up, or split into a myriad of languages, each for too narrow a group. At the other extreme, the DSM solution could stagnate, either not being maintained or because users do not update to newer versions. Most of the dangers here are obvious, but we will pick out a few to look at.

Here, we have mostly assumed a single DSM solution, a single group of users, and a single version of the DSM solution in use at a time. In reality, there may be pressure to have different variants of the DSM solution for different groups of users. This pressure mostly can and should be resisted if the variations are small. If a group has a clearly different domain, this may require either a new modeling language in the same DSM solution, or a new DSM solution entirely.

As individual projects move through their own life cycles, there will also be phases when they are unwilling to adopt new versions of the DSM solution, for fear of introducing unexpected changes or bugs. To some extent this is justified, although the risks must be re-evaluated compared to code-based development. Clearly, nobody would want significant changes to many pieces of code shortly before a release, and in a way a change to the generator produces that. However, the real danger is not in the number of changed lines of code, but in the number of places where a person might make a mistake. In the code-based world, this would be directly proportional to the number of changed lines. With DSM, the level of risk is measured largely by the number of lines changed in the generator. If changing that number of lines would be acceptable in manual coding at the current stage of the project, the generator change should also be acceptable.

Another possible comparison would be with upgrading to a new version of a compiler at a late stage of a project. This provides a better match with DSM upgrades, but there are still some important differences. The compiler is made by an outside agency, and any bugs found in this project will normally take far too long to be corrected. Since the DSM solution team is in-house, any bugs found in the new version can be corrected quickly. Also, the compiler is made by people unfamiliar with the domain and the code of this organization. The DSM solution is, of course, made by people at the opposite end of the scale, experts in both the domain and the code. The chance that the compiler team will introduce an error that affects this organization, because they were not aware of or did not consider its needs, is much higher than the corresponding risk for the DSM team.

13.7.2 The DSM Solution Team

An important source of continuity in DSM evolution is the DSM solution team. As far as possible, try to keep the team together. Since the team must be staffed by your top experienced developers, there are few enough people qualified to be part of it. The people who have served on it for the first round will probably also be your only people with experience of creating a DSM solution. Of course, if it has become clear that this was not the right place for a certain person, common sense must be applied. A good place to look for a possible replacement is the pilot project team. Not only have they had more experience with DSM and your solution than others, but also they have been more exposed to the internals of the solution and the process involved in creating it.

The size of the DSM solution team will largely be dependent on the DSM environment or framework you use. A high-level DSM environment will keep the amount of work low enough that the evolution of the DSM solution will require less than one person’s full-time work, even for a large language. Lower-level frameworks will require a few full-time staff, possibly even several people. In addition to these, you may need one or more people working on the domain framework: whether these are considered part of the DSM solution team or your general in-house components team is perhaps a matter of taste.

If your organization has more than one domain for which DSM seems applicable, there is also the question of the next DSM project. Certainly the first DSM solution team will have plenty of experience that can be used beneficially on the next project. How best to use it is however dependent on the particular situation: how much the domains are related, what the people in the first DSM team want, how mature the first DSM solution is, and what kinds of people are available in the new domain.

In some ways, an external DSM consultant may still be the best solution for the second project. The leader of the first project has one success under her belt, but this may not yet mean she is best suited to teach others. If there are other applicable domains after the second one, and someone from the first team is keen to become the in-house expert, perhaps having her work alongside an external DSM expert on the second project is the best solution. This should help avoid the danger of falsely abstracting as a general principle something that was a particular feature of the first solution, while at the same time setting you up to become more self-sufficient in the future.

13.7.3 The Rise and Fall of a Domain

In our experience, as long as a given domain remains useful to an organization, and the DSM solution for it is allowed to evolve, there is no built-in aging process in DSM. On a technical level, it appears that DSM languages actually have a longer life span than generic modeling languages. Among our own customers, there are hundreds of people actively using DSM languages that predate UML, at a time when UML is clearly heading toward its sunset.

Perhaps the key difference is that a generic modeling language in use is only really able to grow, and eventually it either stagnates or collapses under its own weight. The

DSM AS A CONTINUOUS PROCESS IN THE REAL WORLD

latter fate is perhaps more common for languages with an evangelistic bent, as they try to increase their range of applicability to an ever wider and more diffuse group of users. A DSM language can grow, but it can also evolve in other ways. If a change needs to be made, it can be: there may be some pain associated with it, but the gain to the organization can be seen to be greater. With a generic language, such a change will often be left undone through fear—and well-grounded fear at that. If the language changes, will the third-party tool vendors update their tools to support the new language, and provide a way to update models to match the new language? The tool vendors must face a similar fear: if we update the tool, will users choose to stay with the old version?

While a DSM solution will remainviable in its domain, even as that domain evolves, there is also the question of the viability of the domain itself. Businesses change, and it may be that at some point customers are no longer asking for products in that domain. Companies must also use their limited resources to best effect, and sometimes this may mean even a viable domain is shut down, and the developers are moved to a different project that offers a better return on investment or promise of growth. Unfortunately, we must also take into account the possibility that a new manager or owner may feel the need to assert themselves by making radical changes—even where those changes are not merited by the situation. Sometimes this means a wholesale change to different tools, languages, or processes, and that may mark the end of a good DSM project. Given the all too common alternative way of asserting authority—firing large numbers of people—this is perhaps the lesser of two evils.

In all of these cases, our suggestion would be to make a good end of it. If the change is inevitable, there is nothing sacred about a particular DSM solution. Fight bad ideas by all means, and make sure those in charge understand the productivity benefits of DSM, but look at the situation with a healthy dose of realism. If the use of this DSM solution is going to stop, it is time to wrap up the loose ends. Although there will obviously be limited resources for this task, make the most of what you have.

Make sure that the final versions of the DSM solution components are recorded and that the components themselves are stored. The models themselves are assumed to be under good version control and so should present no extra problems. If your tool support is based on a framework that is tightly integrated with an IDE, or for other reasons may later be challenging to reconstruct, it may be a good idea to make a disk image or a virtual machine containing a working environment. Should it later be necessary to resurrect the DSM environment, for example, to allow some final bug fixes to a released product, this will allow those changes to be made much faster and with more certainty that no unintended differences are introduced.

Although most of the history of the DSM solution development will be under version control, there may well be extra information that is only stored in the email folders or on the hard disks of the DSM solution team. A quick trawl through the relevant locations should turn up some useful gems to record for posterity.

In many ways, closing down a DSM project is more like putting it in mothballs than killing it off. With a traditional hand-coded project, changes in libraries, platforms, and even programming languages will quickly render the old code useless. With DSM models, the modeling language and models would remain valid even in the face of such major changes: new generator and domain framework are all that would be needed. The same forces that resulted in the domain being retired could just as easily result in its resurrection. While a full-scale revival may be unlikely, the ability to quickly churn out a couple more products in the old range next year may be a useful option from a business point of view. In particular if a new manager’s demand to revert to a “standard” programming and generic modeling approach is inexplicably taking longer than expected to produce results!

13.8 SUMMARY

This chapter has seen your DSM solution evolve from an initial idea to a full solution in production, reflected throughout in experience of its use. The evolution of the DSM solution has been a major factor, but the evolution of your organization has been even more important. The developers have learned to use the DSM solution to good effect, shifting their thinking to a higher level of abstraction, closer to the requirements and to the end users. The DSM solution team have had their first experience of creating, introducing, and maintaining their own domain-specific modeling language and generators, taking control possibly for the first time of their own development process, tools, and languages. No longer will the offerings of third-party vendors and outside organizations seem like the only possible “real” choices, with in-house solutions seen as little hacks. There will also probably be a change in attitude toward such third-party tools and solutions: any unrealistic feelings of their unassailable authority and superiority will have evaporated; any unrealistic demands for perfect solutions will have been replaced with some measure of respect for the trade offs those vendors are forced to make.

There will also be an increased maturity in the overall issue of introducing new technology to your organization. In someways, the first DSM project will be the major change; in other ways, each subsequent DSM project will be its own change process. While such processes are not at the heart of what most top developers love about their work, most would acknowledge that such “soft,” people-related issues are a major factor affecting the success of any project, even when the “hard” technology side is in great shape.

Most importantly, running through the process suggested above even once will have given you some insight into how to adapt its necessarily generic advice to the specific situation of your organization. Even on your first run through you will have tailored some parts and made your own adjustments. If the first project was successful, in subsequent domains you should find you have less need to market and prove the approach to management before getting the green light.

Should you go on to apply DSM in several domains within your organization, and it becomes a standard tool in your palette of approaches, tools, and languages, you will have entered a select band of a very few organizations in the world. If after a few DSM projects you have found changes or additions to the above process that seem to work in general, please do share your feedback with the authors or the wider DSM community.