### CHAPTER 15

## DSM IN USE

Section 13.4.3 already looked briefly at some of the issues of the use of the DomainSpecific Modeling (DSM) solution from the point of view of the modelers. In this chapter, we will discuss in more detail those aspects of modeling that are not already covered by the decisions made in the DSM solution, and also look at some such decisions that can only really be made when wider issues of use are considered.

In many of these areas, DSM works rather differently from code-based development. Although this is to be expected, it often comes as a surprise to find that some accepted wisdom and truths held as self-evident were actually only valid in the context of code-based development. Some of the “version 1.0” DSM environments have not yet discovered these differences and try to store the new wine of DSM in the old wineskins of textual programming languages. In this chapter, we will assume an object-oriented approach to models and model storage and seek the best solutions for DSM: once we know where we are heading, we can better decide whether we want to make transitionary concessions to existing practices.

15.1 MODEL REUSE

In many ways, the productivity of DSM can be attributed to various kinds of reuse. Existing expert experience is used in building the modeling language, generators and domain framework, and all of these are reused automatically each time a developer makes a model.

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

397

DSM is however particularly well suited to further reuse: existing models and model elements can be applied and referred to when making new models. Such reuse should generally aim to be by reference, not repetition.

15.1.1 Copy by Value Versus Copy by Reference

Traditionally, there has been a distinction between two methods of copying: copy by value and copy by reference. Although this will be familiar ground to most readers, let us lay down some basics first. We will say that there is an existing element, already used at one place in the system. The element has some content, and we want to use it in a new place too. We will keep the details deliberately vague: element, content, place, and system could map to many things, depending on our development process and artifacts.

In copy by value, the content of the existing element is copied to the new place. The content thus exists as two separate copies, and changes to one will not affect the other. In copy by reference, the content of the existing element is simply referred to from a new place. Any later changes to the element are thus visible in both places where it is used.

The advantages and viable use cases for these different methods should be clear, at least for this limited, abstract view. Taking a wider view and demanding more details reveal some extra complexities.

15.1.2 Names and Copy by Reference

First, how do we actually make a reference to the existing element? The traditional method from coding has been to give the element a name. More precisely, the name is a sequence of characters that serves to uniquely identify a particular element in the contexts in which the name is found. This introduces a distinction between the place where the element is defined and where it is used: the definition includes the name and the content, the uses include just the name, or to be precise, the same sequence of characters.

The content of the element thus remains pure and simple: it exists in one place, so everything just works. The name of the element however exists in two or more places in the system. If the name is changed in one place, the same change must be applied to all other places, if the references are to remain intact. Ideally, this support would be provided automatically. This can prove to be surprisingly difficult: there may be no easy way to find all such places; in more complex cases, it may even be practically impossible.

In copy by reference, we can always change the content of the element, affecting all places where it is used. We can also always change one or more uses to point to a different element, simply by using the new element’s name instead. The loose coupling between the various occurrences of the name also gives rise to a new possibility. We can deliberately change the name in the definition of the element and create a new element whose definition includes the old name. All users of the old element are now automatically users of the new element instead. This kind of

### MODEL REUSE

indirection can prove useful and powerful, but it introduces a problem: when changing the name in the element definition, a developer must remember to specify whether she wants to change the name in the references too.

15.1.3 Copy by Reference in Models

In a modeling tool, we can use copy by reference via the names of modeling elements, i.e., particular string properties. The tool itself need offer no particular support: it is enough if the generated code is formed so that the references lead to the right elements in the code.

In the dirtiest approach, it is even possible for copy by reference in a model to map to copy by value in the code: the content is included many times in the code. This is similar to the inlining performed by optimizing compilers: program size is increasing, but speed may also increase. Although any duplication like this should ring warning bells in developer’s minds, the dangers here are significantly less than if such code were handwritten. As the code will not be edited by hand, we avoid the problems of copy by value such as loss of synchronization.

Using copy by reference in this way in a modeling tool may however be simply perpetuating a poor solution. In textual program languages, a name was the best available way to refer to an existing element. The atomic elements that programmers and compilers worked with were characters, and the next largest element was the file. In a modeling tool, we have at least characters, properties, objects, and graphs. Each of these has its own unique identity, regardless of its contents or any part of that content we consider its name. When a modeler reuses an element in a model, he most likely simply selects the existing element directly. The reference can thus easily and permanently be made to be to that particular element.

Modeling tools thus allow an even purer form of copy by reference: we can call this copy by direct reference. Since modeling tools also support the older form, we can call that copy by name reference. As the direct references remove many of the problems of name references, most reuse will happen that way.

The only benefit of name references over direct references is the extra level of indirection provided by the name. We will later see a situation in which we can take advantage of that, but for most uses equivalent results can be achieved by direct reference, without introducing the problems of name references. We can simply add an extra model element between the referrers and the reused element. The references are directly to this new element, which in turn refers directly to the reused element. At any stage, the new element can be altered to refer to a different element, thus affecting all of its references.

15.1.4 Copy by Value in Models

When programmers copy an element of code by value, that is, by copying and pasting it, it is immediately apparent which atomic elements have been copied: the particular characters of the selection. These characters are however later interpreted by the compiler, adding semantics above the level of characters. Some of the copied text will map to individual primitive commands, which will be copied by value. Other parts will map to references by name to other functions, and while the references will be copied, the content of the functions will not. Copy by value thus always includes the idea that the copy is to a certain depth, at least if the copied content also makes use of copy by reference.

When copying by value in a model, there is a greater range of choices. Model elements link to other elements in a wide variety of ways: relationships, shared properties, objects as properties, subgraphs, and so on. The exact set of link types and their semantics will be determined by the tool’s meta-metamodel. The metamodel may specify additional information about some of these links, for example, an object might be held in a property as a simple reference or as a strict aggregation. This information can be used to make choices about which subelements should also be copied and in which cases the copy should share the same subelement as the original.

However, even with this information from the meta-metamodel and metamodel, there will often be choices that could be left to the modeler. The choice to use copy by value rather than copy by reference tends to be a concession to pragmatic needs, and so too with the choice to copy deeper than normal. For these cases it is useful if a tool offers the ability to choose what kinds of links to follow and how deeply.

One possible cause for deep copying is ad hoc, opportunistic time saving: an element to be created may be similar to an existing element. Although many kinds of similarities like this can be factored out into their own variation points, allowing a simple reference or higher level object to make the difference, there will always be cases that the modeling language cannot handle perfectly. Even if the modeling language were perfect, there would still be times when two model fragments would be similar in a number of respects and different in others, but purely by chance rather than any underlying relationship between them.

Another possible cause is a factor outside of the scope of the DSM solution, for example, a business agreement or a divergent branch of development. A model for one client may be similar to that required for a subsequent client, yet the agreement specifies that the models must also be editable by the client. If the development for the previous client is not frozen, the second client obviously cannot be allowed to edit the same set of models. The models must therefore be copied deeply, at least for those that will be passed to the second client. Even if the client does not request the right to edit the models, a project-based organization will often prefer to keep the two projects disjoint. Although this is not ideal, the situation is still significantly better than with code-based development: the modeling language and generators ensure that the majority of commonalities are shared by all from one central source.

15.2 MODEL SHARING AND SPLITTING

So far, we have largely considered one set of models in one instance of a tool. This seems to be the theoretical ideal for DSM, taking the greatest advantage of the possibilities for model reuse. These advantages can and should be maintained when

### MODEL SHARING AND SPLITTING

scaling to multiple users. This has been demonstrably achieved through multiuser repositories, but not yet for disjoint files.

15.2.1 Disjoint Files

Splitting a set of models into individual files, one per model, may seem like the most natural approach. In the unlikely event that the models are completely disjoint, in terms of both references and semantics, this may be a viable choice. Even so, there are still several decisions to be made:

. Does a file hold one graph or several (e.g., its subgraphs)?

. Does the file also hold the representational information (e.g., the diagram)? . Can a file hold graphs of more than one modeling language?

In the more likely case that the models contain references, there needs to be some way of expressing those references. For such a reference to work, it needs to express three things:

1. A unique reference to the file in which the other element is saved.
2. A unique reference to the element within that file.
3. A unique reference to a particular version of that file (optional).

Since the references are made to an element within a file, rather than to the element in memory, we must also ensure that the version on disk is the same as that in memory. Otherwise, we may refer to an object that has been created since the last save but deleted before the next save. In practice, this means that to make a reference, or at least to save a file containing a reference, we must also save any changed files for all other models whose elements it refers to. Similarly, we must also make sure that these files are placed immediately into version control: otherwise even though an element made it onto disk, it may never have made it into the version control system. If the referring file did then make it into version control, the version there would be inconsistent.

The reference to the element must be by some automatically generated unique identifier. In a textual programming language, it is clear to all developers which names may not be changed. In a graphical model, users are used to being able to change names freely and have other elements cope with the change. If the referring elements are within this model, that would indeed be true, but for elements outside it cannot be. Using a GUID or similar approach will allow users to change names freely where that is desired. Where it is not, the metamodel can use reuse by name reference anyway, and the file will contain no explicit link.

Using a GUID may also reduce the need to store the target file’s version information in the link. If a simpler counter index is used, there is the risk that another version of the same file could contain the same index for a different object. Saving the maximum value of the counter so far in the file does not help: two branches from a file where the counter is 10 will both call their respective next elements 11, even though they have nothing in common. A reference to 11 can thus later be misinterpreted to point to the wrong element.

When sharing these files, we must also examine what makes a good unique identifier for the file itself. It is probably safest to use the unique path or internal identifier from the version control system. It may well be wise to include this information within the file itself to provide an extra measure of security.

Whatever the official process, if you give developers text files some of them are guaranteed to try to hack them in some way by copying and pasting content between them or editing them. Since the chance of them being able to do this correctly when links are involved is vanishingly small, and the consequences of getting it wrong both large and possibly not noticed until later, links in disjoint files will always present a risk.

Even if developers can be prevented from editing the files (uninstall Notepad?!) and the version control system pressed into service as a makeshift repository, there is still no guarantee that links will be in a consistent state. One reason is that version control systems do not protect the version that is the target of links. Your model A may link to an object X in model B version 1.0, on the same day as the user responsible for B has it checked out for changes. She may delete X and save B as version 1.1. This is bad enough, as X will disappear completely when you next look at your model (assuming your tool loads updates from other models—if it does not, there really is no hope of ever making a coherent build).

Unlike with textual programming languages, you will not even see the name of the object you referred to (presuming the tool can even open the file). If the tool has saved the version information of the link, you will at least know which version of B to look in. The tool might even theoretically be able to fetch a shadow of version 1.0 of B and somehow handle having both 1.0 and 1.1 in memory. Alternatively, links could always save at least some basic content information about their targets, for example, their name and type.

It does not take a particularly evil mind to imagine that your colleague might also have chosen to refer to one of your objects. Her model 1.1 may thus refer to your model 1.0, and your model 1.1 to her model 1.0: a cyclic dependency. While these are all perfectly normal problems with textual languages, trying to correct them at a textual level for models is something of a nightmare. The only real solution is that the tool be able to handle these issues, but current evidence shows this to be too much to expect: sadly, it is the “version 1.0” tools that use disjoint files. While some do not yet offer links between files, most of these say they are considering it.

For now, then, if you are stuck in a tool that offers intermodel links between disjoint files, but not the tool support for protecting you from the problems this may cause, our advice would be to avoid reuse by direct reference across model boundaries. This is a shame, as such reuse, when supported by a true repository, is something we have used to good effect in almost all cases of DSM we have worked on.

15.2.2 Multiuser Repositories

In a multiuser repository, all models are available in one coherent space. Models and their elements can be freely browsed and reused by direct reference. User permissions

### MODEL SHARING AND SPLITTING

and modeling language rules can be layered on top of this basic freedom to limit it where necessary. The repository and tool take care of the consistency of links between model elements. Elements are not deleted from the repository while others refer to them, so situations such as those in the previous section are avoided. In most cases, referring to an object that has been removed from another model can still generate good code. If reuse should only be allowed from a particular collection of objects, it is easy to add constraints or generator-based checks to ensure that this is so.

When multiple users are simultaneously logged in, the repository and tool also take care of locking issues, preventing one user’s changes being overwritten by another user who has not seen them. As we saw in Section 14.3.5, the level of granularity of such locks should be as fine as possible to allow the maximum permeability and freedom of use. Changing an element should not prevent others from seeing the most recent committed version of that element: developers are used to working with the latest published version of interfaces or components.

Multiuser repositories are found in MetaEdit+ and GME, and at least MetaEdit+ fulfills these requirements (lack of experience with large-scale use of GME prevents us from talking about it in more detail). Both tools also have a single-user version, and both make the difference between the versions largely invisible to users. This indeed is a useful goal: it should be possible for a user to work on his area of a shared project mostly as if he were the only user. Making this possible requires sensible decisions in the meta-metamodel, the metamodel, and the models.

Software development has found modularization to be a powerful tool, and models too should be separated along lines based not only on functionality but also on developer responsibility. The modeling language and process should be designed to make this possible. With a DSM environment that has taken multiuser access into account in its meta-metamodel, achieving this tends to be relatively easy: the hard work has already been done. Any issues caused by the particular metamodel made for the modeling language should be revealed by the pilot project.

As with most multiuser databases, the repository will probably be stored physically as a set of files on a server. Although there are multiple files, the division into files is determined more by internal details of the tool, repository, or operating system than by any perceived semantic borders in the models. The files are thus treated as forming a single conceptual whole.

15.2.3 Multiple Repositories

For the smallest projects, or areas where only a little simple reuse of whole models is used, disjoint files may prove suitable. For larger projects, or where more reuse is required between models in the project, multiuser repositories are ideal. For the largest projects, or large projects with disconnected teams, the project can be split into disjoint multiuser repositories.

In the simplest case, each repository only contains the models made by its team. Each team works on its models, releasing the results as code and documentation to a central version control system, from where they are replicated to other teams. The interface between teams should be slight in this case, as it will be limited to the legacy approach of design documentation and code. (Admittedly, the situation is thus no worse than it has been earlier, but it will certainly appear worse when it sits next to the shiny new DSM solution!)

In a more complicated case, there may be a set of core models that all users should have access to. These can be maintained in a separate repository and periodically exported and distributed to all satellite repositories. There they are imported, making them available for transparent access in those repositories. Since that access will almost certainly include reusing and referring to elements from those models, a simple import and export will not suffice.

When an updated version of the core models is reimported to the satellite repositories, the existing elements must be updated in place, so that any references from the local models to the core models will now point to the updated elements. Based on our experiences with a system like this, which has been in MetaEdit+ since 1997, such an approach works well, provided it is used as part of a defined process. Uncontrolled import and export in all directions is more likely to lead to a mess than to the miracle of making disconnected repositories appear connected, by automatically merging their differences into a cohesive whole.

15.3 MODEL VERSIONING

In moving from a single-user, separate-file approach to a multiuser repository, many practices need to be reexamined. A repository allows true reuse of elements across multiple models and multiple users. The smallest actionable unit is thus no longer a file—often corresponding to a model—but an object or even property in a model. Locking and versioning thus move away from the compromises required by files toward new solutions.

Looking at the origins of file-based version control from the days of C and FORTRAN, the real units of reuse tended to be functions within a file, rather than the file itself. The units of reuse do not directly correspond to the larger units of versioning or locking.

With object-oriented languages, we tend to think of the class as the unit of reuse, and often a class corresponds to a file. However, little actual reuse takes place at the level of the class itself: normally, just a single instantiation operation refers to it. Instead, the majority of references are to the operations and attributes within the class, so the situation is largely unchanged from early version control systems.

In models in a repository, the units of reuse correspond directly to the units of locking. In general, however, they do not correspond to visible units of versioning: the repository always reflects the current state of the whole project. An explicit version is made as a snapshot of the status of the whole repository. The difference in scale between the smallest units of reuse and the units of versioning is thus larger than with textual programming languages.

Are there any benefits to be gained by having these units aligned, or any disadvantages if they are not? Aligning reuse with locking seems useful: nobody can change the thing we are referring to. This removes problems similar to those at the end

### MODEL VERSIONING

of Section 15.2.1. Having such a large unit of versioning however tends to give cold shivers to most programmers used to source fileversion control. File-based versioning has been the way they have accomplished many tasks, and so the lack of it makes working with models seemingly impossible. Below we will look at the most common questions in the context of a multiuser repository: where repositories are single user or models are stored in files, little changes from traditional practices.

15.3.1 But How Do I Just Save My Work?

Normal development in file-based version control consists of checking out your own file or files for changes, then working with them for a while. If the results are satisfactory, you publish the changed files back to the version control system. If the results show this was a bad idea, you throw away the changes and release the checked out files. If you want to try again, you revert your checked out files to the versions in version control (possibly also updating to newer versions of other files).

With a repository, you open the models you want to work on. As you modify the models or elements, or alternatively when you open them for editing, you are given the lock on those models so that nobody else can change them. If the results of your editing are satisfactory, you commit your transaction. This publishes your changes to the repository, so other users can see them. It also updates your view of the repository to include any changes committed by other users. If these changes were a bad idea, or you want to try again, you abandon your transaction. This discards the changes, and as with commit it also updates your view of the repository.

15.3.2 But How Do I Branch?

Branching is used in code-based development to handlevariants. Normally, many files will need to be branched to support a pair of variants, and each of those files must proceed in two parallel version branches. Because of the large costs associated with this, branching can only be used for a small number of variants.

With DSM, most situations that earlier required branching no longer occur. The modeling language factors out these changes, sometimes into the generator or domain framework and sometimes into small property or other changes in a top-level configuration model. DSM can thus support a much greater range of variants and expresses the variants in the simplest way for the modelers.

Even in the worst case, the branching is relatively painless: shallow copies are made of the relevant models, all referring to the exact same elements as before. The elements that must be changed are replaced individually by new elements (perhaps shallow copies for minor changes). A new top-level configuration model brings together the set of models needed for the variant. The variants are thus made explicit in the models and the coherent view of the project includes these variants, rather than only ever seeing one variant at a time. This broader view helps keep variants working when changes are made elsewhere, and also makes it easier to spot ways in which this method of handling variants could be improved towards one that requires less work.

15.3.3 But How Do I Merge Parallel Changes Made to the Same Model?

Deciding on the boundaries between files is an important task in code-based version control. The project architect will try to avoid the situation where the same file contains parts that will be worked on by multiple people. Always, however, cases like this will occur, even if the version control system uses some kind of locking to prevent them. Since the files are just text, developers can move them around and access old copies outside of version control. Merging multiple versions of the same file into one version containing all changes is a difficult task and has given rise to a whole mini-industry of diff and merge tools. None of these is as intuitive and informative as developers would like, but almost all of them are better than attempting the same task manually.

With a model repository, this problem simply does not occur: elements are locked when changed and no other user can change them until he has loaded the committed version of the user who obtained the lock. While normal modularization is good practice in DSM too, in a good tool it is perfectly possible for several users to edit different objects in the same model at the same time: each object is locked separately, so everyone can obtain the lock on just the object they need.

15.3.4 Where Do the Version Comments Go?

Version comments are an important part of development. In our experience, we refer frequently to version comments in both code- and model-based development. They provide information about why something changed, rather than just what changed, and allow a much quicker overview than even the best diff tool.

In a version of a multiuser repository, the number of changes made by all developers across all models will be too large and too disparate for some of the normal uses of version comments. Instead, it is better practice to keep version comments as part of the models, either as a visible object or as a property of the model. This is of course a practice familiar to many from programming, and brings the added benefit that the version history remains part of the model, even after export and import to another repository. The individual model version comments since the last repository version can easily be collected by a generator as the comment for the version of the whole repository.

15.3.5 How Does It Work in Practice?

Our experience is that versioning has simply been a nonissue in all the projects we have worked on. Of the material in this chapter, customers ask far more questions about how to build modeling languages and models effectively to allow reuse. Most customers have simply zipped up the repository for storage in version control, as version control systems generally require a versionable unit to be presented as a single file.

Where customers have decided to use a multiuser repository, the only issues encountered were technical ones during installation, for example, how to help

### SUMMARY

customers correct name server misconfigurations within their organization so that client computers could communicate with the repository server. Where customers opted for single-user repositories, a single feature, application, or project was assigned to a single developer and stored in a single repository. In a few cases, an individual repository grew too large to be managed by a single developer, and so those cases simply moved to using multiuser access to that repository when extra developers were added.

15.4 SUMMARY

DSM already offers large amounts of reuse and productivity through the work done in the modeling language, generators, and domain framework. The amount of reuse can be increased still further by reusing models and model elements. Storing models in a multiuser repository allows the greatest scope for reuse, makes it easiest, and avoids many of the problems familiar from reuse by name reference in code-based development. Where necessary or desired, work can also be split over several repositories, each used by a smaller team, or even one or more repositories per developer. These more disjoint approaches inhibit reuse and make the process of using DSM more like traditional programming. While this may not be a good fit, there may be some transitional benefit from the more familiar process.

Attempting to force DSM into the file-based model, with multiple files per developer and a single model per file, appears to be a poor solution, at least with current tools. The main expected benefit, better integration with code-oriented version control systems and practices, seems not to be achieved if any reuse is allowed.

Having large repositories as a unit of versioning was expected by some to lead to problems: previous code-based development needed version control systems with small units of versioning to solve some of its common problem situations. Experience shows that with DSM those problems either do not occur or are solved more effectively here in other ways.