Worst Practices for Domain-Specific Modeling

Steven Kelly and Risto Pohjonen, *MetaCase*

Little guidance exists on creating domain-specific modeling languages. Learning what not to do—including how to deal with common pitfalls and recognizing troublesome areas—can help.

In computing's early days, language creation was a common activity, but by the millennium's end it was relegated to a few gurus. Early articles[1] citing "1,700 special programming languages" or hundreds of modeling languages were smothered by the ubiquity of languages such as Java and UML. The current decade has seen a resurgence of interest in domain-specific languages, particularly domain-specific modeling (DSM) languages. The reasons for this renewed growth include the availability of tools to create and work with such languages, and the frequency of cases in which productivity increased by a factor of 5–10 with the introduction of DSM.[2]

All too often, however, language developers have had to fly by the seats of their pants because little material is available to teach them how to create a good language. Although industrial books offer solid background on why we need such languages,[3,4] and academic research offers theories and analysis of them,[5–7] both fields mostly omit instruction on how to actually build them.

There are a few good guides to creating a DSM language, including articles[8,9] and a recent book.[2] Still, many readers are left feeling uncertain, and many languages repeat basic mistakes. Perhaps in language creation, as in music, it's easier to teach what not to do and thus help even first-timers create something acceptable. At the least, knowing what to avoid can be a valuable addition to a set of best practices, enabling language developers to recognize troublesome situations early and thus save themselves from later rework. Here, we outline the common pitfalls, focusing on language creation and use; length restrictions prevent us covering generators or wider organizational issues.

Method Overview

We've identified several worst practices during our experience over the years. To refine our categories, we analyzed 76 DSM cases. This sample is relatively broad, spanning 15 years, four continents, several tools, around 100 language creators, and projects having from three to more than 300 modelers. Among the problem domains are automotive, avionics, mobile, medical, consumer electronics, enterprise systems, system integration, and server configuration. Solution domains include assembler, Basic, C, C++, C#, Java, JavaScript, shell scripts, Python, Prolog, Matlab, SQL, and various XML schemas. That said, the sample does contain a preponderance of cases in Europe or involving MetaEdit+, which is somewhat excused by the

 0740-7459/09/$25.00 © 2009 IEEE

Authorized licensed use limited to: Konkuk University. Downloaded on August 02,2010 at 08:18:30 UTC from IEEE Xplore. Restrictions apply.

fact that these conditions probably accounted for the majority of DSM cases worldwide.

We present the worst practices here in the order you'd encounter them over the life of a project: the initial starting conditions; the domain concept sources; the resulting language; the language's notation; and the language's use. We also list the percentage of cases in which we observed the practice. Because a single case might exhibit zero or many worst practices, percentages might not sum to 100 percent. Finally, we changed some details in example diagrams to protect the identities and rights of those involved.

Initial Conditions

Even before language creation begins, wrong attitudes and decisions can have a serious effect on later success.

Only Gurus Allowed

Believing that only gurus can build languages (4 percent) or that "I'm smart and don't need help" (12 percent)

Decades of experience with theoretical fundamentals, software systems, and language creation might be helpful when developing general-purpose languages. However, such a background isn't the key success factor when developing DSM languages. Because DSM languages try to solve fewer problems than general-purpose languages, they're typically simpler to create. They're not, however, simplistic; they require in-depth understanding and experience with the problem domain. So, appropriate domain expertise is more important than knowledge of language theory.

The other extreme to avoid is trying to do everything yourself, ignoring other people's expertise on how to make good languages. Although it's good for organizations to view their own resources as the key element for developing their DSM language, excessive complacence and a "not invented here" attitude can prove counterproductive. The cruel truth is that, without help, everyone's first language—like everyone's first program—is unlikely to be a masterpiece.

Lack of Domain Understanding

Insufficiently understanding the problem domain (17 percent) or the solution domain (5 percent)

Creating a DSM language requires a good understanding of the problem domain. Normally, this shouldn't be a problem, but occasionally companies make the mistake of delegating the task to a summer intern, or seasoned developers take it on and fail to lift their noses above the level of the code. The language must also set a reasonable boundary around the kinds of applications to be built, sparing at least a thought for future expansion.

Other possible problems when assembling domain concepts into a language include a lack of conceptual or abstract thinking skills or a lack of experience in building nontrivial systems. Such skills can come from fields other than programming. However, programming is perhaps the best teacher because it offers a good vocabulary for principles such as *DRY* (don't repeat yourself; that is, avoid duplicating code or data) and modularization (aim for high cohesion and low coupling between system parts). These principles are at least as necessary when building a language as they are when building an application.

Although creating a DSM language should focus on the problem domain, inexperience in the solution domain can cause problems later. The best DSM language creator is an experienced developer who focuses only on the problem domain, but lets his solution domain experience inform his choices among otherwise equally viable solutions.

Analysis Paralysis

Wanting the language to be theoretically complete, with its implementation assured (8 percent)

The motivation for this kind of mistake is rather obvious: fear. For most of us humans, it's rational to be cautious when entering unfamiliar territory, such as creating a language for the first time. Another form of this problem is a desire to solve every possible problem: that is, a tool isn't useful unless you can use it for everything.

DSM isn't about achieving perfection, just something that works in practice. It will always be possible to imagine a case that the language can't handle. The important questions are how often such cases occur in practice, and how well the language deals with common cases. To avoid analysis paralysis, concentrate on the core cases and build a prototype language for them.

The Source for Language Concepts

The first step in building a DSM language is identifying its concepts. The problem domain is the ideal source; relying too much on secondary sources is a recipe for trouble.

UML: New Wine in Old Wineskins

Extending a large, general-purpose modeling language (5 percent)

Although it's obviously tempting to build on an established language's constructs and semantics, such languages are typically too generic and broad

Because DSM languages try to solve fewer problems than general-purpose languages, they're typically simpler to create.

Authorized licensed use limited to: Konkuk University. Downloaded on August 02,2010 at 08:18:30 UTC from IEEE Xplore. Restrictions apply.

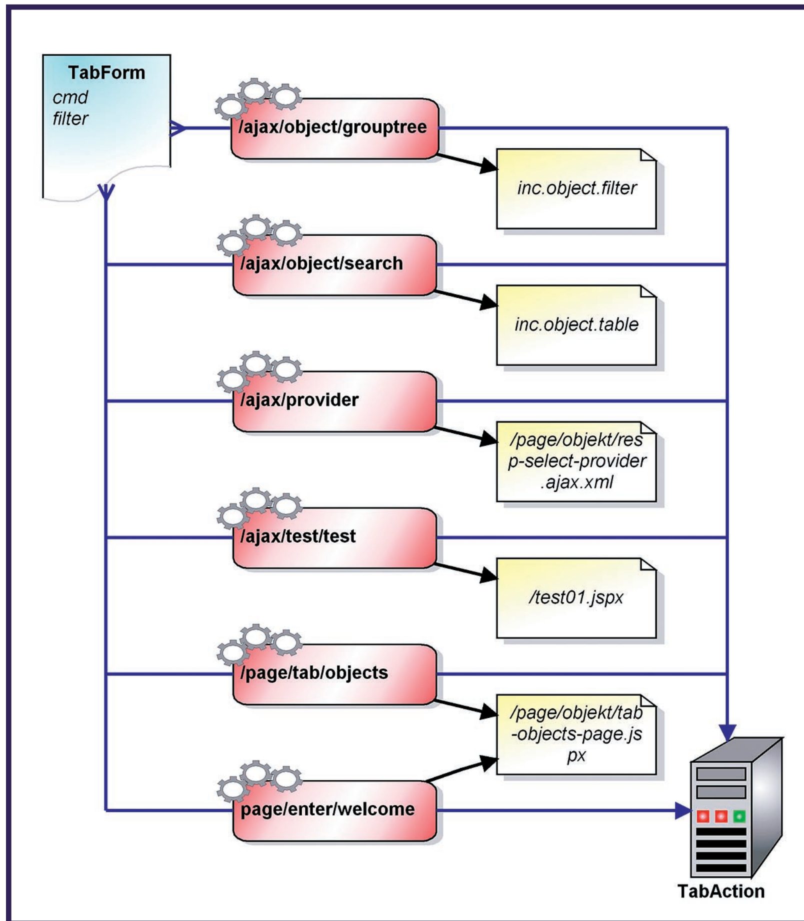

Figure 1. Focusing on framework code. Overemphasizing the target framework or component library can result in low-level details and unnecessary duplication.

for any specific domain. Stripping off parts of the original language and adding new concepts and semantics is often more work than simply starting from scratch. That said, it's obviously good to reuse the basic ideas and concepts of established languages, such as states, data flow, control flow, and inheritance.

In theory, the opposite is also possible: an existing language might be too small or narrow. In practice, however, this seems uncommon and is easier to correct by extending the existing concepts.

3GL: Visual Programming

Duplicating the concepts and semantics of traditional programming languages (7 percent)

Although incorporating programming language concepts such as choices or loops in DSM languages can be useful, you shouldn't let them become the core concepts at the expense of those in the problem domain. The peril in this case is to end up with generic visual programming instead of DSM, leading to a language with a poor level of abstraction. Visual programming languages of this type often have lower expressive power and are more difficult to use than the manual code they're designed to replace.

Code: The Library Is the Language

Focusing the language on the current code's technical details (32 percent)

Although you should derive the modeling language concepts primarily from the problem domain, some solution domain influence is acceptable. However, if the language overemphasizes the target framework or component library, it can drag the abstraction level down toward the code level, preventing retargeting to other platforms. This directly opposes DSM's idea of achieving the best possible level of abstraction for software development. Solution-domain-based languages often expose the implementation details and repetition common in code. Figure 1 shows an example of both: each object pair in the middle could be replaced by a single object, with the implementation details abstracted out.

This was the most common worst practice in our sample, which is hardly surprising when you consider the domain framework's role. At the beginning of a language development project, a framework often represents the solution domain's best existing abstraction; it's also well understood by the domain experts and familiar to the programmers. Given this, a framework is a plausible candidate for the language concepts, but it's typically best to return instead to the source: the problem domain itself.

Tool: If You Have a Hammer ...

Letting the tool's technical limitations dictate language development (14 percent)

Ensuring good tool support for a language is an important aspect of its development, but focusing on tool issues or getting trapped into seeing the world through the tool's limitations is a mistake. Different DSM tools have different emphases, and not all tools support all parts of DSM equally well. Using a poorly suited or weak tool can lead you to make decisions on the basis of what the tool supports, rather than what's needed for the problem domain or the modelers. Figure 2a shows an example where a tool led even an experienced developer to create a language for menu structures that's hard to read and use; Figure 2b would be clearer. Also, practices you learn as workarounds for weaknesses in one tool can all too easily be carried over when you work with another tool that's stronger in that area.

Similarly, people often get carried away with a

Authorized licensed use limited to: Konkuk University. Downloaded on August 02,2010 at 08:18:30 UTC from IEEE Xplore. Restrictions apply.

tool's new or cool features at the expense of getting the language's substance right. A sound foundation has more effect on a language's usefulness and success than do the latest bells and whistles. Also, don't feel obliged to use all tool features: just because a tool supports something doesn't necessarily mean it's a good idea.

The Resulting Language

Building a language is a balancing act between a number of forces, both technical and psychological.

Too Generic/Too Specific

Creating a language with a few generic concepts (21 percent) or too many specific concepts (8 percent), or a language that can create only a few models (7 percent)

Finding the proper generic-specific balance is a key success factor in DSM development—and is thus a rather common place to make mistakes. Developers often create a language that's too generic for its domain, with concepts and semantics that are either too few, too generic, or both. In Figure 3, for example, adding the concepts of "lights" and "heating" would improve the language. A good benchmark here is to see whether you can use your language to model in domains other than your target problem domain. If so, your language is probably too generic.

The other extreme is a language with too many concepts, which are probably too narrow semantically or overlap. This creates problems during language deployment and use; overly complex languages are difficult to learn, master, and maintain.

An interesting variant on the theme of genericity is a language that enables users to create only a few potential models. DSM solutions are mass-production environments first and foremost; if users can't create many applications, building the language might be a waste of effort.

Misplaced Emphasis

Too strongly emphasizing a particular domain feature (12 percent)

By definition, DSM languages should have a strong emphasis on the domain concepts. Unfortunately, language developers can stretch this good practice too far by focusing on a particular feature or concept at the expense of others. This is especially troublesome if that concept has little or no value for the DSM solution. Typically, such a situation arises when you let too many stakeholders influence the language development. It's good to listen to different voices to understand the domain and the prospective language usage, but you should always retain a clear vision of the language's "big picture" and objectives.

Similarly, some developers might be tempted to put every domain element into the language,

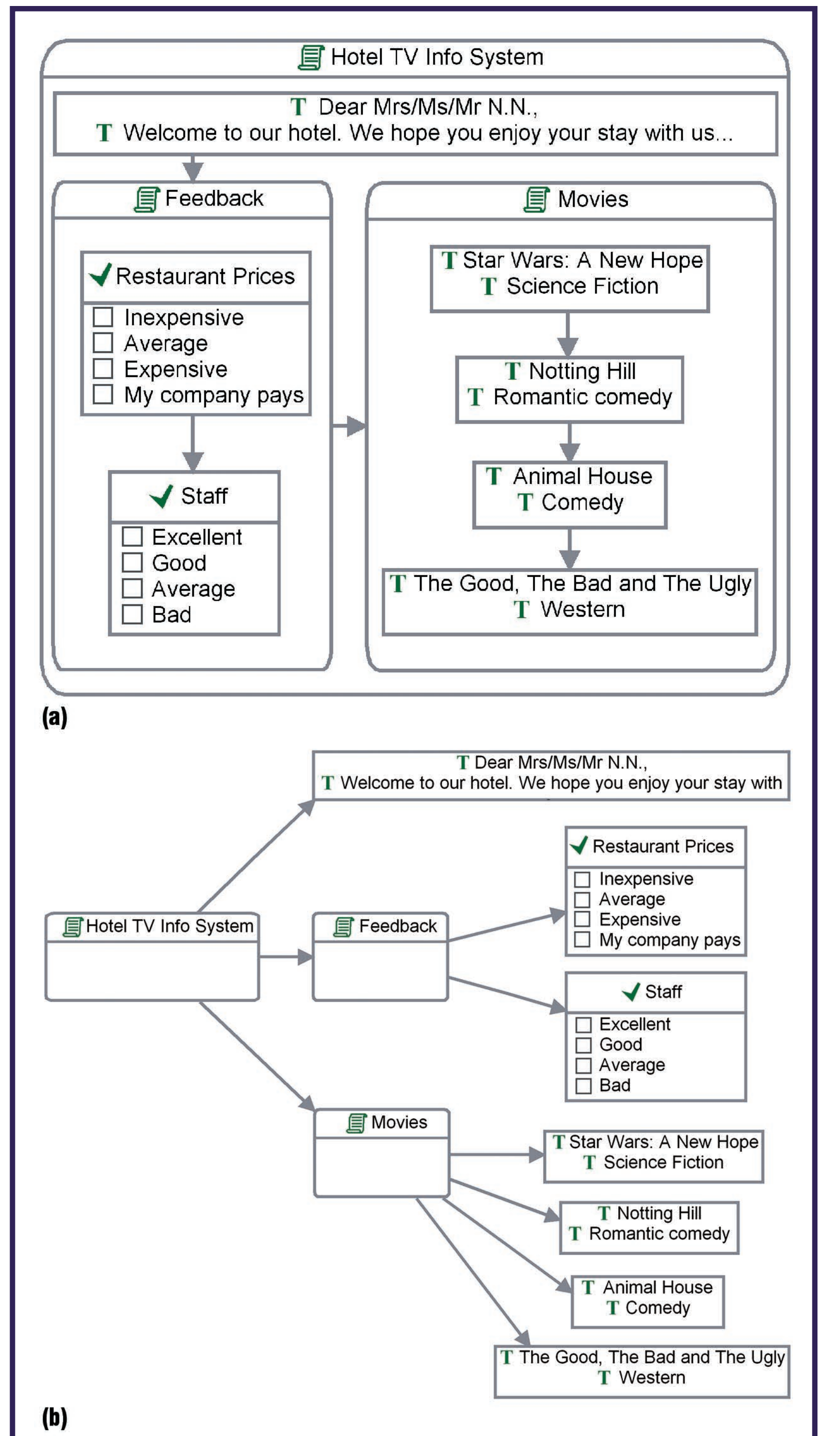

Figure 2. Tool choice and outcomes. (a) A tool focused on strong containment leads to an odd, labor-intensive model structure. (b) Replacing the visual containment with relationships makes the menu structure clearer.

Authorized licensed use limited to: Konkuk University. Downloaded on August 02,2010 at 08:18:30 UTC from IEEE Xplore. Restrictions apply.

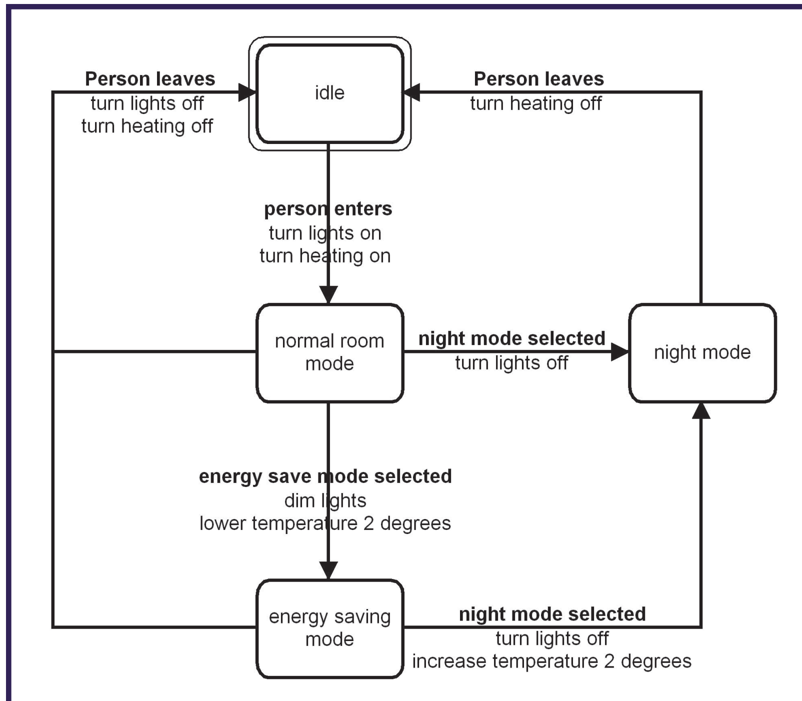

Figure 3. Insufficient concepts. This language has too few concepts, and they're too generic for this domain. Adding explicit concepts for "lights" and "heating" would improve the language considerably.

forgetting the importance of deciding what not to incorporate. Many DSM cases are essentially software product lines, and their languages should model variability—you can omit any commonalities among all products, handling them instead in the generators or domain framework.[4]

Sacred at Birth

Viewing the initial language version as unalterable (12 percent)

This rather common mistake occurs for several reasons. Most of us don't like the idea of "build one to throw away" and are thus reluctant to discard or radically modify our first draft. People often view language creation as a waterfall process, neglecting its iterative nature and the need for prototyping. This mistake can also result from spacing development milestones too far apart. In this case, language creators often invest too much effort into a development step without testing the language in real life, which makes it difficult to step back if needed. Tool support plays an important role here: inflexible tools often lead to extra work in rebuilding models when the modeling language changes.

Language evolution is inevitable, and modifying a language is easier when only a few people know it and only a few models exist. The language is also less proven at this stage, so there will be more flaws and more room for improvement.

Language Notation

A poorly chosen concrete syntax will drive users away, stopping them from using even the most wonderful language.

Predetermined Paradigm

Choosing the wrong representational paradigm on the basis of a blinkered view (7 percent)

Many people approach DSM with a fixed idea of how to represent systems, such as through text or graphical diagrams. Although 75 percent of the general population reportedly prefer visual rather than textual representations,[10] a higher proportion of developers might be predisposed to choose text given its traditional prevalence in programming. Choosing either representation purely on the basis of prejudice is bad, as is ignoring other possibilities such as matrices, tables, forms, or trees. The correct representational paradigm depends on the audience, the data's structure, and how users will work with the data. Making the wrong choice can significantly increase the cost of creating, reading, and maintaining the models.

This error is almost certainly underreported in our sample because, of the available tools, MetaEdit+ supports the widest variety of representational paradigms. Also, developers who prefer text might have self-selected themselves out of the sample by using a simpler, purely textual editor.

Simplistic Symbols

Using symbols that are too simple or similar (25 percent) or downright ugly (5 percent)

One of the most common failure areas is in the language's notation—its symbols or icons. Unlike more abstract or general-purpose languages, DSM languages can often find familiar, intuitive representations directly from the problem domain. All too often, however, the symbols for different language concepts are just boxes with the concepts' names as labels. People recognize things by their shapes, not by labels (if you doubt this, stick the label "lemon" on a banana and see how people react). Also, symbols differing in color alone are suboptimal: the brain views color change primarily as a different version of the same thing, not as a completely different thing. Figure 4 shows an example of both mistakes.

Alan Blackwell has shown that the best symbols are pictograms, not simpler geometric shapes or more complex bitmap or photographic representations.[11] Although our sample contained no cases with overly complex bitmap symbols, you should avoid these as well—bitmaps scale poorly (particularly with aspect-ratio changes) and have little room for text or other contents.

Authorized licensed use limited to: Konkuk University. Downloaded on August 02,2010 at 08:18:30 UTC from IEEE Xplore. Restrictions apply.

Symbols have an aesthetic role, and few people are fortunate enough to have both the abstract thinking that language design requires and the artistic skills needed to create great symbols. Not surprisingly, the few truly ugly languages in our sample encountered significant opposition from users. Take such opposition seriously: find someone with decent graphic design skills to improve your symbols.

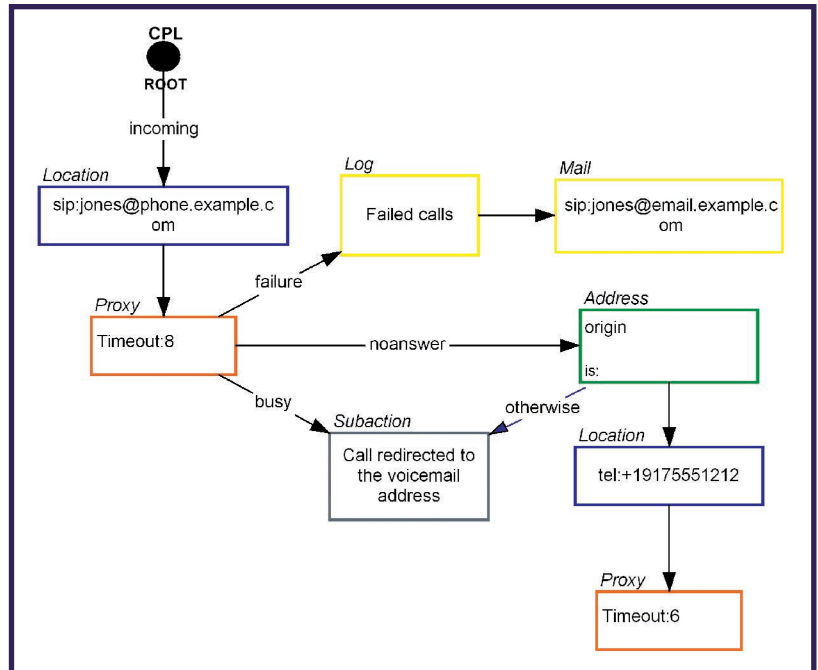

Figure 4. Inadequate symbol differentiation. Symbols differing in only color and label are insufficient. Research shows that the best symbols are pictograms rather than simple geometric shapes or photorealistic bitmaps.

Language Use

All too often, language creators forget that languages are made to be used and to serve their users. Percentages here are only of those languages that have already seen significant use by people other than their creators.

Ignoring the Use Process

Failing to consider the language's real-life usage (42 percent)

It's notoriously hard to predict how people will use a new system or how group members' individual efforts will interact when brought together. Language developers ignore this topic at their peril: To have any value, the language and its use process must serve the modelers. This category involves five areas of concern.

First, generally, multiple people will use a DSM language to make multiple models. To avoid having modelers reenter or copy-and-paste the same information multiple times, plan for reuse and referencing among models in advance. Models that interconnect should do so with minimal coupling. Data duplication and a lack of modularization invariably lead to maintenance nightmares. Figure 5 shows a particularly unpleasant example: The user copied the whole model to achieve a variant without the small time-out object on the left. Instead, the language could have offered concepts for reusing models or made the generator or framework ignore time-out objects on platforms that don't support them.

Second, semiautomated model transformations help users create more data quickly, but with poor long-term results. Unlike full transformations, users must maintain the extra data by editing generated source code or model transformation results as in MDA (model-driven architecture). Anything that transformations can create automatically can be created at generation time, avoiding the maintenance burden and letting transformations change freely over time.

Third, language creators often try to prevent modeler error by creating myriad strongly enforced rules that serve only to annoy, preventing modelers from breaking the rules even temporarily while they're changing their models.

Fourth, unsurprisingly, developers using DSM often uncritically apply processes that have evolved to support source-code-based development. Many such practices are simply crutches and bandages evolved to fix problems inherent in source code and its single-user editing. Repository-based multiuser editing works much better for models, as does a proper modularization and division of labor.

Finally, debugging DSM models at the source-code level is a bad idea if the structure of models and source code differ significantly. When the model-to-code mapping is unclear, it's hard to know where to insert a breakpoint in generated code. When the code-to-model mapping is unclear, it's hard to correct a bug found during debugging. It's better to have running code call back to the modeling tool to highlight the current symbol, and let the modelers set breakpoints there.

No Training

Assuming everyone understands the language like its creator (21 percent)

Although the use of familiar domain concepts makes DSM languages easier to learn, it doesn't mean users will immediately understand them completely. Language creators often overlook this fact and become disconnected from the modelers. The

Authorized licensed use limited to: Konkuk University. Downloaded on August 02,2010 at 08:18:30 UTC from IEEE Xplore. Restrictions apply.

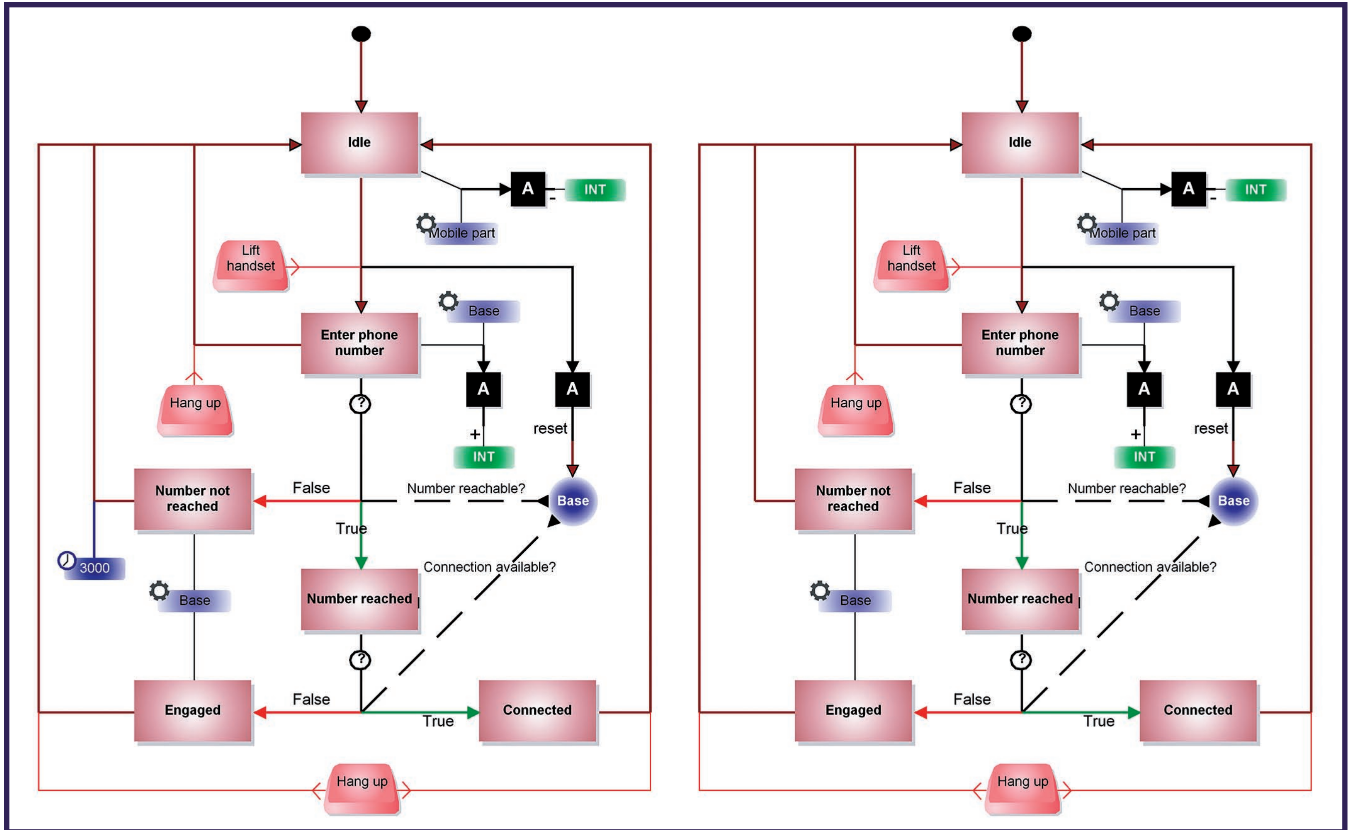

Figure 5. Poor planning for reuse of models. The modeler in this case had to copy the entire diagram to account for a minor variation: the small time-out object on the left.

task of language creation doesn't stop when everything works: you must create documentation and training materials and communicate them to users. DSM research indicates that failures here lead to problems and long-term resistance, even when support later improves.[12] As with any project, it's worthwhile to involve users early, both to get practical feedback and to achieve smooth acceptance.

Post-adoption Stagnation

Letting the language stagnate after successful adoption (37 percent)

Successful adoption of a DSM language implies many models and modelers. The greater the number of models and modelers, the harder changing the language is. Although the best tools can automatically update models when the language changes, you can't automatically update the modelers' brains.

Fortunately, our experience indicates that the problem domain changes that affect a deployed language tend to be additive—that is, they involve new concepts or concept extensions that both modelers and tools adopt with relative ease. To avoid language stagnation, you should make such changes promptly rather than postpone them. You should also avoid passing off language maintenance to someone unsuited to the task.

After several years, a problem domain might change sufficiently to create problems. (However, this situation is rare.) Trying to shoehorn such changes into the old language might not work, and a massive update of the modeling language and all models might be impractical. Another option is to create a new language for the new domain: the better fit can create increased productivity that often balances out the cost, just as it did when creating the first language.

Preliminary Analysis

Our sample covered 76 cases, mostly of companies using MetaCase as consultants or tool providers; in some cases MetaCase was not involved but we have been able to discuss the case with participants. In all, 7 percent of the cases were carried out by MetaCase alone, 57 percent by the customer with consultancy from MetaCase, and 36 percent with no consultancy from MetaCase. In 15 percent of cases, participants used a tool other than MetaEdit+ (at least initially).

In assessing cases by worst practices, we agreed on landmark cases to determine the watershed—

Authorized licensed use limited to: Konkuk University. Downloaded on August 02,2010 at 08:18:30 UTC from IEEE Xplore. Restrictions apply.

for example, to be counted as "ugly," symbols had to be at least as ugly as case X. We normalized worst practices to questions with either a simple yes-or-no answer or a three-point scale, such as *too generic*, *acceptable*, and *too specific*.

As a preliminary analysis, we calculated the correlation among practices, given below as Pearson's coefficient, *r*, expressed as a percentage. All correlations below are statistically significant ($n = 76$, $\alpha = 0.05$, one-tailed, $|r| \geq .190$), but the relationship's direction and its possible causality are our own interpretation.

The single largest factor that led to a language not being used was when organizations gave the language design task to someone with insufficient experience in the problem domain (26 percent).

Basing the language on code led developers to try to take everything into consideration (33 percent). This desire for theoretical completeness was often accompanied by ascetic symbols (28 percent). Using code as a basis also led to stagnation (37 percent).

If the language developer didn't accept help initially, the language was likely to become sacred (24 percent). Sacred languages were likely to stagnate (31 percent). However, sacred languages were also more likely to be used in practice (35 percent)—perhaps because their developers loved them and pushed for their use.

Using a poor tool required extra effort, so developers were less willing to change their languages and those languages thus became sacred (31 percent). Poor tools also led to languages whose abstraction level was no higher than programming languages (34 percent), while poor facilities for defining symbols led to ugly notation (41 percent). A lack of attention to symbols correlated with insufficient training (47 percent), showing a consistent disregard for users.

Examining our sample cases in relation to the initial set of worst practices helped us tighten up the boundaries between practices and identify some extra facets. We were surprised by the rarity of certain practices—including using existing languages as sources for concepts and making the initial language draft sacred. However, we often try to warn customers about such issues early, and they probably avoided them as a result. It would be interesting and instructive to repeat the analysis for cases with other tools or extend the preliminary analysis with more detail on the division of labor and the language developers' relative experience. The most important result, however, would be if our honesty about these failings in our own cases could help others avoid falling into the same traps.

Acknowledgments

We thank all the MetaCase staff, particularly Juha-Pekka Tolvanen and Janne Luoma, and all the people we've worked with on these cases.

References

1. *Computer Software Issues, An American Mathematical Association Prospectus*, July 1965, quoted in P.J. Landin, "The Next 700 Programming Languages," *Comm. ACM*, vol. 9, no. 3, 1966, pp. 157–166.
2. S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation,* John Wiley & Sons, 2008.
3. J. Greenfield and K. Short, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools,* John Wiley & Sons, 2004.
4. D. Weiss and C.T.R. Lai, *Software Product-Line Engineering*, Addison Wesley Longman, 1999.
5. J. Ralyté, S. Brinkkemper, and B. Henderson-Sellers, eds., *Situational Method Engineering: Fundamentals and Experiences*, Springer, 2007.
6. D. Spinellis, "Notable Design Patterns for Domain Specific Languages," *J. Systems and Software,* vol. 56, no. 1, 2001, pp. 91–99.
7. G. Costagliola et al., "A Classification Framework to Support the Design of Visual Languages," *J. Visual Languages and Computing*, vol. 13, no. 6, 2002, pp. 573–600.
8. D. Roberts and R. Johnson, "Evolve Frameworks into Domain-Specific Languages," *Proc. 3rd Int'l Conf. Pattern Languages*, 1996; www.cs.wustl.edu/~schmidt/PLoP-96/roberts.ps.gz.
9. M. Voelter and J. Bettin, "Patterns for Model-Driven Software Development"; www.voelter.de/data/pub/MDDPatterns.pdf.
10. *Train the Trainer*, Int'l Assoc. Information Technology Trainers, 2001; http://itrain.org/pdf/itrain_ttt_course_outlines.pdf.
11. A. Blackwell, *Metaphor in Diagrams*, PhD thesis, Darwin College, Univ. of Cambridge, 1998.
12. J. Ruuska, "Factors Influencing CASE Tool User Satisfaction: An Empirical Study in a Large Telecommunications Company," master's thesis, Dept. of Computer Sciences, Univ. of Tampere, 2001 (in Finnish).

About the Authors

Steven Kelly is chief technology officer of MetaCase and has more than 15 years' experience building domain-specific modeling tools and languages. He has a PhD in information systems from Jyväskylä University. Contact him at stevek@metacase.com.

Risto Pohjonen is a domain-specific modeling (DSM) consultant and developer at MetaCase, with over 10 years' experience building DSM tools and languages. Contact him at rise@metacase.com.

Authorized licensed use limited to: Konkuk University. Downloaded on August 02,2010 at 08:18:30 UTC from IEEE Xplore. Restrictions apply.