IEEE Software, June/August, 2009.

Worst Practices for Domain-Specific Modeling

Steven Kelly and Risto Pohjonen, MetaCase

JUNBEOM YOO

Dependable Software Laboratory
KONKUK University

http://dslab.konkuk.ac.kr

2010.08.06

DMS (Domain Specific Modeling)

- Software productivity increased by a factor of 5-10 with the introduction of DSM.
- There are a few good guides to creating a DSM language, but not sufficient.
- This paper outlines the common pitfalls, focusing on language creation and use.

Method Overview

- 76 DSM cases
- Spanning 15 years
- 100 language creators
- Projects having 3 ~ 300 modelers
- Domains: automotive, avionics, mobile, medical, consumer electronics, enterprise systems, system integration, and server configuration
- Languages: assembler, Basic, C, C++, C#, Java, JavaScript, shell scripts, Python, Prolog, Matlab, SQL, and various XML schemas

- Presentation order
  - Initial conditions
  - The source for language concepts
  - The resulting language
  - Language notation
  - Language use

Initial Conditions

- Even before language creation begins, wrong attitudes and decisions can have a serious effect on later success.
  - Only Gurus Allowed
  - Lack of Domain Understanding
  - Analysis Paralysis

Only Gurus Allowed

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

- Such a background is important, but they require in-depth understanding and experience with the problem domain. So, appropriate domain expertise is more important than knowledge of language theory.

Lack of Domain Understanding

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

- 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.
- 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)

- DSM isn’t about achieving perfection, just something that works in practice.
- 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
  - 3GL: Visual Programming
  - Code: The Library Is the Language
  - Tool: If You have a Hammer …

UML: New Wine in Old Wineskins

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

- It's obviously good to reuse the basic ideas and concepts of established
- languages, such as states, data flow, control flow, and inheritance.

- Stripping off parts of the original language and adding new concepts and semantics is often more work than simply starting from scratch.

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.

Code: The Library Is the Language

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

- 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 was the most common worst practice in our sample.
- A framework often represents the solution domain's best existing abstraction, not the problem domain

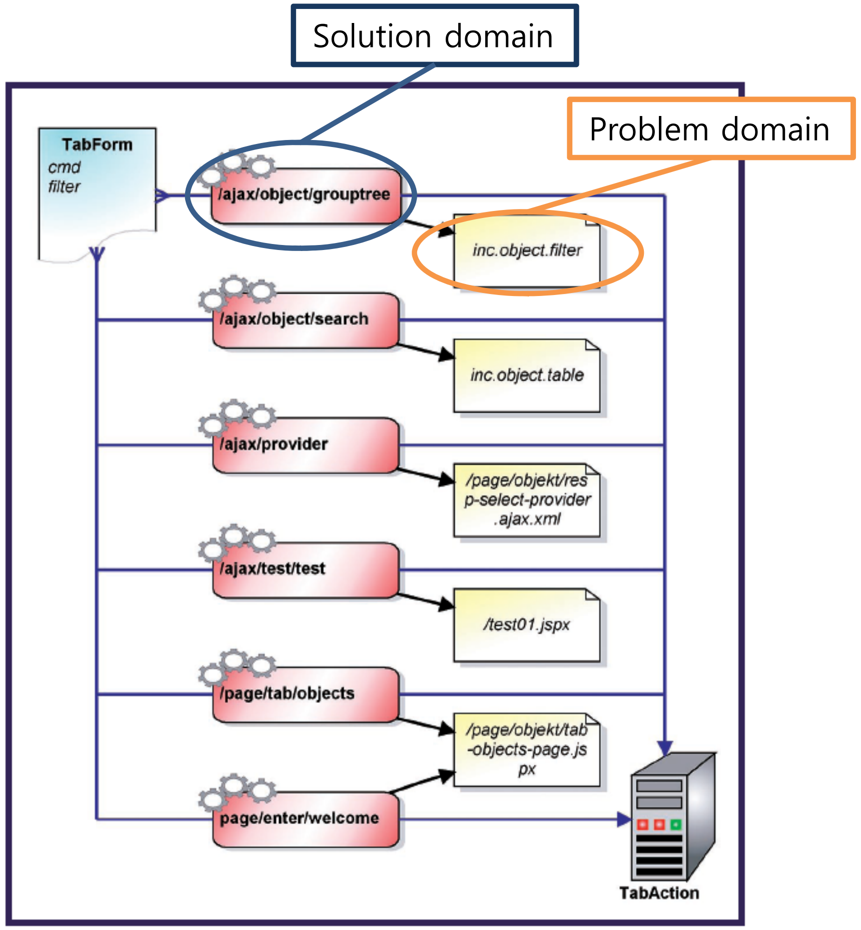

Figure 1. Focusing on framework code. Overemphasizing the target framework or component library can result in low-level details and unnecessary duplication.

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.

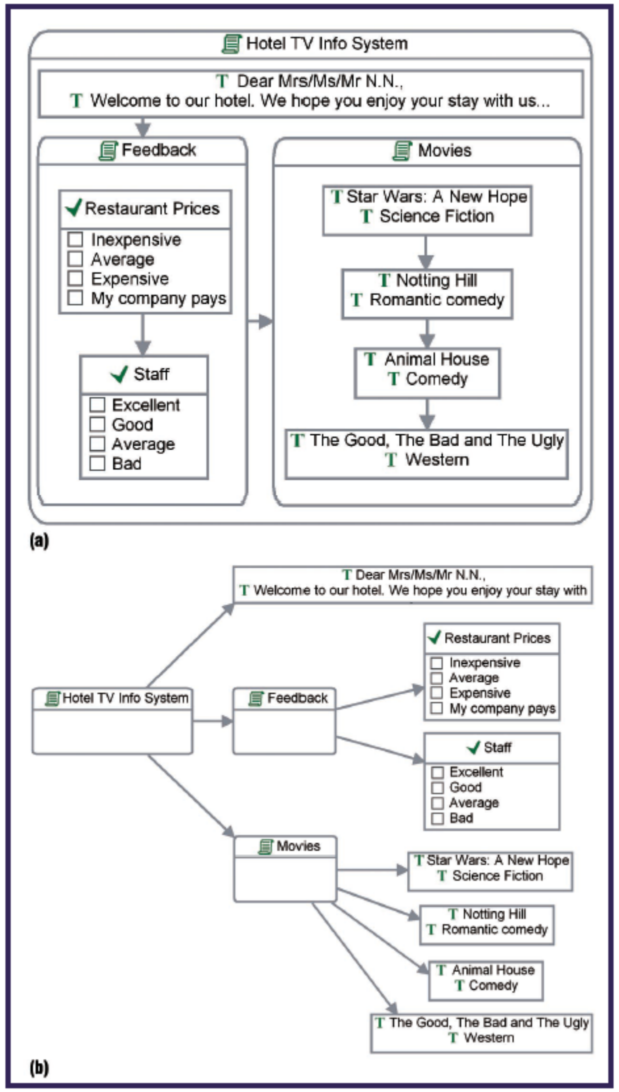

Figure 2. Tool choice and outcomes. (a) A tool focused on strong containment leads to an odd, labor-intensive model structure. (b) Replacing the visual containment with relationships makes the menu structure clearer.

The Resulting Language

- Building a language is a balancing act between a number of forces, both technical and psychological.
  - Too Generic / Too Specific
  - Misplaced Emphasis
  - Sacred at Birth

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.
- The other extreme is a language with too many concepts, which are probably too narrow semantically or overlap.

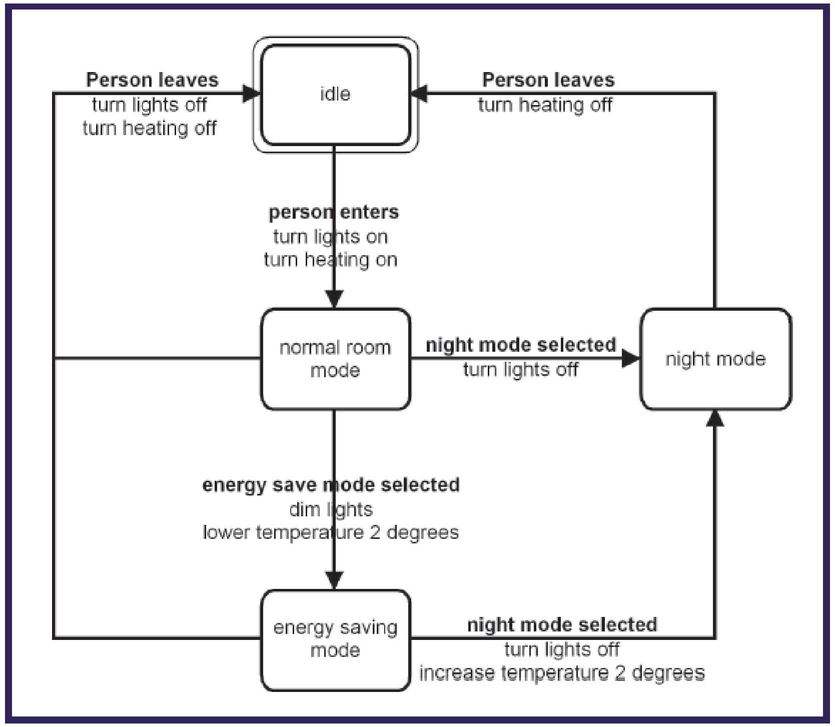

Figure 3. Insufficient concepts. This language has too few concepts, and they're too generic for this domain. Adding explicit concepts for "lights" and "heating" would improve the language considerably.

Misplaced Emphasis

Too strongly emphasizing a particular domain feature (12 percent)

- Language developers can stretch this good practice (domain concepts) too far by focusing on a particular feature or concept at the expense of others.
- Similarly, some developers might be tempted to put every domain element into the language, forgetting the importance of deciding what not to incorporate.

Sacred at Birth

Viewing the initial language version as unalterable (12 percent)

- People often view language creation as a waterfall process, neglecting its iterative nature and the need for prototyping.
- 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.
- Language evolution is inevitable.

Language Notation

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

Predetermined Paradigm

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

- 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.
- MetaEdit+ supports the widest variety of representational paradigms.

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.
- Alan Blackwell has shown that the best symbols are pictograms, not simpler geometric shapes or more complex bitmap or photographic representations.
- Find someone with decent graphic design skills to improve your symbols.

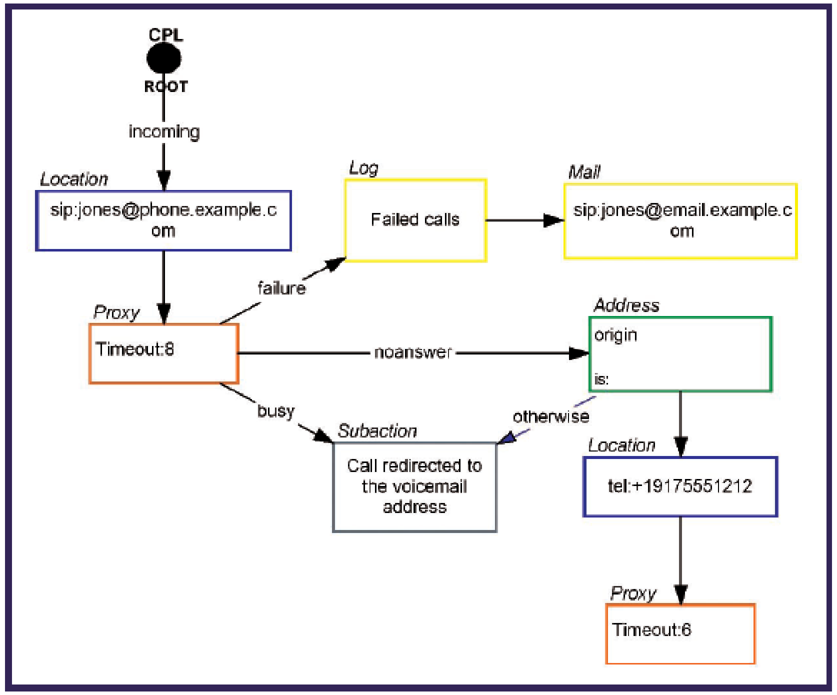

Figure 4. Inadequate symbol differentiation. Symbols differing in only color and label are insufficient. Research shows that the best symbols are pictograms rather than simple geometric shapes or photorealistic bitmaps.

Language Use

- All too often, language creators forget that languages are made to be used and to serve their users.
  - Ignoring the Use Process
  - No Training
  - Post-adoption Stagnation

Ignoring the Use Process

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

- Five areas of concern
  - Multiple people / reuse
  - Semiautomatic model transformation
  - Strongly enforced rules
  - Involving processes for supporting source-code-based development
  - debugging

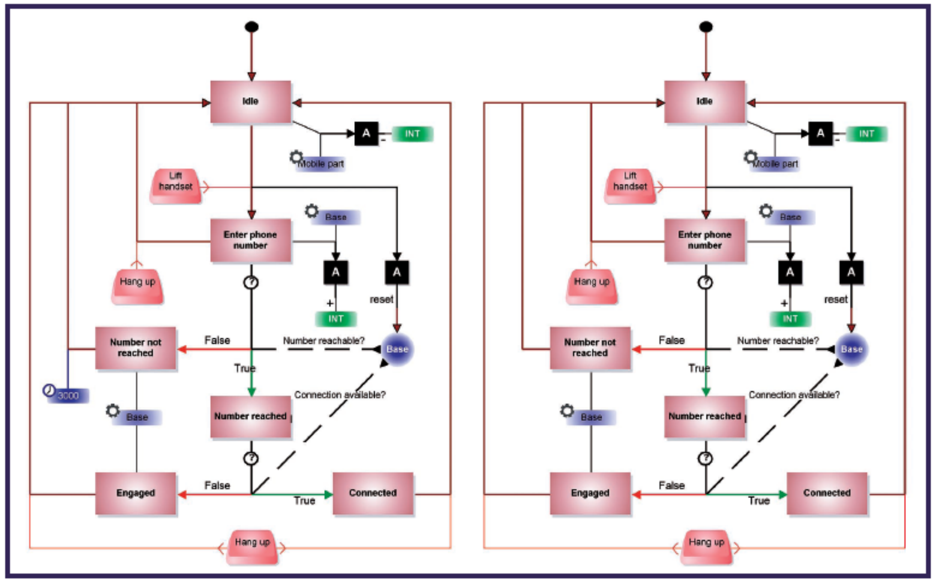

Figure 5. Poor planning for reuse of models. The modeler in this case had to copy the entire diagram to account for a minor variation: the small time-out object on the left.

No Training

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

- 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)

- The greater the number of models and modelers, the harder changing the language is.
- 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.

Preliminary Analysis (1/2)

- 76 cases
- 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).

Preliminary Analysis (2/2)

- Multiple factors are as following:
  - Basing the language on code led developers to try to take everything into consideration (33 percent).
  - Desire for theoretical completeness was often accompanied by ascetic symbols (28 percent).
  - Using code as a basis also led to stagnation (37percent).

  - 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).
  - 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.