The Practice of Deploying DSM
Report from a Japanese Appliance Maker Trenches

Laurent Safa
EMIT Middleware Laboratory
Matsushita Electric Works, Ltd.
1048, Kadoma, Osaka 571-8686, Japan
+81-6-6908-6752
safa at mail dot mew dot co dot jp

Abstract: *Domain-specific modeling (DSM) and code generation technologies have been around for decades yet are not widely used when compared to traditional software development practices of using general purpose languages ("C", Java...) and design techniques (sketches, UML, OOP...). This is surprising considering the availability of mature tools capable of generating product quality application code, configurations, documentations, test suites and other artifacts, from a unique source, a domain-specific model [1]. Why is it so? The problem may lie in the difficulty of integrating DSM into legacy processes and mindsets. Based on real experience in the domains of home automation and embedded device networks developments, we present some key aspects of deploying DSM. After presenting our context of modeling and the rationales behind our decision to use DSM, we describe our approach to the problems of promotion, process integration, usability and sustainable deployment of domain-specific solutions. We conclude with the recognition that most challenges to deploy DSM are not technical but human by nature, and we elaborate on the perceived advantages of using Cognitive Dimensions to help build better domain-specific languages and tools.*

Introduction

The home and building automation divisions of Matsushita Electric Works, Ltd., Japan (MEW) are contemplating a steady increase of software development costs combined with growing difficulties to satisfy quality requirements for appliances. These problems are caused by the constant growth in scope, size, and complexity of the features implemented by way of embedded software, while the fundamental practices and tools have not significantly evolved.

More embedded features	Same old development tools
- Internet connectivity	- Text-based editor
- Multi-media	- Low-level programming language “C”
- Ubiquitous computing	- Limited use of patterns
- Plug-and-play behavior	- Ad-hoc approaches to problem solving
- Peer-to-peer networks	
- Mesh networks	
- Application mashups	

Table 1: Poor practices for today's challenges

MEW has engaged several projects to address this challenge, the so-called “Software crisis”:
- CMM-based software process improvement (SPI)
- Definition of common development platforms and modules
- Deployment of software automation practices and tools

This paper discusses the later project. We start with an explanation of our rationales for selecting domain-specific modeling (DSM). We subsequently describe our promotional approach based on the resolution of measurable problems related to the use of embedded software. We then discuss the issue of process integration and propose a life cycle for software development with DSM. Next we analyze the problem of devising visual languages that do not get in the way of the practitioners, and we introduce the concept of *escape semantics* that enables creative modeling and collegial language construction. Finally we present a test-driven approach to facilitate the deployment of families of custom languages.

The Context of Modeling and the Decision to Use Domain-Specific Modeling

At MEW, new technologies and disciplines are created in the Advanced R&D Lab before being transferred to product divisions to help these enter new markets. The phases of creating appropriate new technologies and deploying these in time to product divisions constitute two major challenges (cf. {C} for creation and {D} for deployment in Diagram 1).

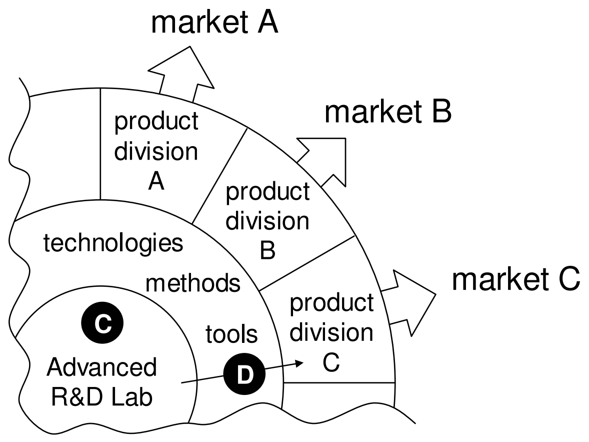

Diagram 1: Position and role of the R&D lab

MEW had mixed experiences with past attempts to use CASE tools and UML-based modeling to facilitate the new technology creation phase {C}.

The following table lists the most significant attempts at using modeling tools to develop better software. Although these were local successes, each one failed to go beyond the category of early adopters.

Tool type	Design method	Generation	Platform	Application
commercial	state-transition matrix	"C"	8-16bit CPU No RTOS	Room control
	UML		16-32bit CPU RTOS	IP routing and filtering
in-house development	sequential functional charts	ASM	4-8bit CPU No RTOS	Remote sensors and actuators

Table 2: Past experiences of software modeling

The first reason invoked by practitioners for the failure to deploy software modeling is specific to off-the-shelf commercial tools: for full code generation, it is necessary to use the tool vendor's underlying framework which raises questions of suitability (does the vendor's framework perform correctly within the product specifications), adaptability (recurrent cost of porting the vendor's framework to new hardware platforms), availability (MEW products for home and building automations have a typical lifespan of 10 to 20 years), and loss of differentiation factor (use of same framework as competitors who purchased the same vendor's tool).

A second reason is specific to UML: practitioners consider UML's object-oriented notations too far apart from the "C" procedural world in which they evolve. Instead of class-centric designs, practitioners think in terms of concurrent tasks, interlock mechanisms, software stacks, data formats and communication protocols.

A third reason was the lack of support over the long period of product development. Innovators did not have sufficient organizational support to pursue the promotion long enough for their new methodologies to be integrated in the organization's development process. Active promotions were abandoned after their initiators were assigned new responsibilities.

After previously promising methods felt short of expectations, users built-up natural defenses against novelty and focused instead on known-practices: assembly language and "C" programming. With these, the team can program on the bare metal, be in control of the detailed implementation, and predictable behavior can be produced.

A corporate language has evolved naturally over the years to express requirements, designs and implementations matters. It has notations, conventions and semantics that map precisely the problem domains, and it evolves incrementally when the problem domain changes as described in following table.

Problem domain change	**Consequence / Response**
Application of Building Automation technologies to the Home Automation market	Downsizing of specifications. Reuse of selected sensors, actuators and communication mediums. Porting of selected software modules and hardware components to lower-end hardware platforms.
Home appliances get connected to the Internet	Addition of Internet protocol stacks for machine-to-machine and machine-to-human communications (TCP, HTTP, SMTP/POP...).
Reduction of single points of failure	Addition of peer-to-peer features to move from top-down hierarchical control to grid-like computing.

Table 3: Examples of corporate response to some domain changes

This corporate language survived all the changes and it evolved just in time at the pace required by practitioners to be used for internal communication and development purposes. It is well understood not only by developers, but also across the board by testing divisions, marketing people, sales people and managers. Models are written in the form of diagrams with free-form graphic tools, or simply tables with text editors.

Format	**Defines**
Table	Message format, Product specifications, I/O map, Memory map
Graph	Network system architecture, Device role, User-interface, Data-flow, Hardware layout
Sequence diagram	Communication protocol, Feature implementation
Sequential functional graph	Input-driven decision logic (decision-tree)
Stack	Software architecture
Bag	Features selection

Table 4: Some concepts found in MEW models (N=13 projects)

We concluded that past failures to deploy software modeling practices were caused principally by the strategy of targeting the fragmented problem of new technology creation with uniform methods (cf. {C} in Diagram 1), the requirement to use notations and concepts apart from the practitioners' concerns, and the lack of organizational support.

Furthermore, although the methods employed to create new technologies are not always optimal, practitioners generally succeed to complete their technical development. However, practitioners often have troubles getting their new technology deployed to product divisions and spread to many development groups, which results in underused software modules. In other words, previous modeling promotion efforts aimed at improving what was working (creation), failing to provide a solution for what was not working (deployment).

With that respect, we decided to focus our new software modeling project on the issue of deploying new technologies to product divisions (cf. {D} in Diagram 1), and to use the on-going CMM-based process improvement effort as our organizational support. To adapt to the needs of stakeholders from various backgrounds, we selected Domain-Specific Modeling (DSM) for its versatility and adaptability. To enable quick development of solutions with few resources, we selected a DSM tool with a metamodeling facility (language definition and visual editing) based on configuration rather than programming. To reduce the risk of losing support over a long period of time, we selected a commercial tool from a well-established vendor. The promotion of general purpose modeling was delegated to our Software Engineering Group (SEG) within the software process improvement project.

DSM Medicine

In the course of our DSM developments, we found evidences of a wide range of problems that can be solved with DSM technologies, although these are not necessarily defined in terms of application code generation. Hereafter we list the problems we encountered, and we briefly describe the DSM solution we proposed to the respective stakeholders.

Stakeholder	**Activity**	**Needs**	**DSM Solution**
Product engineer	Development of systems of systems	Complex system configuration	Generation of configuration files from system model
		Too many misuse cases to take in account	Automated discovery of misuse cases from models
Product developer	Porting of existing software to new hardware (re-targeting)	Quality issues	Injection of generated code into code templates
Product tester	Development of test suites	Test suite development is costly	Automated generation of test suite from models
		Possibility to overlook test cases	
Development team	Product development	Late requirement changes	Provide agility by way of visual modeling and code generation as visual models are closer to the requirements than source code
Development manager	Pass-over of a working software to the product division	Up-to-date design documentations	Generate up-to-date design and architecture documents from the model
R&D planning office	Measure of gap between current development practices and foreseeable market needs	Visibility of current development practices	A central model repository that can be scanned (a special model compilation) to extract information such as usage frequency of a module, of an operating system, of a combination of domain concepts…

Stakeholder	Activity	Needs	DSM Solution
Core technology developers	Develop new technologies for tomorrow's products	Reduce learning curve of innovative technologies to help deploy these to the developers	Reduce learning curve by embedding new APIs and guidelines into the code generator, and by providing a familiar visual language atop of it.
Software Process Improvement Group	Promotion of best coding practices	A method to enforce code layouts, naming conventions, folders conventions, etc…	Automated code generation according to well-defined rules.
Software Security Group	Reduction of software security risk	A method to avoid dangerous code structures (scanf…)	Automated code generation that complies with the corporate security policy
		Difficulty to analyze risks induced by design	A special model compiler that derives risks from the model
Top management	Strategic planning	Costly software development	Software automation
		Difficulty to enforce reuse of common platforms across the company	Automated selection of reusable models

Table 5: A selection of problems that can be addressed by way of DSM

Note that care should be given to select pains (problems) which resolution can be measured to demonstrate progress to both practitioners and management. Our pain killing method is composed of six steps:

- Get embedded into the practitioner team
- Observe the way people work to understand their context
- Ask practitioners for the few problems that most disrupt their core activity
- Select the problems that can be measured
- Present the DSM solution as a pain killer
- Deploy the DSM solution and verify the problem reduction with the practitioners

Process Integration of Modeling

Contrary to what happened with past efforts to promote general purpose modeling, where practitioners questioned the ability of specifying their software particularities, or the opportunity of replacing in-house frameworks with the tool vendor's, we found no such resistance to our DSM effort. The perceived reason is that DSM tools adapt to the methodology in place, allowing us to use the domain concepts and frameworks that practitioners have been developing for years. This seems to corroborate Seth Godin when he writes [2] "a key element in the spreading of the idea is the capsule that contains it. If it's easy to swallow, tempting and complete, it's far more likely to get a good start."

In order to clarify the positioning of DSM into the corporate process, we defined the three activities of creation {C}, deployment {D} and evolution {E}. As illustrated in the following diagram, new technologies are first created using appropriate software engineering techniques {C}, and later deployed to the product domain with the help of DSM {D}. Finally, necessary evolutionary steps {E} are engaged to keep both technologies and DSM capsules up to date with the constantly changing market needs.

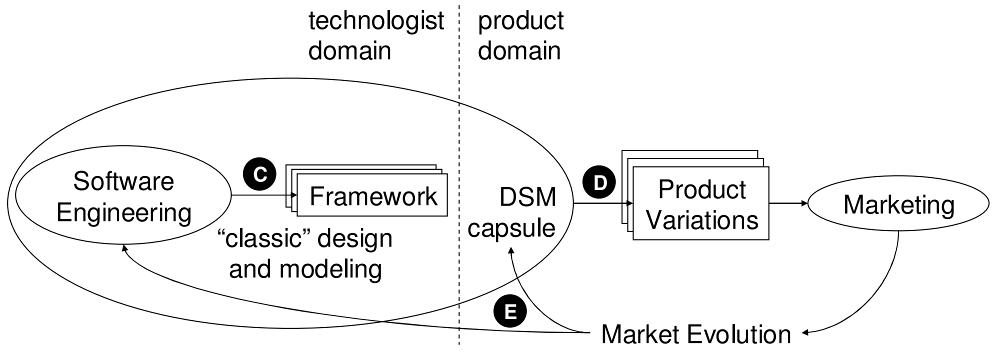

Diagram 2: DSM capsule fills the gap between technologists and marketers

When looking from a life cycle perspective (cf. Diagram 3), the creation activity {C} corresponds to new product developments, while deployment activity {D} represents domain-specific modeling. Finally, the evolution activity {E} maps to the incremental changes applied to both framework and modeling tool to follow the domain changes. Furthermore, this view reveals a well-established practice we have no plan to change: during the fundamental research phase practitioners often use off-the-shelf DSM tools for algorithm research purposes (ex: Simulink®).

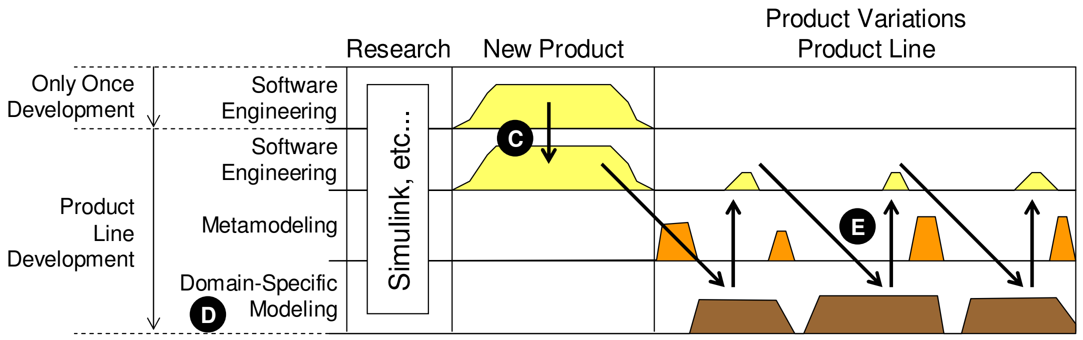

Diagram 3: Life cycle for software development aimed at deployment with DSM

Agile Modeling

Language agility is critical to the tool-smith, because lack of language agility puts the DSM tool at risk of being abandoned by practitioners for more convenient methods. After all, what matters most to practitioners is producing a working product, not using modeling tools.

The following diagram illustrates the gap between present needs, practices and available tools. Due to the metamodeling delay necessary to define visual languages, editors and compilers, the DSM tool lags behind practices, so it is at risk of being perceived as constraining, especially for practitioners used to drawing with free-format whiteboards, pen and paper and general purpose diagram tools like Microsoft© PowerPoint.

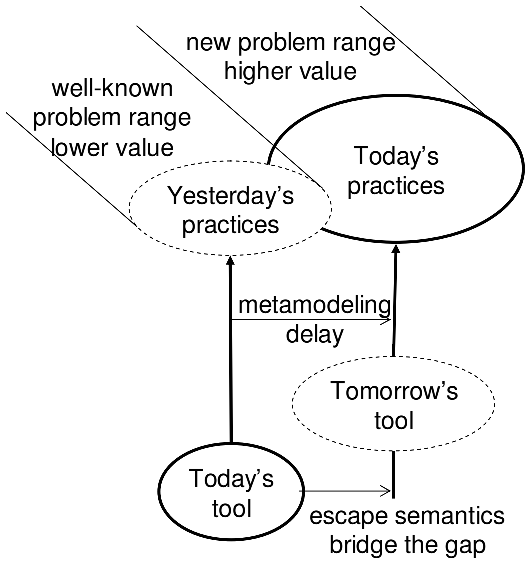

Diagram 4: Reduce the gap between tool and practices

To address this issue we implemented *escape semantics* in our languages, with the purpose of improving the modeling tool's stickiness by making it applicable to new problems not taken in account at language-design-time. The *escape semantics* allow for free-form modeling within boundaries set by the tool-smith, letting the modeler augment the official notation as necessary, typically when devising designs for new market segments. This opens the door to a collegial form of custom language construction where the DSM tool-smith and the domain expert initiate the reflection and practitioners add their thoughts and knowledge from field applications.

We identified several *escape semantics* that can empower the tool user:
1. *Joker objects* to augment the official language with new concepts
2. *Joker links* to augment the official language with new kinds of relationships
3. *Overwritable list-boxes* that can be augmented on the fly with new entries
4. *Code generator aspects* to let tool users augment the model compiler
5. The ability to extend model concepts with properties created on the fly

We noticed that young practitioners are more inclined to "invent" new notations to represent the world as they see it, while senior practitioners have been trained to the corporate notation and limit their usage of *escape semantics* to fixing purposes. Typical usage patterns of *escape semantics* we identified include:

- Add a concept that was overlooked by the tool-smith and expert.
- Augment the expressiveness of an existing language to enter a new domain.
- Adapt existing models to new corporate regulations.

Following is a real example of *escape semantics* occurrence. A Field-bus Definition language had been defined to declare the type, cardinality and mapping of data points found in communication protocols used to interface sensors and actuators. Because this language was too simple to describe Full2Way field-bus, Mr. Tanaka proposed the addition of a union relationship by using one *Joker object* (yellow box) and three *Joker links* (red dashed lines) to represent the fact that terminal unit data points (*tu*) and lighting dimmer data points (*dimmer*) are interchangeable.

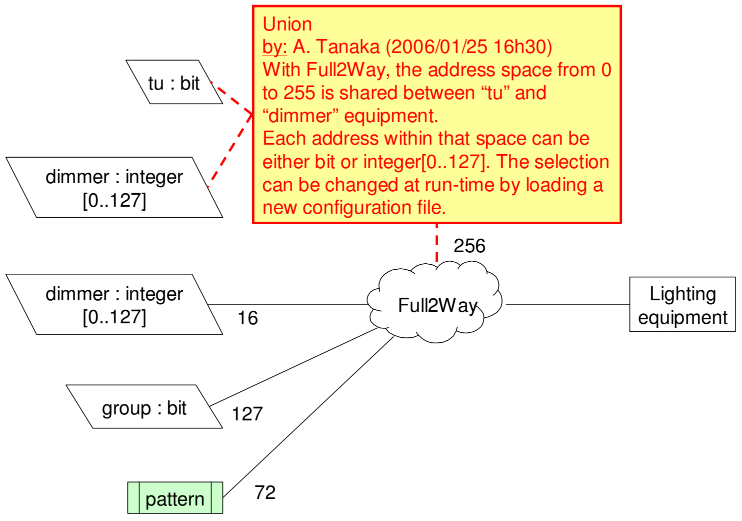
Union
by: A. Tanaka (2006/01/25 16h30)
With Full2Way, the address space from 0 to 255 is shared between "tu" and "dimmer" equipment.
Each address within that space can be either bit or integer[0..127]. The selection can be changed at run-time by loading a new configuration file.
tu : bit
dimmer : integer [0..127]
256
dimmer : integer [0..127]
16
Full2Way
Lighting equipment
group : bit
127
pattern
72

Diagram 5: Using *escape semantic* to convey the meaning of union which does not exist in the language yet

In addition to language adaptability via *escape semantics*, we find necessary to design the languages for modeling flow. That means reducing the number of double-clicks, text-field editions, list and menu navigations necessary to draw a complete model.

As a rule of thumb, all activities introduced by the DSM but not found in sketching should be minimized, because practitioners will compare modeling with tool to sketching models. Some form of automation can be introduced in modeling languages to protect the modeling flow:

- Default values (object name, property value) to separate the creative activity of drawing pictures from the activity of specifying attributes. This can be facilitated by following the principle of convention over configuration [3] in the language design.
- Special values *undefined* and *unknown* to model fuzzy problems were some specifications remain unclear.

- Integration of the DSM tool with the corporate IT system to avoid duplicate input of information

Sustaining Deployment of Many Custom Languages

By introducing his tools into the product development process, the DSM tool-smith is exposed to several risks, including but not limited to:

- A broken visual editor does not load old models
- The visual editor does not support current modeling practices
- A broken code generator produces malfunctioning software
- A valid code generator has not been updated to support changes in the target framework

These risks are worsened by several factors specific to DSM:

- Most domain-specific languages (DSL) are proprietary and maintained by a limited team
- Proprietary DSLs suffer from limited scrutiny and peer-reviews
- Proprietary DSLs have a limited user base and are applied to a limited number of applications when compared to main-stream languages like UML, "C"

To address these issues we implemented some test-driven practices from the agile software development community.

For example, the opposite diagram illustrates our solution to test the correctness of `(modeling tool, framework)` pairs by generating executables from well-known models and by running these against well-known data sets. Doing so, the tool-smith can periodically verify all well-known model compilation cases after each modification of existing DSLs, reducing the risk of releasing broken model compilers to the user.

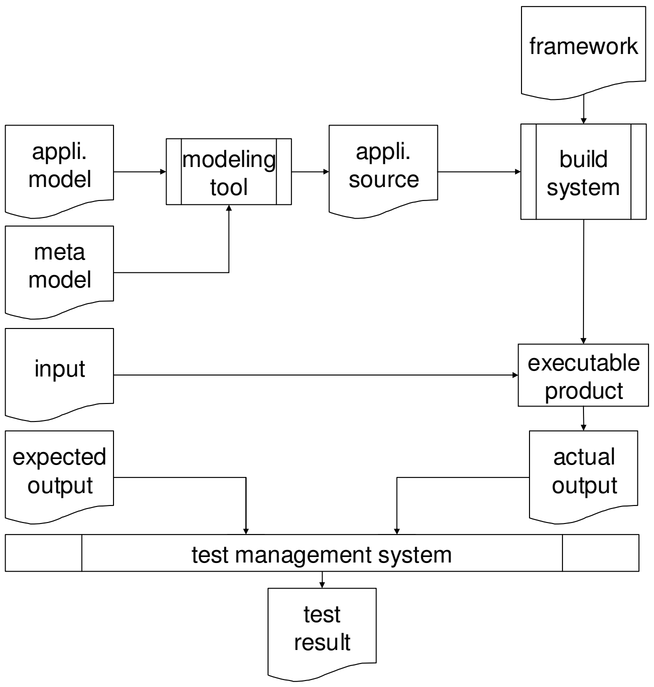

Diagram 6: Test-driven language development

Another step consists in checking the model repository for occurrences of *escape semantics* by way of *daily model analysis*. For example, the tool-smith could be emailed an alert on his mobile phone whenever a user would have used *escape semantics*, due to some limitation in the modeling language, or to lack of knowledge from the practitioner, which either is bad news. This mechanism could prove to be a powerful communication means between the tool-smith and his users.

Finally, we use Scrum [4] to manage the development of visual language editors and code generators. The product backlog proved to be a very practical tool to negotiate work items between the tool-smith and the stakeholders. For that purpose, we slightly customized the backlog format by adding columns *Example models* and *Generation samples*. Backlog items with more *Example models* and *Generation samples* are given priority because the more variation samples, the better DSL we can devise. The message is well understood by stakeholders who naturally do their homework to find or create more samples to get their problem higher in the list. Holding monthly Sprint Reviews open to all stakeholders and interested persons is also an efficient way to demonstrate progress, to keep stakeholders and users interested and involved, and to expose other practitioners to the DSM, fostering inquiries and requests for help.

The DSM Tool-smith's Commandments

We propose to summarize this paper in the form of seven principles for the DSM tool-smith:

- *You shall find the measurable pain of each user.*
- *You shall promote DSM as the medicine for each user's pain.*
- *To product and solution developers, you shall give DSM. To technology developers, you shall offer well-known software engineering practices. To all you shall give Agility.*
- *You shall keep your tool up-to-date with your user's changing practices.*
- *You shall offer escape semantics to your users.*
- *You shall design your languages for ease of modeling.*
- *You shall daily-test your languages and code generators.*

Conclusion

We described key aspects of MEW's approach to deploy domain-specific modeling (DSM) in the developments of systems of embedded devices, and we proposed practices to support the DSM tool-smith. We found that most challenges are not technical but instead human and organizational, and we interpret this as a testimonial of the maturity of DSM tools, but also as recognition of the lack of associated methods and practices.

Usability of DSM tools remains the most challenging issue, because these are typically developed internally by a limited pool of software engineering specialists who lack expertise in ergonomics.

To address this problem we are exploring the discipline of human-computer interaction (HCI), and we found in Cognitive Dimensions (CD) [5] a promising candidate as some cognitive dimensions map precisely to several topics we discussed in this paper. For example, *premature commitment* and *viscosity* relate to our effort for preserving *modeling flow*, when *secondary notation* relates to our *escape semantics*. And *progressive evaluation* could correspond to the ability of simulating models with *undefined* and *unknown* values.

Further research will tell whether Cognitive Dimensions can help build DSM tools that are not only efficient in solving technical problems, but also comfortable to work with.

References

[1] Juha-Pekka Tolvanen, Steven Kelly. (2006). "Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences" (SPLC05).

[2] Seth Godin. (2006). "What makes an idea viral?" http://sethgodin.typepad.com/seths_blog/2005/09/what_makes_an_i.html

[3] Dave Thomas, David Heinemeier Hansson. (2005). "Agile Web Development with Rails", Pragmatic Bookshelf.

[4] Ken Schwaber, Mike Beedle. (2001). "Agile Software Development with SCRUM".

[5] Alan F. Blackwell, Thomas R.G. Green. (2006). "Cognitive Dimensions of Notations: A Tutorial" (VL/HCC06).