Domain-Specific Modeling: The Killer App for Method Engineering?

Steven Kelly
MetaCase
stevek@metacase.com
http://www.metacase.com

Abstract. The method creation heyday of the 1980s was characterized by convivial chaos, leading to the idea of a discipline of method engineering. Before it could grow, the unification and marketing machine of UML crushed method development into "one size fits all" design by committee in the 1990s. A scattering of brave souls went against the current, creating modeling languages specific to their own problem domain, and were rewarded with significantly higher productivity. As they seek to scale their solutions, they need help from the research world to analyze their results, and to bring to bear the learning from the early days of method engineering.

1 Introduction

The 1980s were in many ways the heyday for the creation of new modeling languages and associated processes, together known then as methods. It seemed everybody with software development experience and a theoretical bent was creating their own method. Many of the methods shared common features, whether by shared ancestry, loaning, or convergent evolution. These factors gave rise to the idea of a discipline of method engineering, to improve the process of creating a new method.

The 1990s poured cold water on the fledgling discipline, as UML progressively out-marketed or subsumed most common methods. Rather than a chaotic yet convivial mass of method creators, method development largely became something only a single committee were doing. Some brave souls carried on under the radar, but always with the stigma of working on a "non-standard" modeling language. Their reward was often significantly higher productivity, as their modeling language was made to fit tightly with just their problem domain, and production quality code could often be generated directly from the models.

Please use the following format when citing this chapter:

Kelly, S., 2007, in IFIP International Federation for Information Processing, Volume 244, Situational Method Engineering: Fundamentals and Experiences, eds. Ralyté, J., Brinkkemper, S., Henderson-Sellers B., (Boston Springer), pp. 1-5.

With the turn of the millennium, there were already some published success stories of Domain-Specific Modeling [1]. Metamodeling tools offered method engineers automatic tool support of any modeling language they chose to specify, significantly reducing the cost of building and supporting a new method. The method engineering research community began to emerge from hibernation, gaining new members and a new direction. Rather than assembling methods from smaller common building blocks, the focus was now on creating entirely new modeling languages, and the skills and tools necessary for that task.

Now, the giants of Microsoft and Eclipse have joined the fray, bringing the ideas of Domain-Specific Modeling to the masses. More mature tools go beyond the giants' entry-level offerings, providing support for true method engineering: new parts are created whenever necessary, but existing fragments can be reused, and several modeling languages are integrated into an effective whole. To push the field further requires four things:

1. understanding the state of the art from mature tools and previous research,
2. empirical research on the use of DSM in the field,
3. condensing that information and experience into advice for method engineers,
4. hypothesizing from it a set of new requirements for the next generation of tools.

As guardians of point 1 and one of the few trustworthy parties to carry out point 2, the method engineering community is thus in a unique position of responsibility. Those unaware of past results or current practice will leap to points 3 and 4 to form unfounded conclusions, sending research and practice back to the dark ages. Many commercial attempts to jump on the bandwagon are doing just that, with productivity of "UML-based MDA" only increasing by 35% [2], and of "Software Factories" by 20% [3]. Even allowing for the vagaries of these particular cases, one by a major MDA tool vendor and one by the most referenced users of Software Factories, these meager productivity increases compare poorly with the 500%–1000% commonly encountered with better-founded DSM approaches [1].

2 Meta- and meta-meta-development, but no development?

In order to achieve our goals, we must first know and agree what those goals are. In our Working Group, the focus is on the planning, analysis, design and evaluation of information systems. Conspicuously absent from the list is the construction of information systems: their implementation in a programming language. This omission is all the odder when one looks at other working groups in TC8, and sees that none of them include a focus on construction in general; only for specific kinds of systems such as smart cards. How can this be? Construction is clearly central, the one phase without which there is no hope whatsoever for a project, and without which there would be no information systems, good or bad, for us to study. Construction is also more clear-cut than many phases, amenable to hard scientific analysis: an IF statement is unequivocal, whereas a UML Association can mean different things to different people.

Perhaps the answer lies in that very clarity. In other phases we can believe that we have a generic solution: there is enough fuzziness that a concept of "Requirement" or "Entity" can be seen as applicable to almost any project. In the unforgiving glare of compilers, let alone users, a particular part of a program can clearly be found lacking. Bubble sort may be fine for a PC home address book, but too sluggish in other domains such as mobile phones or enterprise databases. Academic research strives to come up with results that are universally true, but IS construction is clearly dependent on the domain.

Whilst only implicit by omission for construction, this situational contingency has always been recognized by WG 8.1 for IS development methods: different modeling languages are suited for different problem domains. Historically, such domains have been considered broadly, e.g. vertical domains such as banking, or horizontal domains such as database systems. Modeling languages have taken a similarly broad outlook, e.g. ER diagrams for database design. This breadth was taken to its extreme in UML, which claims to be a universal method for any discrete software system. Unsurprisingly, in reaching out for this goal UML has grown to include a great number of modeling concepts, each of whose semantics are deliberately vague.

3 Broad or Narrow?

While UML claims to be able to model everything from space shuttles through database applications to mobile phones, no one development group needs this breadth. Rather, each group works in a far narrower domain: not just mobile phones, say, but user-visible applications for the Nokia Series 60 range of phones. As this is one of many thousands of other such domains in IS, a modeling language developed specifically for this situation can thus be a much better fit than UML. Such a Domain-Specific Modeling language could have fewer modeling techniques, and their concepts would be much more precise. Above all, the concepts can be at a significantly higher level of abstraction: rather than having a general "Event" causing transitions in a State Diagram, their can be different concepts for events like "Button Press", "Soft key press", or "Incoming call".

When comparing other related domains, e.g. Ericsson mobile phones, it quickly becomes apparent that while there may be a number of similarities between the DSM languages, there is no way to produce an integrated modeling language without losing precision or introducing significant bloat. However, the differences between the modeling languages pale into insignificance when compared to the difference between the code written in the two related domains. Partly because of different underlying components, frameworks and platforms, but above all because of separate evolution of in-house standards and traditions, the code in the two domains is virtually unrecognizable. What remains common, however, is that for either domain it is a relatively simple task for an expert developer to specify the mapping from their DSM language to their code.

Generating full production quality code directly from high-level models has long been a goal of the software industry. Earlier attempts have largely failed, except in a few narrow domains. Unfortunately, while narrowness has been a virtue for good code generation, it is normally bad news for business. Another major factor in the failure of earlier code generation attempts has been the difference between the code different groups expect or want, as seen above. Since an expert from the group is now creating a domain-specific code generator, that problem is largely overcome, as are issues of vendor lock-in and long change cycles. Narrowing down the problem domain space that the modeling language and generator need cover also addresses another problem: earlier generated code tended to be bloated and inefficient, having to cope with so many possible situations.

4 Back to the Future?

Any right-thinking UMLer will of course respond with the standard argument: standards (pun unavoidable). But will the sky really fall if we use more than one modeling language? DSM has consistently shown productivity increases greater than any since the move from assembly language to 3rd Generation Languages. These figures stand up to empirical experiment [e.g. 4] and industrial experience of several hundred developers working on hundreds of products over a dozen years [5].

Looking back at assembly languages, we can note an interesting fact: there was a different language for each family of chips. Indeed, there could be more than one language for a family: assembler vendors added their own higher-level constructs, which did not map one-to-one with a single machine code instruction on that chip. Was the move from assembly language to 3GLs therefore a welcome escape from a confusing plethora of languages to a single universal language, e.g. C? Emphatically not, although many students and practitioners today seem to have assumed this. Rather, there was a surprisingly broad range of languages available, and many focused on a specific domain, e.g. scientific programming, business systems or graphics.

The move was also a gradual one, with the majority of systems at the start of the 1980s still being built in assembler. Some groups made the move earlier than others, but all made it for the same reason: the productivity in the 3GL was higher than in assembler. The main reasons for this were that one statement in a 3GL corresponds to several in assembly language, and that 3GL statements are closer to the way we look at the world, rather than the way a chip interacts with binary data. Both these reasons are also factors in why DSM is more productive than programming in a 3GL.

The rate of evolution of hardware has always been higher than that of software, possibly constituting another factor: users of a 3GL did not have to rewrite their programs each time a new chip was released. Instead, one compiler writer rewrote the mapping for the new chip, and all users of that 3GL simply upgraded to the new compiler version. Again, with DSM the situation is similar: to cope with a change in platform version, or even platform or programming language, there is no need to build all models from scratch, or even to edit them at all. Instead, the expert developer updates the generator, and all models by all developers now produce code for the new platform.

5 Conclusion

In the 1980s, the focus of method engineering was on helping method users to select from among the many available modeling languages. With the advent of UML, the choice has been so restricted that the focus has shifted to other areas such as requirements or processes. With Domain-Specific Modeling, there is a call for a new category of developer, creating a modeling language for their group. Whether that group is a single project, a company, or even similar projects across many companies, the task of building a new modeling language is a challenge.

Method engineering research of the 1990s led to some of the meta-metamodels and metamodeling tools that are easing the adoption of DSM, enabling today's method engineers to concentrate on building a good language, rather than getting stuck down in reinventing the wheel of modeling tool construction. The basic task of identifying suitable constructs has been analyzed from dozens of industrial cases, giving useful initial guidance to the method engineer [6].

To move forward, method engineers will need advice on modularization and continuous integration of models by separate developers, research results on the changes needed in version control systems for models, new algorithms for identifying and displaying differences between similar models, and above all empirical studies on which modeling approaches work, which do not, and where these results hold. Those building tools for method engineers will need theory and in-depth analysis: are the existing meta-metamodels really different, or are people twisting UML and MOF semantics to get closer to some actual set of useful constructs.

To be relevant, researchers in method engineering must be at least as smart as the method engineers. Having had the pleasure of working with many of today's method engineers, I can promise you this is a tough challenge. In my quarter century in this field, I have only met one group that could make that claim: the method engineering researchers of the early 1990s, now 15 years older. It is high time for this conference, and for a new group of leading lights. I have every hope for the class of 2007!

References

1. DSM Case Studies and Examples, 26.5.2007; http://www.dsmforum.org/cases.html
2. M. Burber and D. Herst, Productivity Analysis — Model-Driven, Pattern-based development with OptimalJ, 26.5.2007 http://www.theserverside.com/tt/articles/article.tss?l=SymposiumCoverage
3. J. Warmer, Case Study: Building a Flexible Software Factory using Small DSLs and Small Models, (unpublished discussion from talk), Code Generation 2007
4. R.B. Kieburtz, L. McKinney, J.M. Bell, J. Hook, A. Kotov, J. Lewis, D.P. Oliva, T. Sheard, I. Smith and L. Walton, A software engineering experiment in software component generation, *18th International Conference on Software Engineering (ICSE'96)*, 1996
5. Nokia case study, MetaCase, 1999; www.metacase.com/papers/MetaEdit_in_Nokia.pdf
6. J. Luoma, S. Kelly and J-P. Tolvanen, Defining Domain-Specific Modeling Languages: Collected Experiences, 4th OOPSLA Workshop on DSM, TR-33, University of Jyväskylä, 2004