Domain-Specific Modeling for Full Code Generation

THE ADVANTAGE OF USING EXTERNAL LANGUAGES IS THAT THEY CAN BETTER GUARANTEE THAT NORMAL DEVELOPERS FOLLOW THE DOMAIN-SPECIFIC CONSTRUCTS AND RULES.

by Dr. Juha-Pekka Tolvanen

Domain-Specific Languages are becoming hard to avoid. There is a natural reason: they are more expressive and therefore tackle complexity better, making software development easier and more convenient. Most importantly, they raise the level of abstraction and, together with domain-specific generators, can automate the creation of production quality code. In this article we introduce Domain-Specific Languages (DSLs), and Domain-Specific Modeling (DSM) in particular, along with industry experiences. We give guidelines for moving successfully from coding to modeling with full code generation.

Today there are two schools of Domain-Specific Languages: those preferring to embed the new language constructs into an existing host language, and those who prefer to keep them in a separate external language. While both have their place, the advantage of using external languages is that they can better guarantee that normal developers follow the domain-specific constructs and rules. This resembles past developments in textual programming languages; inline assembler in C did not become widespread since most people wanted to keep these abstractions separate.

The other division is in representation. Should specifications be expressed in text, diagrams, matrices, tables, or some other form? The right answer obviously depends on the application domain. We should use the representation that feels most natural to solve the problem. For many cases, I favor graphical models because they have several advantages over textual specifications, which are inherently linear and often written to satisfy the compiler rather than support problem solving. Numerous scientific studies have shown that graphical models are easier to read and understand since they can express things like conditions, parallelism, and structures better than text. Graphical models are also especially good for humans since we are good at spotting visual patterns. Remember the well-known saying that a picture is worth a thousand words. Let's next look at two examples of DSM.

Examples from Practice

The examples are selected from practice and from different domains. Figure 1 shows a case from the automotive industry,

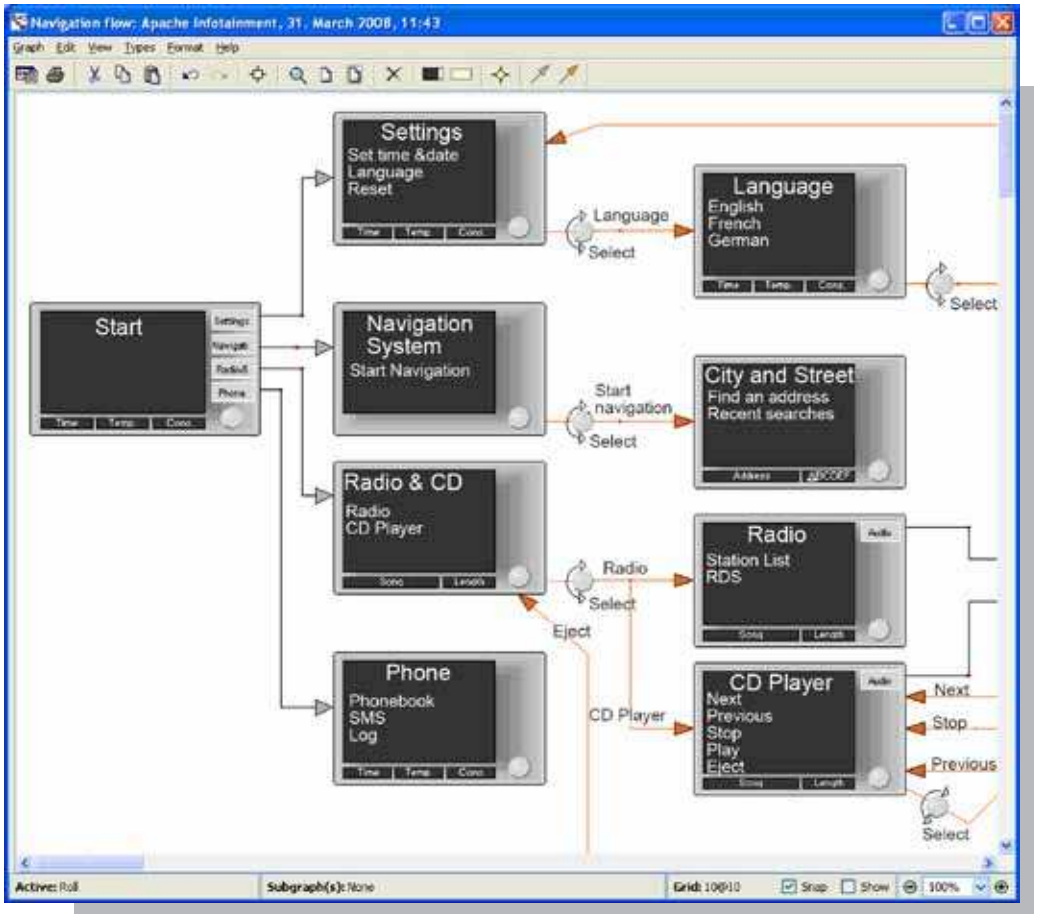

Figure 1: Designing automotive infotainment system with a DSM language

the menu design for an infotainment system. The language used for this model covers the user interaction, navigation and logical structure. Other languages are integrated with this one to cover the detailed display layout and graphics settings for fonts, icons, colors, etc. From these models the full code for controlling the infotainment system can be generated. Here the language is clearly operating at a higher level, above the code, since the developer does not need to know which target code is generated. Several generators can be created, building the same system but for different target platforms. Each generator can take advantage of the features of that particular target platform to best implement the modeled system.

The other example is from a domain where there is no User Interface, embedded device control logic. The model in Figure 2 describes how phone calls are handled by a telephony server. An unanswered call from a certain location is routed to a second number, email or voice mail. Similar telephony services are created with this language by telco service engineers, e.g. for companies or emergency services. The code generator creates the full service specification in the XML format required by the telephony server. The service engineer, however, can focus on call processing and not on the low-level details of the XML. The modeling language's rules for connections between objects prevent the creation of unimplementable or ambiguous specifications.

Industry Experiences

Raising the level of abstraction with DSM as illustrated above always improves productivity. The exact improvement is largely set when the language and generator are defined for a particular case. Panasonic ran a comparison with traditional manual practices by implementing the same features by manually coding them and by using a DSM solution. The productivity with DSM was about 500% higher than manual coding – across the whole lifecycle from design to running production code. Polar Inc. conducted a study in which six engineers implemented features for a heart rate monitoring device using DSM. The engineers estimated the productivity improvement to be 750% compared to manual practices. At Nokia, mobile phone developers experienced a 1000% productivity increase, with development time for a standard feature cut from two weeks to one day. At Lucent, several DSLs were developed and deployed, giving productivity increases between 300% and 1000% depending on the case. Figure 3 shows further examples of productivity increases in various domains.

Productivity is not the only benefit. For safety critical applications, quality matters more. US Air Force studies on military systems have shown that a dedicated specification language and generator resulted in 50% fewer errors than manual coding using components. Both the language and the generator contribute to the improved quality. The language prevents modelers from creating specifications that are illegal, lead to errors, or even cause poor performance. Preventing errors at this early stage is, as always, significantly cheaper than

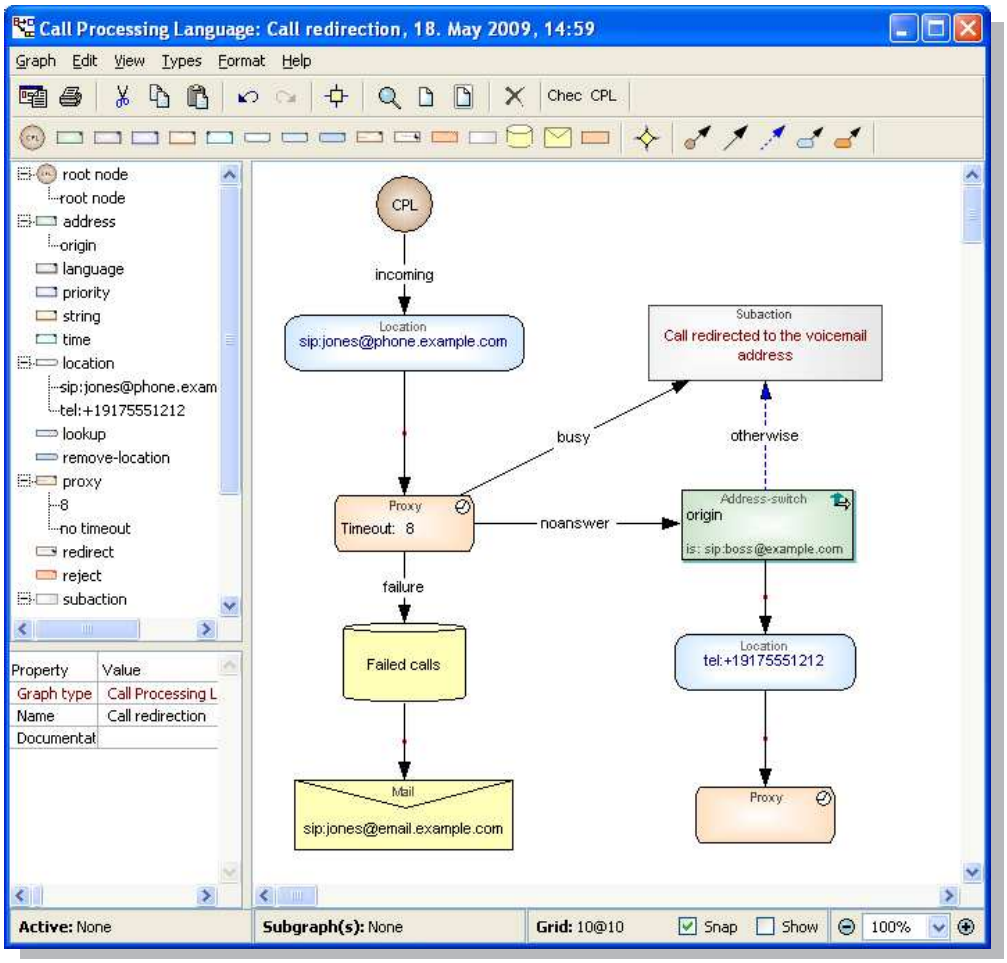

Figure 2: Example design using Call Processing Language

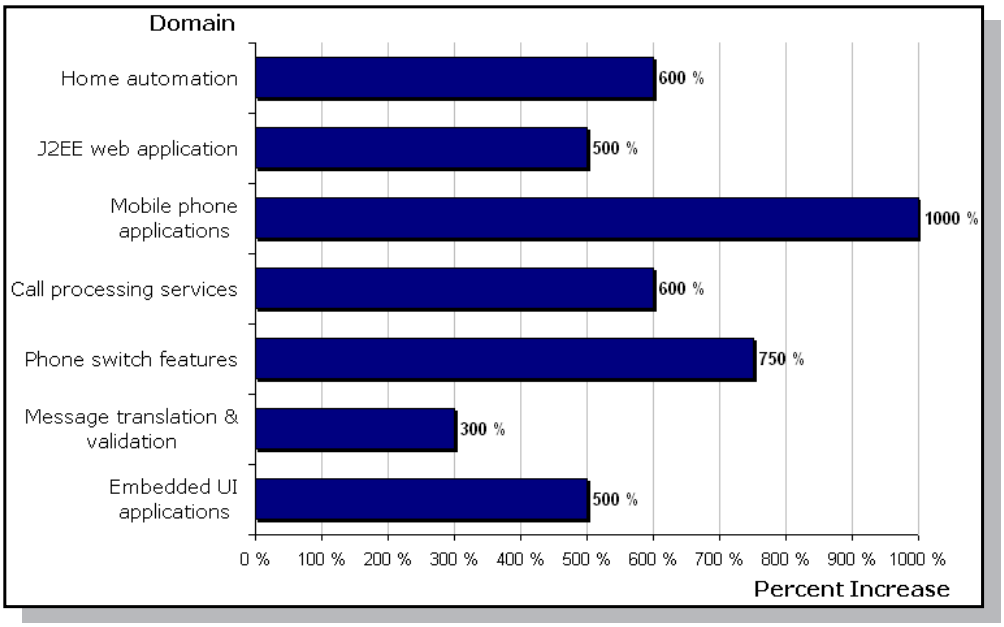

Figure 3: Measured productivity improvements in various domains

catching them later. Automotive software developer Denso found that implementation and testing requirements are both reduced by having the domain rules (AUTOSAR in their case) in the language. With code generators many typical errors — missing references, typos, forgotten initializations, referencing freed memory — simply don't occur anymore. For instance, EADS Secure Networks has detected that there are significantly more errors, and a wider range of errors, with manual coding than with DSM and code generation.

Best practices for moving from coding to modeling and code generation

These improvements in productivity and quality are possible because experienced developers, knowing the domain and how to write good code for it, have defined the DSM solution. It is therefore worth emphasizing that only one or two developers need to master language and code generator development. The other developers simply use the modeling languages these experts have created. This division of labor enables the best developers to package their experience, empowering other members of the development team.

The process of creating a DSM language and generators for a given domain often starts with a feasibility study, which creates a partial implementation over a few days. After this proof-of-concept phase, a pilot project will be launched. This phase usually lasts a few weeks. The language is fleshed out, and then incrementally tested and improved by building a real system with it. Concrete outcomes of the pilot include the full version of the modeling language and its tool support, plus code and documentation generators. Figure 4 illustrates the resource usage for developing modeling languages and generators in some industry examples from our customers; results with other tools vary.

Defining a DSM language is a new experience for most people, and they may thus see it as an obstacle. In reality, all developers are already using the concepts and terms that are specific for the domain and company they are working for. These are the terms found in requirements documents and initial whiteboard sketches, that developers have had to learn how to translate into UML or code. DSLs thus do not contain newly invented concepts, but rather offer the possibility to build systems directly using terms that are already known,

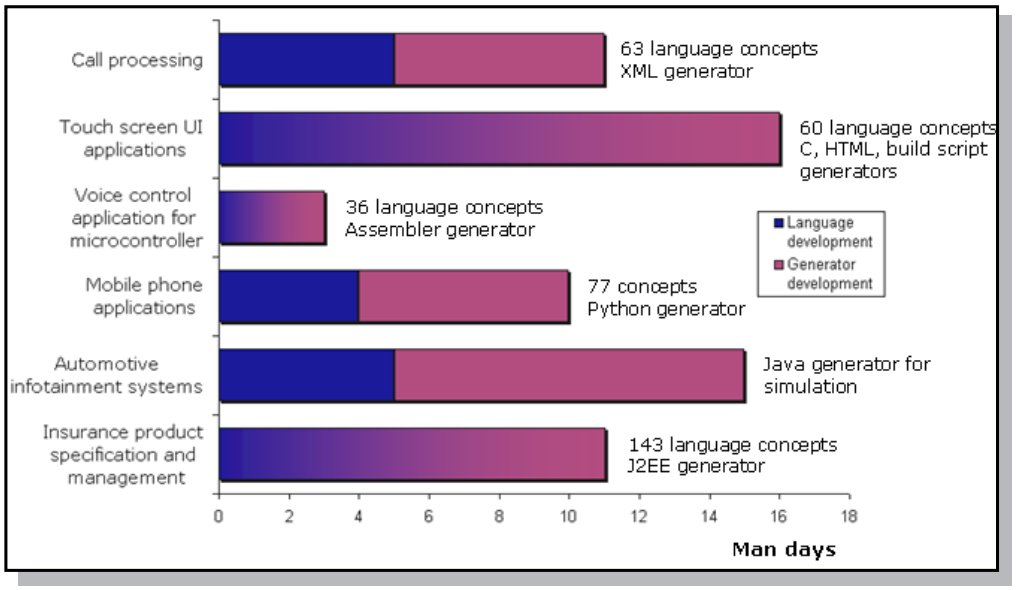

Figure 4: Use of resources in DSM creation

tried and tested in that domain.

The best approach for building DSLs is incremental. Build a little of the language, model a little, make some changes to the language, model some more, etc. This iterative process minimizes risks, allows early testing, and smoothes the path of organizational change: moving from coding to modeling. Iteration does not stop with initial language deployment. Once in use, the language will be evolved and extended later as the domain evolves and gaps are filled. Good tools can greatly reduce the time and effort required to build languages, keeping the language definition process agile by automatically updating earlier models as the language changes.

The other recipe for success is focusing only on a narrow area of interest — as in the cases above. The narrower the domain, the easier it becomes to build a good, high-level language and make generators produce first class code. This is the reason that most successful DSM solutions are defined inside a single company: you only have to solve your own problems, not the whole world's. Keeping language development in house gives full control of the automation to the company creating the language, rather than to the tool vendor or other forces outside the company.

The bottom line for DSM languages is how much they raise the level of abstraction up from coding to the problem domain. Languages that apply programming concepts as modeling constructs often end up visualizing source code, and thus fail to provide any real improvements in abstraction — or productivity. Basing the language on problem domain concepts raises the level of abstraction more reliably, as the cases above illustrate.

Concluding remarks

The consistency with which DSM has improved productivity and quality has led companies to increasingly invest in creating DSLs. These languages will be production quality, but not productized. Their benefit is in their narrowness, unlike general purpose modeling tools that must appeal to the widest possible market. The task of language creation will thus become increasingly common, as did library and framework creation before it.

Increasing system complexity and the move from hardware to software in new domains also drive our industry towards languages on a higher level of abstraction. Rising popularity comes from improved tooling. DSM pioneers had to create the modeling tools for their languages from scratch, making DSM practical only for large projects. Today's mature language workbenches make language creation cost effective, providing quality tools for modelers with the minimum effort.

References

EADS Case Study, MetaCase, 2007; www.metacase.com/papers/MetaEdit_in_EADS.pdf.

Kelly, S., Tolvanen, J-P., Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Society Press, 2008

Nokia Mobile Phones Case Study, MetaCase, 2007; www.metacase.com/papers/MetaEdit_in_Nokia.pdf.

Safa, L., The Making of User-Interface Designer, A Proprietary DSM Tool, Proc. 7th OOPSLA Workshop Domain-Specific Modeling, Montreal, 2007; www.dsmforum.org/events/DSM07/papers/safa.pdf.

About the Author

Dr. Juha-Pekka Tolvanen is CEO at MetaCase and has over 15 years' experience on the creation of modeling languages and code generators. He has recently co-authored a book on Domain-Specific Modeling (Wiley, 2008).

Author Contact Information

Email: Dr. Juha-Pekka Tolvanen: jpt@metacase.com