Moving from Writing Code to Generating It

Generate full production code from precise, high-level models in your own modeling language

Steven Kelly

Domain-Specific Modeling Languages

I always remember the day when my bluff was called. I had just been handed 50 pages of Java code and was told, "Generate that!" I had always said that if you gave me a sufficiently precise model of a system, and the code that implements that system, I could build a code generator that would produce that system from the model.

While I normally prefer to work on code in a text editor, this task clearly called for more scalable weapons from the architect's tool chest: hard copy, scissors, sticky tape, and the full palette of highlighter pens. Shutting the door, I set to work at analyzing the code to find repeated sections, common blocks where just a few elements changed, and larger scale patterns. These then had to be crosschecked with the graphical model of the system, to find which parts of the model determined them. Several hours, and more than a few shouts of, "But that's just the same as *that!*" later, I figured I had it nailed. In a couple of hours more, I had implemented a generator that crawled the structures in the model to spit out the corresponding blocks of code, replacing the changing elements with values from the models.

Running the generator the first time produced…exactly nothing. After fixing the offending minor bug, I got to work on its bigger brothers: comparing the generated output with the code I had been given. After an hour or two, I had it down to about five or six spots where I just could not figure out why the code was what it was. Clearly, sometimes a given situation would require one kind of code, but in one or two spots, that code was different or not present. I sent the results off to the customer, asking for clarification of what improvements were needed to the modeling language to capture the differences between the situations.

The next day was scheduled for implementing those changes to the modeling language, and updating the modeling language and generator. When I got the customer's reply the next morning, I realized that for once, I would go home well rested. The generator was already finished: every single one of the differences was a mistaken in the hand-written Java code.

If You Have The Code, Why Generate It?

The background to this story was a German company that wanted to build an eCommerce site where their customers could compare insurance policies. The server would understand the policies sufficiently to be able to calculate things like costs and benefits. Their problem was that this effectively required "implementing" the policies in Java, so the web site could "run" them for comparison. The Java to do this was non-trivial, and the policies themselves were written in dense legalese that only the insurance experts could understand. And the insurance experts didn't happen to be Java programmers.

Since the final system would have hundreds of such policies, each with a similar mass of code to the story above, something had to be done. In effect, there was too large a disconnect between the level of abstraction of the problem, and the level of abstraction offered by Java. If this were something that only had to be faced once, in a small section of the application, it might have been acceptable. But here, there clearly had to be a better way: something at a higher level of abstraction than Java, closer to the language used by the domain experts, and yet with the precision to enable us to generate code.

This is in fact a problem faced by any company building a range or family of applications in the same domain. How can we raise the level of abstraction beyond today's third generation programming languages? One way is to move away from trying to specify all kinds of applications using only one generic set of program language concepts. Instead, each company could use the concepts of the problem domain it works in, giving each concept its own visual representation.

Symbols in such a domain-specific modeling language (DSM language), along with the rules about how they can be connected and used, would thus come directly from the world in which the application is to run. This is a whole level of abstraction higher than UML models. It results in a very expressive, yet bounded, design language that can only specify applications in the problem domain that it was designed to cover. While it may be useless elsewhere, in its own domain it is both concise and precise.

Together with the insurance experts, we thus built a domain-specific modeling language for describing insurance policies. It had symbols for concepts like "risk," "sub-policy," "insured," "accident," and "payout calculation," and relationships that connected these to show how the policy was made up, and who paid what under which conditions. An example from a top-level diagram is seen in Figure 1.

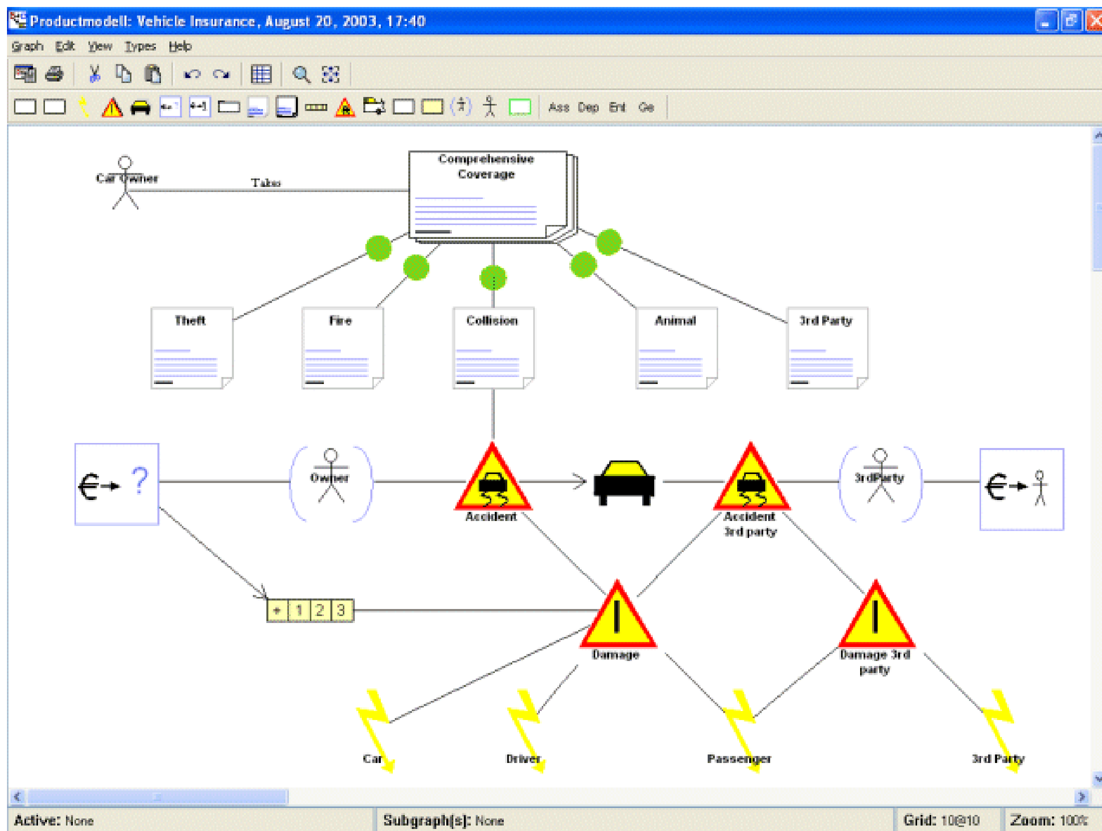

Figure 1—An example diagram in the domain-specific modeling language for insurance

Based on one policy’s hand-written code from their Java programmers, we built the code generator that turned this modeled information about the static structure and dynamic behavior of the product into J2EE Java code. Now, the insurance experts could easily describe a policy in the new modeling language, and the appropriate code could be generated instantly. New policies could be added to the system up to five times faster, and with fewer errors than with manual coding.

When, And Where Is DSM Applicable?

Domain-Specific Modeling requires a DSM language and a matching code generator. Although defining these is much easier than defining a generic modeling language, it is still not a task for every developer. It requires a good knowledge of the problem domain and the code that is written for it. This domain expertise is usually found in situations where we deal with product family development, continuous development of the same system or configuration of a system to customer-specific requirements (phone routing systems, CRM, Workflow, payment systems etc.).

Experienced developers who work in these types of domains know what the concepts of their domain are, are familiar with the rules that constrain them and know how code or configuration files should be written best. Sometimes it is one expert who possesses this knowledge, sometimes it is shared among a small group of people with different skills, as in the above example. Whether one person or several, the expertise they possess makes them well-qualified to define the automation mechanism that will make the rest of the developers much more productive.

Benefits of DSM

DSM requires an investment of development resources and time to set up the DSM environment. Although this may clash with the urgency of current development projects, the improved productivity is usually worth it. Industrial experiences of DSM consistently report productivity being between five and ten times higher than with current development approaches. Reported examples include Nokia's cellular phones, Lucent's 5ESS telecommunications switch systems, Microsoft's Whitehorse tools for SOA, and EADS Tetra terminals.

The last time our industry saw such productivity gains was with the move from assembly language to Third Generation Languages such as FORTRAN or BASIC: that produced a leap in productivity of 450%. The 40 years since then have had little effect: from BASIC to Java is an improvement of just 20% (Software Productivity Research, 2005). This is not so surprising if you consider that all 3GLs are at roughly the same level of abstraction.

Traditional modeling languages like UML have not increased productivity, since the core models from which code can be generated are on the same level of abstraction as the programming languages supported. Current MDA tools are based on UML, and thus suffer from its problems. Even a tool vendor sponsored study only found an increase in productivity of 35% compared with hand coding (TheServerSide, 2003).

In addition to productivity gains, it is surely better that the expert formalizes development practices once, and other developers' generated code automatically follows them, rather than having all developers try to manually follow best practices all the time. All architects will be familiar with how well their wonderful guidelines and standards documents are actually read and followed by developers! If the rules and guidelines are embedded in a tool, it can enforce the rules and guide or warn developers appropriately.

DSM models capture the information about the actual system and its behavior, rather than its implementation, with a certain programming language and set of libraries. This means that the exact same models can be used to generate code for a different programming language or libraries: all that is needed is that one developer writes the generator. Flexibility like this significantly reduces the risk of obsolescence and the costs of migration to different technology. On a smaller scale, code changes required by newer versions of the same technology can also be made by one person in the generator, rather than by every developer in every piece of code.

Tool Support for DSM

A software development method is of little value if it has no tool support. Designs in a DSM language on a whiteboard do not generate any code, and thus while they help in defining the functionality of the application or system, they do not improve productivity. DSM has been around for quite some time but companies that chose to adopt it often had to resort to building their own environments to support their DSM language and generators. The huge investment in man years required to do this often led to the decision to look for other ways to improve productivity.

While a DSM language and corresponding code represent the core competency of the company that makes them, in most cases generic modeling tool functionality is not something the company has experience in. Also, the basic behavior of a modeling tool generally remains the same irrespective of the specific modeling language. It is thus possible for a company with modeling tool experience to make a generic modeling tool, which can be customized appropriately by each organization building a DSM language.

In early examples of this kind, the customization was more a case of writing your own code that ran on top of the vendor's framework. Today's more advanced tools make it possible to build your own modeling language without writing any code: the modeling language definition is input as data through graphical models and forms, and the generic modeling tool configures itself automatically from that data. This has the significant benefit that changes to the modeling language can be instantly reflected in the modeling tool—even updating existing models.

Most existing DSM takes place with DSM environments, either commercial such as MetaCase's MetaEdit+, or academic such as Vanderbilt University's GME. The increasing popularity of DSM has led to DSM frameworks being added to existing IDEs, for instance Eclipse's EMF and GMF, or Microsoft's DSL Tools for Software Factories (Greenfield and Short, 2004). A list of tools is available in the industry forum (DSM Forum, 2007) at http://www.dsmforum.org/tools.html.

Why Is DSM Different?

As few domains are static, the ability of DSM environments to let you evolve the modeling language is vital. This ability is something that separates DSM from earlier attempts to generate full code directly from models. In the CASE tools of the 1980s, both modeling language and code generator were provided by the vendor. If the modeling language did not fit your way of describing applications, you had to change; if the code was not good for your needs, you were stuck. In UML, the modeling language is fixed by committee and the generator by vendors. In DSM, the modeling language is designed by your best domain expert, the generator is crafted by your best developer—and if either of them needs changing or fixing, a good DSM environment will let you do it straight away, updating existing models.

Summary

Domain-Specific Modeling allows an organization to make major improvements in productivity where developers would normally be forced to resort to copy-paste reuse of code snippets. A top developer in the company creates a new graphical modeling language whose concepts map to the company's problem domain, and a generator that maps these concepts into the code he would teach others to write. The other developers build models with the new language, and full code is automatically generated. Today's DSM environments make creating and using such languages and generators accessible to all sizes of project.

Critical Thinking Questions

- Where are your developers resorting to copy-paste reuse?
- How much better is the code from the top 20% of your developers than the bottom 40%?

- How much time is spent on problems stemming from poor communication between domain experts and developers?

Sources

[DSM Forum, 2007] DSM Forum. 2007. DSM Tools List. <www.dsmforum.org/tools.html>. Accessed 2007 Jan 10.

[Greenfield and Short, 2004] Greenfield J, Short K. 2004. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. New York: Wiley. 666 p.

[Software Productivity Research, 2005] Software Productivity Research. 2005. SPR Programming Languages Table™ (PLT2005). <http://www.spr.com>

[TheServerSide, 2003] TheServerSide. 2003. Productivity Analysis—Model-Driven, Pattern-based development with OptimalJ. <http://www.theserverside.com/tt/articles/article.tss?l=SymposiumCoverage>. Accessed 2007 Jan 10.

About the Author

Dr. Steven Kelly is the CTO of MetaCase and co-founder of the DSM Forum. He has over ten years of experience of building metamodeling and modeling environments and acting as a consultant on their use in Domain-Specific Modeling. He writes and speaks frequently in major industry venues and journals such as SD Best Practices, OOPSLA and Dr. Dobbs.

Glossary

DSM. Domain-Specific Modeling. Creating and using modeling languages targeted for a narrow problem domain, often to generate full code from the models.

MDA. Model Driven Architecture. The Object Management Group's approach to model-driven development, relying largely on UML.

Software Factories. Microsoft's approach to automating development, using DSM in a central role.

UML. Unified Modeling Language. The Object Management Group's set of general-purpose modeling languages for any problem domain, but focused on implementation with an object-oriented programming language.