developer.*
The Independent Magazine for Software Professionals

Improving Developer Productivity With Domain-Specific Modeling Languages

by Steven Kelly, PhD

According to Software Productivity Research, the average productivity in Java is only 20% better than in BASIC. C++ fares no better than Java [SPR, 2005]. In fact, with the exception of Smalltalk, not a single programming language in general use today can show a substantial increase in productivity over BASIC.

So after all of the language wars, objects, components, and frameworks, we are still scarcely more effective than 20 years ago. However, go back a couple of decades more and there is a radical change: a leap in productivity of 400% from Assembler to BASIC. Why was that leap so big, and how can we achieve it again today?

The 400% increase was because of a step up to the next level of abstraction. Each statement in C++, BASIC or Java corresponds to several statements in Assembler. Most importantly, these languages can be automatically translated into Assembler. In terms of productivity, this means you effectively get five lines of code for the price of one.

Did UML Increase Productivity?

Traditional modeling languages like UML have not increased productivity, since the core models are on the same level of abstraction as the programming languages supported: When designing in UML we still work with objects, their attributes and return values. One day of modeling with a UML modeling tool produces a day's worth of code.

However, after using the code generation functionality of these types of tools you still need to go in by hand and add or edit the majority of the code. You're thus trying to maintain the same information in two places, code and models, and that's always a recipe for trouble. Ever seen anybody hand edit Assembler and try to keep their C++ code in synch with it?

Of course, UML has its benefits: the visual presentation can be read much faster to get an overview. But one look at the ongoing development of UML speaks volumes. Half of the developer world finds that UML cannot represent what they need in their models, and so want to add something to it. The other half says UML is too complex, and want to reduce it down to its core elements. UML tries to be all things to all men, and thus cannot raise the level of abstraction above the lowest common denominator.

Raising Abstraction With Domain-Specific Modeling (DSM)

How then can we raise the level of abstraction beyond today's 3rd 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 products it makes, giving each 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 problem domain—the world in which the application is to run. This is a whole level of abstraction higher than UML. This results in a very expressive, yet bounded, design language that can only specify applications in the problem domain that it was designed to cover.

For instance, a DSM language for developing mobile phone applications could make use of concepts like “soft button”, “menu”, “send SMS” and “notification”, which are descriptive of the mobile phone domain but can hardly be used for developing web applications, ERP software, or a business intelligence application. This narrow focus makes defining a DSM language a lot easier than a more generic language like UML or Java. The improved expressiveness on the other hand makes it possible to define an application completely with a relatively small modeling effort.

Because of this completeness, a good DSM language is well suited to be the basis for automatic generation of full code. This leads to a situation where you can have graphical models that form the core of your software development effort, and automatically generate all necessary code and documentation from them. The models are then both the design documentation—a view of the product or system at a high abstraction level—as well as its implementation in code. Documentation and implementation remain synchronized throughout the whole lifecycle of the application, and software development becomes truly model-driven.

Full code generation from a DSM language requires a code generator. The mapping from model to code is defined in the generator. DSM languages and the way we write code differ from domain to domain. A DSM language therefore requires a matching code generator that also meets the requirements of the problem domain. In other words, to make sure that you are able to generate full code—written in the way you want it—from your DSM language, you need complete freedom in defining how your language maps to code.

Vendor-proprietary or fixed code generators are quite useless for a domain-specific modeling language. This becomes even more valid with the knowledge that both problem and solution domains evolve over time. Therefore, you need to be able to evolve your modeling language and code generator and don't want to be at the mercy of a tool vendor doing this for you.

When and How to Implement DSM?

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. These experts are therefore well qualified to define the automation mechanism that will make the rest of the developers much more productive.

Situations where we deal with a repetitive development effort are thus well suited for DSM adoption. A facet of repetitive development is the use of, or move toward a common product platform upon which product variants are developed. The product platform based approach to software development has its roots in the drive to better manage the complexity of offering greater product variety.

Key in this approach is the sharing and reuse of components, modules and other assets across a product family in order to develop new variants faster by assembling them from standard components. These standard components in a standard environment form the product platform, which raises the level of abstraction and at the same time limits the design space.

In most situations however, development teams use generic programming languages to develop software on top of this product platform. Why use a programming language that has been designed to develop any type of application when we are already working inside the boundaries set by the product platform? It is this product platform that is supposed to make things easier, so why not make optimal use of this by using a DSM language on top of the product platform? The code generator then maps the models to code that interfaces with the product platform.

But what if the product platform itself is subject to a lot of change, or several platforms or platform variants must be targeted? In these cases it makes sense to create a framework layer between the generator and the product platform(s). A change in the platform then only requires the expert to make a change in the framework layer. The code generator and DSM language that all other developers use can remain unchanged—and of course there is no need to change the models the developers have made.

This is in stark contrast to cases where applications are coded directly on a platform, when platform changes generally mean wholesale changes throughout most of the written code. Building a framework between the generator and platform can be a good idea even for a single platform. It can make creating the code generator a lot easier as many of the variation issues can be handled inside this framework layer, instead of having the code generator deal with all the details.

To summarize: DSM implementation requires the definition of a DSM language and code generator by the expert. It makes most sense in cases of repetitive development where it allows making optimal use of the development platform. DSM thus requires an investment of development resources and time to set up the DSM environment. Although this often clashes with the urgency of current development projects the improved productivity is often worth it.

Industrial experiences of DSM consistently report productivity being between 5 and 10 times higher than with current development approaches. Additionally, one has to bear in mind that it is better that the expert formalizes development practices once, and other developers' generated code automatically follow them, rather than having all developers try to manually follow best practices all the time.

Tools for DSM Implementation

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 much. 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.

Today, increased competition requires companies to reducing development cost and time to market. Improving the productivity of software development teams has become an important factor in achieving this. After realizing that UML in its standard form does not offer the possibility to significantly improve productivity, companies like my employer, MetaCase (MetaEdit+), along with Microsoft (Software Factories), Eclipse (EMF & GEF), and Xactium (XMF-Mosaic), have taken steps toward allowing users to build domain-specific model-based code generation environments.

Some do this by allowing the user to make UML more domain-specific by adding profiles to it, while others offer complete freedom in designing the DSM language and code generator. The effort that is required to do this differs from tool to tool. The ideal of course is a tool that gives plenty of freedom to truly raise the abstraction level of the language beyond code and make it as domain-specific as is needed, where more is usually better. Furthermore, the code generator should be fully open to customization, allowing the generation of code that looks like the best previous hand-written code.

In both of these areas, the tool should allow the expert to produce the desired results with as little effort as possible. In particular, the initial definition process should be as efficient as possible, allowing the expert to try out various language approaches quickly. Later on, when the language has been taken into use and real models exist, the tool should handle the evolution of the modeling language and automatically update models as much as possible.

How Does DSM Differ from MDA?

Currently, the Object Management Group (OMG) is intensively promoting its Model-Driven Architecture (MDA), a model-driven development method that comes down to transforming UML models on a higher level of abstraction into UML models on a lower level of abstraction.

Normally there are two levels, platform-independent models (PIMs) and platform-specific models (PSMs). These PIMs and PSMs are plain UML and thus offer no raise in abstraction. It is important here to distinguish between software platforms (what the OMG means with the word platform) like .NET, J2EE or CORBA, and the “product platforms” that I described earlier, which would be at least one level of abstraction higher.

In MDA, at each stage you edit the models in more detail, reverse and round-trip engineer this and in the end you generate substantial code from the final model. The aim the OMG has with MDA is to achieve the ability to use the same PIM on different software platforms and to standardize all translations and model formats so that models become portable between tools from different vendors. Achieving this is very ambitious but also still many years away. This focus however clearly defines the difference between DSM and MDA, and answers the question of when each should be applied.

DSM requires domain expertise, a capability a company can achieve only when continuously working in the same problem domain. These are typically product or system development houses more than project houses. Here platform independence is not an urgent need, although it can be easily achieved with DSM by having different code generators for different software and/or product platforms. Instead, the main focus of DSM is to significantly improve developer productivity.

With MDA, the OMG does not focus on using DSM languages but on generic UML, their own standard modeling language. It is not looking to encapsulate the domain expertise that may exist in a company but assumes this expertise is not present or not relevant. It seems therefore that MDA, if and when the OMG finally achieves the goals it has set for it, would be suitable for systems or application integration projects.

MDA requires a profound knowledge of its methodology, something which is external to the company and has to be gained by experience. Thus whereas the domain expertise needed for DSM is already available and applied in an organization, MDA expertise has to be gained or purchased from outside. In these situations the choice between MDA and DSM is often clear.

Conclusion

Throughout the history of software development developers have always sought to improve productivity by improving abstraction, automation and visualization. Domain-Specific Modeling combines these methods and copies the fundamental idea that has made compilers so successful: When you generate code, the process has to be complete.

These benefits present a strong case for the use of DSM but at the same time form an important obstacle: DSM implementation requires companies to define and maintain their own domain-specific modeling language and generators. A range of new tools are already helping to make this easier, allowing expert developers to encapsulate their expertise and make work easier, faster and more fun for the rest.

References

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

###

Dr. Steven Kelly is the CTO of MetaCase. He has authored over 20 articles and co-authored the first grammar of the Kenyan Orma language. He has over ten years of experience building meta-CASE environments and acting as a consultant on their use in domain-specific modeling.