The Model Craftsman

by Markus Voelter and John D. McGregor

INTRODUCTION

Everyone would like to have craftsman-level quality, but few are willing to devote the time or spend the money craftsmanship requires. Craftsmen working in hard goods, such as leather or silver, mitigate these factors by using devices such as patterns or molds that they can use to quickly replicate a high-quality design many times over. They have built these patterns based on their past experience. They've "materialized" experience into a tool. Using the tool still requires judgment, but the manual work of creating a product takes significantly less time and money.

Programmers can use the same approach as the hard goods craftsman. Domain-specific languages (DSLs), together with their execution environments, can provide the same benefits for the software craftsman. Just as with the specialized tools used by traditional craftsmen, DSLs are optimized for certain tasks in a programmer's field of experience. An effective way to make the development and maintenance more efficient is to reduce the number of lines of code people have to write, review, and maintain. DSLs help significantly with this, especially in the context of a software product line (SPL) where repeated use of DSLs amortizes the effort of creating them!

DSLs are different from general-purpose languages (GPLs) in some ways but similar in others. Traditionalists view the use of GPLs as "programming" and the use of DSLs as "modeling." In this article, we dispute that dichotomy and view programming as a craft that uses a range of languages that support the efficient expression of software on different levels of abstraction, from different viewpoints, and with different "grain sizes." We view the use and construction of DSLs as a natural element in a craftsman's toolbox. We believe code expressed with DSLs can be just as carefully crafted as code in a GPL. To distinguish code expressed with DSLs from code expressed in GPLs, we call the former "models." Please notice that models can be textual and hence quite similar technically to source code — we don't imply graphical notations by using the term "model."

Software product lines address the concerns about the time and expense of craftsmanship in a variety of ways, including modeling in the context of a software development approach that provides roles for both craftsmen and journeymen. An SPL establishes two major development roles: core asset developer and product builder. The core assets are the primary input into product building. They often constitute 75%-90% of the content of each product, and they are reused in many products. The assets include tools, processes, and best practices for building products, as well as models and code. Craftsmen, working as core asset developers, have a major impact on every product built in the product line. Journeymen concentrate on using the assets produced by the craftsmen to build a specific product and to learn the craft by exploring, using, and possibly tailoring the core assets.

In this article we describe the business and technical implications of a proven strategy for developing software-intensive products in which craftsman-level quality is achieved and products are still realized faster and cheaper than by using traditional development strategies. This strategy evolves the production capability of the organization as more of the craftsman's knowledge is made available to a growing community of journeymen. We will conclude with a few case studies.

BACKGROUND

A Bit about DSLs

Domain-specific languages are languages designed to address a specific task, concern, or aspect of a software system. A DSL is tailored very specifically to the task, concern, or aspect it addresses. Because of this, it is very good at what it is intended for. You can express a lot of semantics with very little code. It is not unlike a surgeon's tools. They are very good for, say, removing a tumor. On the other hand, they are not very useful for other types of cutting, such as cutting a hole in a wall. They are not general purpose. The focus on one specific class of problems makes them effective.

For execution, DSL programs are either translated down to more general-purpose languages for which an execution infrastructure exists, or, alternatively, you can provide an interpreter for directly executing your DSL. DSLs can address relatively general concerns, often related to technology or architecture. Examples include component structures, remote communication, persistence, or Web page structure and flow. An example is the communication backbone in the DFS case study or the OSGi artifact generation in the matrixware.net case study, both described below. In this case, there are often preexisting DSLs that can be reused from a library. However, DSLs can address very business domain–specific concerns such as calculation rules for insurance contracts, data processing for telescope images, or protocol definition for embedded systems. In this case it is very likely that a development organization will develop its own DSLs as part of core asset development.

If a DSL addresses business domain–specific concerns, the DSL can be used to effectively communicate with stakeholders — maybe even to allow them to create models on their own. But even if business domain people are not involved, DSLs can be a very powerful tool for developers. They can increase productivity, help formalize architectural or design decisions, and make functionality or technical knowledge more reusable. As a case in point, SQL is a long-established DSL that has saved many developers from knowing the details of a specific database implementation.

A Bit about SPL

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy specific needs of a particular market or mission, and that are developed from a common set of core assets in a prescribed way [Clements 02]. The SPL strategy has resulted in ten-fold productivity gains and as much as a 98% reduction in time to market. These gains have been experienced in a wide range of domains, from Web sites to medical imaging [Bell 09][Pronk 99].

An SPL organization offers an efficient blend of long-range strategy in the form of reusable core assets and shorter-term tactics for building products rapidly. The core asset approach is perfect for leveraging craftsmanship in an affordable manner by spreading the costs over a planned set of products. This deploys craftsman-level quality in every product produced.

An SPL is a restricted set of products chosen in part to maximize the features they have in common. This establishes the perfect conditions for defining DSLs: a well-defined and constrained context. The core asset base offers the opportunity to use a family of DSLs that cover different aspects of the product line feature set. For example, the SystemForge SPL defines DSLs to correspond to the model, view, and controller portions of a Web site [Bell 09].

An SPL organization performs a commonality/variability analysis on the products included in the product line, thereby producing a feature model. DSLs are used to express product specifics in a

way that conforms to the architecture common to all products in the SPL. SystemForge eventually evolved its set of DSLs to use a feature tree to relate pieces of each of the languages as they coexist in products. As features are selected for a product, elements expressed with each of the DSLs are added to the product.

THE MODEL CRAFTSMAN

A model craftsman is a programmer who uses DSLs to describe those aspects of a software system where a suitable DSL is more expressive or concise than a GPL. Model craftsmen spot those places in systems or product lines and build suitable DSLs if they don't yet exist. The model craftsman has many opportunities to add value to the program code, simplifying the life of the journeyman by providing guidance and "materialized" experience:

- The quality of the DSL is directly influenced by the domain knowledge of the craftsman. A domain-specific language is good if the words and idioms in the language adequately capture what needs to be represented. The more expert the craftsman, the finer the distinctions that are made and represented. This reduces the need to repeatedly tweak the language as it matures.
- The development of the interpreter or (more often) code generator for the DSL relies on the expertise of the craftsman in using the languages to which the DSL is translated and platforms on which those translations will execute effectively. Code generators often are template-driven. Each template is a piece of packaged design. The craftsman's skill in defining those packages directly determines the quality of the code emitted by the generator.

The quality of the core asset base (i.e., the inventory of program parts, tools, and processes) of the product line organization directly reflects the craftsman's domain and construction knowledge. Applying commonality and variability analysis, a craftsman can tease out the maximum amount of commonality based on deep knowledge of relationships in the domain. This results in a core asset base that has the appropriate abstractions for the efficient and effective construction of products.

DSLs provide a means for the craftsman to capture his expertise and provide it to journeymen and apprentices. They can use it to quickly produce craftsman-level products. They can study the DSL as they maintain it and come to understand the core concepts in the particular field of expertise. DSLs, in that sense, can help with communicating domain knowledge and programming experience among craftsmen and their followers.

BUILDING DSLs

DSLs have always been used for relatively broad, technical domains. Good examples include SQL, cascading style sheets (CSS), or regular expressions. Because they are useful for a large set of programmers, they are "available" and can just be used.

However, in this article we also argue that it is beneficial for programmers to build their own more narrowly focused DSLs. Whenever a craftsman is a specialist for certain classes of products and plans to build several similar but not identical products over time (i.e., starts a product line), he can exploit the benefits of DSLs for his specific products. Three preconditions must be satisfied for the creation and use of DSLs to be effective (we will elaborate on them below):

- 1. Once they are built, they have to blend easily into a programmer's existing workflows and environments (including general-purpose languages and tools).
- 2. The effort of building the language and the IDE must be justifiably low.

- 3. The effort of learning and potentially teaching the DSLs must pay off by reusing the DSL sufficiently often in different products within an SPL.

Integration into Existing Workflows and Environments

Graphical modeling tools or tools that use a proprietary database for models do not integrate seamlessly into the largely text-oriented infrastructure around GPLs. Textual DSLs integrate well and can, in many cases, be used inside the same IDE as the GPL. Current tools make it possible to deliver DSLs with their own language-aware editors with less effort than previously. These factors improve productivity, reduce training costs, and simplify adoption of DSLs in environments that are not necessarily open to "modeling."

The Effort of Building DSLs

Building DSLs and their IDEs requires two separate but interrelated activities: understanding the problem domain and implementing the language and tools. The time the analysis of the domain takes is largely determined by the domain expertise and general analysis and language definition experience of the assigned developers. However, a deep understanding of the domain is also necessary if DSLs are not used as part of the implementation, so, in some sense, this part of the overall effort does not count. The second point, the effort required for implementation of the languages, is greatly influenced by tools. Tools such as Eclipse Xtext or JetBrains MPS make the task of creating and maintaining DSLs orders of magnitude quicker than earlier generations of tools such as lex/yacc or antlr. Thanks to the use of more modern tools, the IDE can be completely generated from the language definition, and other tools such as code generators are easier to define. Our experience shows that the effort to build DSLs and the supporting tools is more than offset by the gains in productivity in product building.

Teaching and Learning the DSL

We often hear that it is a lot of effort for programmers to learn a DSL (especially for the journeymen who didn't themselves build it). Our experience does not confirm this. Assuming a programmer understands the concern for which the DSL is built (he'd also need to understand this if he didn't use a DSL!), then learning the language is not a big problem, especially if it comes with a good language-aware editor. And a good core asset also comes with suitable documentation called an attached process [McGregor 2010].

ACHIEVING CRAFTSMANSHIP QUALITY ON A BUDGET

JoelOnSoftware says, "Craftsmanship is, of course, incredibly expensive. The only way you can afford it is when you are developing software for a mass audience." A product line organization offers a twist on this by amortizing the cost of craftsman-level work over multiple products reaching mass production levels while customizing individual products through variable configurations thereby achieving craftsman-level quality on a budget. The split between core asset creation and product building offers an opportunity for a differentiated work force. The expensive craftsmen develop core assets while more reasonably priced journeymen and apprentices use those assets to produce products.

An SPL facilitates mass customization by sharing as much as possible and managing the amount of variation between products. The effort of building, maintaining, and establishing effective use of a DSL can be more easily justified in an SPL environment. And just as in traditional craftsmanship, the act of building tools and the act of using the tools to build products are distinct (while possibly interwoven) and build on each other.

DSLs and SPLs improve productivity so that fewer people can do more by programming with suitably tailored and optimized languages and automatically translating the programs to their respective implementation code. The improved productivity can be used to reduce costs by allocating fewer personnel to a product or to improve time-to-market by reducing the calendar hours required to produce the product — or by using the available resources for better analysis, testing, or documentation, tasks that often fall by the wayside.

ACHIEVING CRAFTSMAN-LEVEL QUALITY ON A SCHEDULE

A product line organization can achieve craftsman-level quality on a schedule by deploying craftsmen defining DSLs as core assets in the SPL, using those DSLs to define other core assets and then using agile techniques to produce the products from those core assets.

Establish all of this gradually in a highly iterative manner. Build small languages (or language extensions) for specific problems and integrate this technique with existing practices such as pair programming and unit testing.

Product building in an SPL is a naturally agile activity in which assets can be configured and integrated rapidly and incrementally to allow for intense customer interaction and feedback. Each product can be built by a small team.

SPL practices can easily incorporate such agile approaches as stakeholder involvement, rapid feedback, and value-based prioritization. An SPL organization involves stakeholders in numerous ways, such as feature identification, architecture definition, and testing. Many product line organizations divide the core asset creation and product building roles into separate teams. The core asset team provides releases of the core asset base periodically, and the product building teams provide feedback on improvements and bug fixes. When an organization first adopts the product line strategy, management must make decisions about which core assets should be developed first. Value-based prioritization is a decision-making tool that optimizes the sequence of core asset development.

CASE STUDIES

This section contains a few examples that illustrate the use of DSLs and product line engineering (PLE) in an agile context.

Deutsche Flugsicherung (DFS)

German Air Traffic Control recently reported [DFS10] on a “legacy renovation project” where they reengineered their training and research simulator product line. These simulators are used to train air traffic control (ATC) personnel and to research new ATC concepts. DFS maintains an installed base of dozens of simulators. Each of these simulators has its own particularities, but they are all built on a common base. Specifically they are built on a data replication backbone that connects distributed nodes implemented with different technologies. These include centralized simulation engines as well as various kinds of operator consoles that display air traffic data and allow simulation participants to enter commands.

As part of the reengineering project, one goal was to automatically generate the data replication backbone. Based on a textual DSL that describes the replicated entities, the integration code with the middleware platform for the various implementation languages is generated. The generated entities also contain APIs for publishing updates to code that has subscribed to updates of these particular entities (displays, for example). Tim Gesekus of DFS reported: “Modeling and generating the entities leads to increased consistency throughout the system and avoids re-coding of repetitive aspects.” He adds, “We had started with UML as a modeling language first, but the

implications on our workflow were too severe. Using a textual DSL based on Eclipse Xtext is much more lightweight. Specifically, the models are just text files and can easily be integrated with the rest of the code." The next step in the reengineering project will be to introduce a meaningful component model, where components are also described via DSLs, using code generation to create the the implementation skeletons . This will enable the simple exchange of functionality between different simulators.

itemis/matrixware.net

Over the last months, itemis (the consulting company Markus Voelter works for) has been working on a Web-based patent research application with their customer matrixware.net. The system's backbone is OSGi-based. Programming directly against OSGi requires a lot of tedious work as a consequence of the dynamic nature of OSGi and the fact that a lot of declarative information has to be specified in manifest files. To make developers' lives simpler, itemis built a set of DSLs for describing subsystems, components, interfaces, and deployment structures. From these models, also built with Eclipse Xtext, all the relevant OSGi artifacts were generated: separate bundles for the implementation and the APIs, service trackers and activators for handling dynamic deployment and service discovery as well as various support artifacts for testing. Michael Krauter, a developer on the project who considers himself a let's-get-on-and-code-this-stuff person was overheard as saying "I always considered this modeling stuff useless overhead. But I really like what we do here. It does make working with OSGi much more convenient and does not at all feel like overhead."

JetBrains

The above cases are examples of external DSLs. Separate languages are created. These are used for modeling certain aspects of the overall system, and code is generated that is integrated manually with parts of the system. The JetBrains Meta Programming System (MPS) supports a different approach: incremental extension of Java (and other languages). MPS is developed as open source software by JetBrains, so it is not surprising that they themselves use the tool for developing their new software products such as the Web-based YouTrack issue tracking system. To facilitate Web application development, they have developed among other things an extension of Java for describing persistent entities, a statically typed and "cleaned up" Java script, as well as an HTML template engine that supports embedded Java expressions — all statically checked and with powerful IDE support. Konstantin Solomatov of JetBrains reports that the time to market is reduced and the quality of the products developed with MPS is increased compared to traditional methods. Going forward, JetBrains plans to develop new product platforms with MPS. Because many new products will be based on the same (Web-based) infrastructures, the language extensions can be reused for these products.

SystemsForge

SystemsForge has taken a craftsman approach while still pursuing the goal of low-cost, rapidly deployed products. Traditionally, low-cost Web site development has been more "hack and patch" than craftsmanship. It's extremely difficult to get a Web application for under $30,000 with even the basic best practices such as a repository with meaningful commit messages, a suite of automated regression tests, an automated build script, and a continuous integration testing environment. By using best practices from SPLs and domain-specific modeling, they've been able to deliver well-architected software with professional infrastructure at a lower price point than has typically been available.

Also, because they're continually looking at the value of the SPL rather than just a specific site, they're able to refactor, extend, and invest in their tooling and frameworks to continually make

them more valuable and better crafted in a way they couldn't if they were just looking at each project as a $10,000 or $15,000 work order.

SUMMARY

Model craftsmen are programmers. Their expertise lies in knowing when to create and/or use a domain-specific language and when to rely on more general techniques. Model craftsmen can work in many environments, but a software product line organization provides a natural context. The product line organization is structured to amplify the value added by the craftsmen by making craftsman-developed tools and materials available to the journeymen who are building products. The quality of the code in those products is directly related to the indirectly applied expertise of the craftsmen. The blend of technologies and product development strategy we have described amortizes the costs of craftsmanship over many products and deploys craftsman-quality tools to every product development team.

REFERENCES

[Bell 09] Peter Bell. "A High Volume Software Product Line", PPL 2009, 2009.

[Clements 02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[DFS10] Ralph Kar and Tim Gesekus. Modernisierung eines Legacysystems mit Mitteln der modellgetriebenen Softwareentwicklung, eine Fallstudie, OOP 2010

[Joel 03] JoelOnSoftware, Craftsmanship, **http://www.joelonsoftware.com/articles/Craftsmanship.html, 2003.**

[McGregor 10] John D. McGregor. "Attached Processes", in Journal of Object Technology, vol 9, no. 2, March-April

[Pronk 99] B. J. Pronk. "Medical Product Line Architectures," Software Architecture. TC2 First Working IFIP Conference on Software Architecture (WICSA1), 1999, 357-67. ISBN: 0 7923 8453 9.

BIOS

Markus Voelter works as an independent researcher, consultant, and coach for itemis AG in Stuttgart, Germany. His focus is on software architecture, model-driven software development, and domain-specific languages, as well as on product line engineering. Markus also regularly writes (articles, patterns, books) and speaks (trainings, conferences) on those subjects. In his spare time, he is a glider pilot and podcaster, co-producing the Software Engineering Radio and Omega Tau Science podcasts. Contact him via voelter@acm.org or www.voelter.de.

Dr. John D. McGregor is an associate professor of computer science at Clemson University, a visiting scientist at the Software Engineering Institute, and a partner in Luminary Software, a software engineering consulting firm. His research interests include software product lines and component-based software engineering. His latest book is *A Practical Guide to Testing Object-Oriented Software* (Addison-Wesley 2001). He writes the Strategic Software Engineering column for the Journal of Object Technology. Dr. McGregor is a Model Craftsman using DSLs such as AADL to develop core assets for software-intensive systems. Contact him at johnmc@cs.clemson.edu. His mailing address is School of Computing, Clemson University, Clemson SC 29634; 864-656-5859.