AUTOMATED PRODUCTION OF FAMILY MEMBERS: LESSONS LEARNED

Risto Pohjonen, Juha-Pekka Tolvanen
MetaCase Consulting
Ylistönmäentie 31
FIN-40500 Jyväskylä, Finland
{rise, jpt}@metacase.com

Product family development utilizes family commonalities to enable fast and safe variant design and implementation. The family development approach can be put into action effectively with family-specific methods. A modeling language is defined based on the family characteristics. The language sets the variation space for designing family members and together with generators allows fast and highly automated variant production. This paper reports experiences and lessons learned from designing modeling languages and generators for automating product family development.

1 INTRODUCTION

Most domain engineering approaches (e.g. Arango 1994, White 1996, Weiss and Lai 1999, Kyo et al. 1990) emphasize language as an important mechanism to leverage and guide development in a product family. A few experts do the domain engineering creating a domain-specific language for designing family members and generators for their implementation. Once applied at the application engineering level, all other developers have the opportunity to focus on developing family members with models using directly product domain terms, from which the finished product can be generated automatically. The language (with supporting tools) shifts the abstraction level of designs from coding concepts to the product concepts, makes the product family explicit to developers and effectively sets legal variation space. Basically, domain engineering raises the abstraction for a single range of products. Similar upward shifts in abstraction have occurred in the past when programming languages have evolved towards a higher level of abstraction.

Domain engineering is strong on its main focus, finding and extracting domain terminology, family commonalities and variabilities, but gives little help in designing and implementing languages for the engineered domain. Typically, it offers some parameters of static variation, but behavioral variation, rules between different variation points, and mapping to implementations are not acknowledged. This paper describes practices for complementing the results of domain engineering in order to design and use domain-specific modeling (DSM) languages for product family development. This is achieved by using method engineering, metamodels, and metaCASE tools. Method engineering is the discipline of designing, constructing and adopting development methods and tools for specific needs (Brimkkemper et al. 1996, Kelly & Tolvanen 2000). In particular, it emphasizes the use of metamodels to specify concepts, terms and variation rules of product family domain. Metamodel-based tools can then create modeling languages and generators based on the metamodel (i.e. product family).

This paper is based on the experiences of applying DSM languages and generators in different type of product families, ranging from embedded to financial products. Some families, and related DSMs respectively, can be characterized rather stable whereas others are under frequent change. Some families have their main variability in the user interface, whereas others have it in hardware setting, platform services, business rules, or in communication mechanisms. Size of the families vary from a few to more than 200 members. Size of development teams ranges from 4 to more than 300 developers per family. Largest product families have over10 million model elements and largest DSM languages have about 500 language constructs (metamodel elements in the language). The experiences on language creation were gathered with interviews and discussions with domain engineers, personnel responsible for the architecture and tooling and with consultants creating DSMs for product family development.

This paper is organized as follows. In the next section we present environment architecture for building and using domain-specific methods. This 3-level architecture is used to find appropriate computational models for specifying variation inside a family and to allocate variant specification to application development environment. The variant specification with domain-specific methods and software production with generators are briefly described with an example in Section 4. Section 5 summarizes the main lessons learned from language design and implementation.

2 ENVIRONMENT FOR DOMAIN-SPECIFIC MODELING

For a comprehensive DSM environment with full automatic generation of variants, three things are required: a modeling tool with support for the domain-specific language, a code generator, and a domain-specific framework (Czarnecki & Eisenecker 2000, Pohjonen & Kelly 2002). This basic architecture is illustrated in Figure 1. The left side represents the entities relevant for creating the environment – a task that is carried out by domain engineers, while the right side illustrates the use of the environment by engineers developing family members.

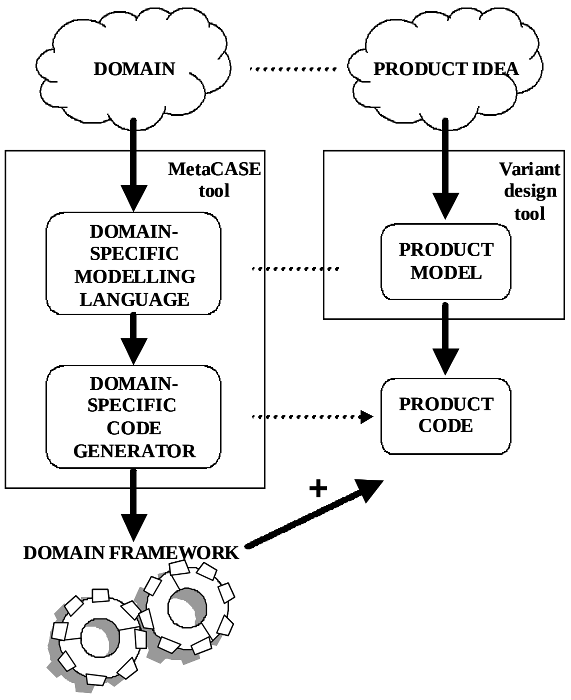

Figure 1. Architecture for designing and using a DSM environment.

The role of modeling language in a DSM environment is pivotal: as a representation of domain concepts and semantics, a modeling language defines static and dynamic variabilities setting the maximum variation space for the product family. The language is formalized into a metamodel and all models describing family members are instantiated from this metamodel. This instantiation of the language ensures that application developers follow the family approach de facto.

According to our experience, it is impossible to design a DSM language by extending current design languages that are based on fixed metamodels (e.g. UML, SA/SD). They simply lack the support for domain-specific concepts – those behind every different product family. These languages are also based either on the code world, using the semantically well-defined concepts of programming languages (e.g. UML, SA/SD), or on an architectural view using a simple component-connector concept. In both cases, the languages themselves “tell” nothing about a specific product family or its members. Instead, such information must be incorporated in an informal fashion into the model instances, e.g. via naming conventions, stereotypes, profiles, changing the original language semantics, using additional constraint languages (e.g. OCL or action semantics) or mappings between implementation independent and implementation dependent models, etc). These kinds of language extension mechanisms do not solve the underlying problem but just add more overhead to the use of language, thus putting even greater responsibility to developers. Instead, it is better to use languages that directly apply as their language constructs common family-specific terminology. This terminology is typically already known and in use, is more natural, reflects already to product variation, as well as is easier to use, remember, and read from specification models.

The task of the code generator is to translate the specific model semantics into an output compatible with the domain-specific framework and to provide variation for output formats. It is worth to note that restrictions imposed by the domain and the expected output make generator development remarkably easier, when compared to generators that are expected to work universally. By executing the generators, application developers can support faster variant production with fewer errors. Experience reports on applying generators with languages targeted to specific domains have shown remarkably fewer errors (e.g. Kieburtz et al. 1996 reports 50% less). The task of the domain-specific framework is then to provide the DSM environment with an interface for the target platform and programming language. This is achieved by providing the atomic implementations of commonalities and variabilities as framework-level primitive services and components.

Experience from various type and size of product families and reported industrial experiences have proven the viability of this architecture. For example, Nokia states that in this way it now develops mobile phones up to 10 times faster than earlier (Kelly & Tolvanen 2000), and Lucent reports that domain-specific languages improve their productivity by 3-10 times depending on the product (Weiss and Lai 1999). The authors are not aware of any other such widely and successfully applied architecture for product family development environments.

3 DEVELOPING FAMILY PRODUCTION FACILITIES

The starting point for building a family production facility is the domain analysis. Domain analysis (see Arango 1994 for a survey) identifies the commonalities and variabilities among possible family members and refines variation points by their characteristics, e.g. static or dynamic variation and parameters of variation space. If such analysis can't be made, most likely due to an unstable or immature domain, it is difficult, if not impossible, to put the (automated) product family approach into use!

Like any analysis, domain analysis does not provide any concrete implementations. As a next step, domain design and implementation are needed to develop the family architecture and facilities for automated variant production. According to our experiences, especially two aspects need to be considered when designing the production facilities. The first one is the computational model that is suitable for specifying the required variation. Another aspect is the required code generator output and its target platform and implementation language. These two aspects affect each other: sometimes the generation output may require a certain computational model to be used, e.g. XML and data models, when most variation points are based on static structures; or vice versa, the state machine as a computational model and the state machine as an implementation of behavior. The computational model(s) of variation and underlying platform for generator output are then represented with the elements of DSM environment, modeling languages, generator and domain-specific framework.

Another important issue related to the development of family production facilities is the tool support. Traditionally, the initial investment for developing production facilities has been very high as there were no cost-effective ways to implement the tool support. Thus, building the language and generator support was only feasible for large organizations. More recently, however, metaCASE tools with customizable modeling languages and code generators have emerged. These tools already include built-in environments for both domain engineering and product development with editors, browsers, multi-user support, etc. With metaCASE tools available, the effort of providing automated production facilities is reduced to a few man-weeks only, focusing mainly on specifying the language and generator definitions into the tool. As a result, the tool makes the defined language and generators automatically available to the developers, thus enforcing the product family development approach to be followed de facto.

3.1 Designing the modeling language

As the modeling language is the only part that is visible for the user and thereby provides the user interface for the development, it has to maintain control over all possible variation within the product family. The modeling language is also the main factor for productivity increase and it should operate on the highest achievable level of abstraction: apply product concepts and rules directly as constructs of the modeling language. Thus, the language is kept as independent from the target implementation code as possible. It may initially appear easier to build the language as an extension on top of the existing code base or platform but this usually leads to a rather limited level of abstraction and mapping to domain concepts.

To ensure a high abstraction level for developers, the language should be based on the product family domain itself. The optimal way to achieve this is to use the elements of product family architecture, common elements, and particularly those related with variation points. The nature of variation (static or behavioral) and level of detail favors selecting computational models that can be represented with certain basic modeling languages. Pure static variability can be expressed in data models, while orderly variation requires some sort of flow model, state machines advocate state models, etc. All these can be represented formally with metamodels and enriching them with variation data and rules allows creating the conceptual part of the modeling language. Once defined, the modeling language (enacted by the supporting tool) guarantees that all developers use and follow the same product family rules.

In most cases it is not possible to cover all variation within just one type of model and modeling language. This raises the important questions of model organization, layering, integration and reuse. Modeling language development efforts typically start with a flat model structure that has all concepts arranged on the same level without any serious attempts to reuse them. However, as the complexity of the model grows, while the number of elements increases, the flat models are rarely suitable for presenting hierarchical and modularized software products. Therefore, we need to be able to present our models in a layered fashion.

An important criterion for layering is the nature of the variability. For example, a typical pattern we have found within the product families is to have a language based on behavioral computational model (like state machine) to handle the low-level functional aspects of the family members and to cover the high-level configuration issues with a language based on a static model (like data and component models). Another aspect affecting the layer structure is reuse. The idea of reuse-based layering is to treat each model as a reusable component for the models on the higher level. In this type of solution, the reusable element has a canonical definition that is stored somewhere and referenced where needed.

3.2 Developing the code generator

To enable the code generator to produce completely functional and executable output code, the models should capture all static and behavioral variation of the target product while the framework should provide the required low-level interface with the target platform. This and nothing less should be always the goal for the DSM environment and its code generator. This ambitious sounding objective can be achieved easier when the sub-domains and related languages provide formal and well-bounded starting point.

As the translation process itself is complex enough, the generator should be kept as simple and straightforward as possible. For the same reason, maintaining variability factors within the generator structure has been found difficult – especially when the family domain and architecture evolves continuously. Instead of generator-centric approach we have detected that before including any variability aspect into the code generator, the nature of the variation must be carefully evaluated: if something seems difficult to support with generator, consider raising it up to the modeling language or pushing down to the framework. This also means that the developer should do all basic decision-making (like choosing the type of the target platform, if there are many) on the model level.

According to our experiences, the generator is a proper place for approximately only two kinds of variation. As each target platform or programming language requires, at least partially, a unique generator implementation anyway, it is widely acceptable to handle the target variation within the generator. Another suitable way to use the generator for managing variability is to build higher-level primitives by combining low-level primitives during generation.

3.3 Developing the domain framework

In many cases the differentiation between the target platform and the domain-specific framework remains unclear. We have learned to rely on the following definition: target platform includes general hardware, operating system, programming languages and software tools, libraries and components to be found on target system. The domain framework consists of any additional component or code that is required to support code generation on top of them. It is must be noted that in some cases additional framework is not needed but the code generator can interface directly with the target platform.

We have found that, architecturally, frameworks consist of three layers. The components and services required to interface with the target platform are on the lowest level. The middle level is the core of the framework and it is responsible for implementing the counterparts for the logical structures presented by the models as templates and components for higher-level variability. The top-level of the framework provides an interface with models by defining the expected code generation output, which complies with the code and templates provided by the other framework layers.

4 EXAMPLE

In the previous chapters we have discussed experiences of building DSM environments without concrete examples. For a better understanding of the "real life" process of a DSM environment creation, consider the following example. Assume that you are developing a family of related digital wristwatch models. According to the domain analysis, each watch consists of a set of watch applications that define behaviours of such elementary functions like showing the current time, alarm and stopwatch. The watch applications are implemented in Java.[1]

The first step for developing a DSM environment for our watch domain was – as explained earlier –choosing the suitable computational model for presenting the watch applications. In this case we found it best to rely on the typical computational model used with embedded software, the state machine. The next step was to enrich and narrow the semantics of state machine to focus on the concept of the watch domain. An example of this kind of extended state machine is illustrated in Figure 2.

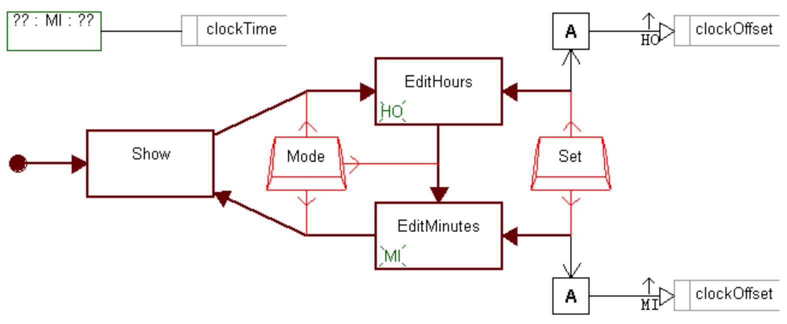

Figure 2. State machine with watch domain extensions

Basically, there are only two watch-specific extensions in our state machine. First, the transtitions can be triggered only by the user interaction when a certain button is pressed. Second, the actions taking place during the transition may only operate on time unit entities. Also the set of possible operations is limited: one can only add or subtract time units or roll then up or down. With these basic operations we can cover all current needs of our watch family. If further needs arise in the

[1] This is a part of a more comprehensive example. For the complete example, please contact us at {rise, jpt}@metacase.com.

future, we can simply extend the set of possible operations or define new entity types to operate on.

The definition of legal operations on time units gives us also a hint what is needed on the framework side of our DSM environment. In this case, as we were working with Java as our target language, we implemented a class of our own that could present time units (from milliseconds to hours) and perform the basic addition, subtracting and roll operations on them. During the code generation, all time unit entities are instantiated from this class and when a time unit operation is encountered, code generator creates a dispatch call to the appropriate platform interface primitive. An example of this kind of call can be found on lines 37 and 40 in Listing 1 that presents the code generated from the state diagram illustrated in Figure 2.

```
01 // All this code is generated directly from the model.
02 // Since no manual coding or editing is needed, it is
03 // not intended to be particularly human-readable
04
05 public class SimpleTime extends AbstractWatchApplication {
06
07   // define unique numbers for each Action (a...) and DisplayFn (d...)
08   static final int a22_1405       = +1; //+1+1
09   static final int a22_2926       = +1+1; //+1
10   static final int d22_977 = +1+1+1; //
11
12
13   public SimpleTime(Master master) {
14     super(master);
15
16     // Transitions and their triggering buttons and actions
17     // Arguments: From State, Button, Action, To State
18     addTransition ("Start [Watch]", "", 0, "Show");
19     addTransition ("Show", "Mode", 0, "EditHours");
20     addTransition ("EditHours", "Set", a22_2926, "EditHours");
21     addTransition ("EditHours", "Mode", 0, "EditMinutes");
22     addTransition ("EditMinutes", "Set", a22_1405, "EditMinutes");
23     addTransition ("EditMinutes", "Mode", 0, "Show");
24
25     // What to display in each state
26     // Arguments: State, blinking unit, central unit, DisplayFn
27     addStateDisplay("Show", -1, METime.MINUTE, d22_977);
28     addStateDisplay("EditHours", METime.HOUR_OF_DAY, METime.MINUTE, d22_977);
29     addStateDisplay("EditMinutes", METime.MINUTE, METime.MINUTE, d22_977);
30   };
31
32   // Actions (return null) and DisplayFns (return time)
33   public Object perform(int methodId)
34   {
35     switch (methodId) {
36       case a22_2926:
37         getclockOffset().roll(METime.HOUR_OF_DAY, true, displayTime());
38         return null;
39       case a22_1405:
40         getclockOffset().roll(METime.MINUTE, true, displayTime());
41         return null;
42       case d22_977:
43         return getclockTime();
44     }
45     return null;
46   }
47 }
```

Listing 1. Code generated from state diagram illustrated in Figure 2.

Listing 1 reveals also another important aspect of the watch implementation: the elementary state machine behaviour is implemented as an abstract class, from which the concrete state machine implementations are then derived during the code generation. This is an example of the implementation of counterpart for logical model construct. Similarly, the general structure of code in Listing 1 is an example of definition for expected generator output in model interface level of the framework.

The above described watch-specific language sets very effectively the variation space that was identified during the domain analysis. With the same language very different watch functionality can be described. For example, Figure 3 presents a more complex variant of our current time application with more editing possibilities. Capturing the variation into the models this way is very feasible mechanism for centralized managing the variation in the single source fashion. To summarize, using the DSM the manufacturer can now produce very fast and safely different variants of its wristwatch family: straight from family concepts in models to executable code.

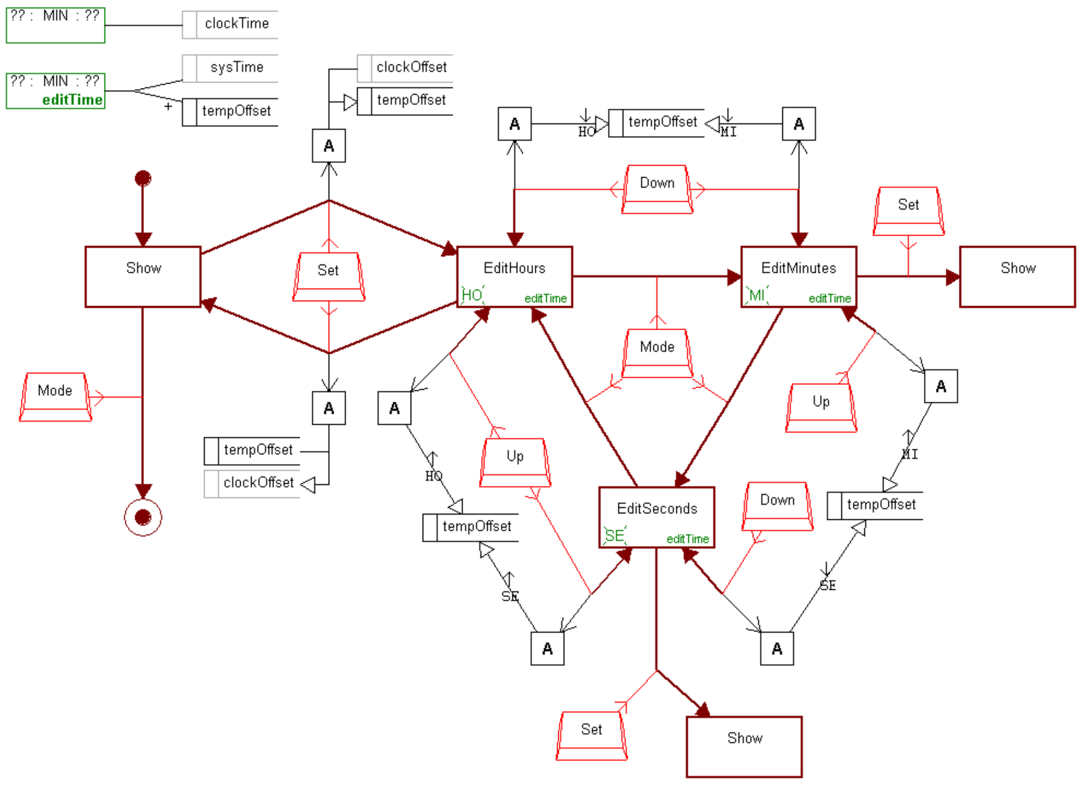

Figure 3. A more complex variant of current time application

5 CONCLUDING REMARKS

The lack of appropriate product specification and design languages has hindered a wider adoption of the product family development approach. Domain-specific modeling languages provide major benefits for product family development. They make a product family explicit, leverage the knowledge of the family to help developers, substantially increase the speed of variant creation and ensure that the family approach is followed de facto. These benefits are not easily, if at all, available for developers in other current product family approaches: reading textual manuals about the product family, mapping family aspects to code or code visualization notations, browsing components in a library, or trying to follow a (hopefully) shared understanding of a common architecture or framework.

In this paper we have presented architecture and experiences for designing languages and generators for product family development. We extend the domain analysis by seeking computational models for describing variation with design models. Our work is based on the use of metamodeling to design modeling languages that make a product family explicit: the family concepts and variation are captured in a metamodel that forms a modeling language. By instantiating the metamodel, models specify product variants within the family. The narrowed focus provided by the domain-specific languages makes it easier to automate the variant production with purpose-built code generators. Generators can incorporate some variant handling, but the possibility to bring it in front, into the modeling language, appears to be a better choice. Generally, the 3-level DSM environment architecture provides a wide variety of options for handling the variation, as opposed to approaches where variation can be handled in one place only. This is also important when supporting family evolution and reflecting the changes to the specifications under development.

Although building an automated family production facility requires an investment of the experts' time and resources, we have found that the investment pays itself back by the time the third variant is created. This approach also scales from small teams to large globally distributed companies. Interestingly, the amount of expert resources needed to build and maintain a language and generators does not grow with the size of product family and/or number of developers.

REFERENCES

Arango, G., (1994) Domain Analysis Methods, In: *Software Reusability*. Chichester, England: Ellis Horwood.

Batory, D., Chen, G., Robertson, E., Wang, T., (2000) Design Wizards and Visual Programming Environments for GenVoca Generators, *IEEE Transactions on Software Engineering*, Vol. 26, No. 5.

Brinkkemper, S., Lyytinen, K., Welke, R., (1996) *Method Engineering - Principles of method construction and tool support*, Chapman & Hall

Czarnecki, K., Eisenecker, U., (2000) *Generative Programming, Methods, Tools, and Applications,* Addison-Wesley.

Kelly, S., Tolvanen, J.-P., (2000) Visual domain-specific modeling: Benefits and experiences of using metaCASE tools, *International workshop on Model Engineering*, ECOOP 2000, (ed. J. Bezivin, J. Ernst)

Kieburtz, R. et al., (1996) A Software Engineering Experiment in Software Component Generation, Proceedings of 18th International Conference on Software Engineering, Berlin, IEEE Computer Society Press.

Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson, (1990) Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University

Pohjonen, R., Kelly, S., (2002), *Domain-Specific Modeling*, Dr. Dobb's Journal, Vol. 27, 8.

Weiss, D., Lai, C. T. R., (1999) *Software Product-line Engineering,* Addison Wesley Longman.

White, S., (1996) Software Architecture Design Domain, *Proceedings of* Second Integrated Design and Process Technology Conf., Austin, TX., Dec. 1-4, 1: 283-90.