MODELLING LANGUAGES FOR PRODUCT FAMILIES: A METHOD ENGINEERING APPROACH[1]

Juha-Pekka Tolvanen, Steven Kelly
MetaCase Consulting
Ylistönmäentie 31
FIN-40500 Jyväskylä, Finland
{jpt|stevek}@metacase.com

Abstract

Current modelling languages are based on the concepts of programming languages, leading to a poor mapping to product family characteristics and difficulties in leveraging the benefits and efficiencies of product family development. Method engineering provides one solution. It suggests developing modelling languages that map to a specific domain: here to a family and its various characteristics. A modelling language (i.e. a metamodel) is defined based on the family characteristics. The metamodel sets the variation space for possible models of variants and provides the basis for generators. In this position paper we focus on modelling languages supporting variant design and generation. Based on method engineering, we investigate principles for defining modelling languages for product family development.

1 INTRODUCTION

It is widely acknowledged that family and variant information should be extracted from implementation. Current modelling languages, however, provide surprisingly little, if any, support for product family development. They are either based on the code world using the semantically well-defined concepts of programming languages or based on an architectural view using a simple component-connector concept. In both cases, the languages themselves "say" nothing about a product family or its variants. Instead such information must be given in model instances, e.g. via naming conventions, stereotypes, changing the original language semantics, etc.

These instance level extensions are very limited for expressing family semantics and the rules of developing a specific product. For example, what does it mean if a class stereotyped <<display>> is associated with another class stereotyped <<button>>? It can be done in UML, but is it allowed as a possible variant specification? Does it follow the family architecture? Use of more powerful constraint languages in models would not improve the situation at all. They would allow even more possibilities to specify variants

[1] We would like to thank method engineers, Janne Luoma and Risto Pohjonen, for sharing their experience and insights into modelling language construction.

that break the family rules! More important than expressive power, however, is the support for following the development approach. If a language does not provide explicit support for product family development there is no guarantee that developers can follow it and that variants are developed as a part of the family. This resembles the situation of programmers being asked to make object-oriented programs where the language does not support any object-oriented principles. Things don't become classes just by calling them classes!

Most Domain Engineering approaches (e.g. DFR/White 1996, FAST/Weiss and Lai 1999, FODA/Kyo et al. 1990) emphasise language as an important mechanism to leverage and guide product family development. Domain Engineering is strong on its main focus, finding and extracting domain terminology, architecture and components; but gives little help in designing, constructing and introducing languages for the engineered domain. Here Method Engineering complements Domain Engineering. Method Engineering is the discipline of designing, constructing and adopting development methods and tools for specific needs (Brinkkemper et al. 1996, Kelly and Tolvanen 2000). In particular, it emphasises the use of metamodels to specify modelling languages and metamodel-based tools to support constructed methods. Recently, the use of metamodel-based technology has become widely accepted (e.g. via ISO and OMG frameworks). Similarly, a global survey by Seppänen et al. (1996) found that the direction in many companies is away from 'universal' methods and towards domain-specific methods implemented with metamodels.

In this position paper we investigate principles for defining modelling languages for product family development. The characteristics of a product family form the constructs of the modelling language, and the boundaries and restrictions of the family form rules and constraints for the models. Together these form the metamodel. The models based on this metamodel specify products and variants within the family.

2 REQUIREMENTS FOR MODELLING LANGUAGES FOR PRODUCT FAMILY DEVELOPMENT

In a family-specific modelling language, the models are made up of elements representing things that are part of the product domain world, not its implementation in the code world. More specifically, at least the following requirements can be set for modelling languages supporting product family development:

- Capture product family semantics. This allows developers to work directly with product family concepts, abstractions and rules, and makes the product family explicit. By using familiar terminology, the models become easier to remember, read and maintain.

- Set variation space within the family to enable easy and fast production of new variants. The design language should guarantee that variants specified follow the product family. This is achieved automatically if variants are specified based on the metamodel of family/language. Note that not all knowledge about variants is necessarily in the design language: it can be also in generators or in the platform into which variant information is input. For example, to the same models we can run different generators (based on the selected variant), or models can include data that provide parameters for the generator to be run.
- Allow both high and low level variant specifications. One fundamental reason to move towards product family development is to improve reuse of high-level components. An often-mentioned vision is a situation where relatively general requirements are mapped to the available services of the family. In practice, however, most cases require a much more detailed variant specification.
- Support variant specification in both static and dynamic forms. Most component-connector languages are declarative, supporting the assembly of static "components". Although this is adequate in a few cases, the more powerful creation of variants requires that developers also specify dynamic characteristics. This necessitates functional languages (e.g. state machines).
- Support use of existing components. The modelling language (and supporting tool) should actively guide developers to use existing components before creating new instances.
- Support definition of components. New functionality made for a variant (following the family characteristics) must be able to be promoted to become a component.
- Maintain family and variant traces. The language (and tool) should support the maintenance of models (and related sub-models) when a design element in one place is changed. Minimum support would be that changes in the family can be reflected to the variants.

3 HOW TO CONSTRUCT A FAMILY-SPECIFIC LANGUAGE

In Method Engineering, language construction is a metamodelling issue: based on the requirements a metamodel of the modelling language is defined[2]. Metamodelling languages are applied to describe domain concepts, their properties, rules, connections, model layering, etc. With a metaCASE environment the metamodel can be instantly tested by making example models.

[2] It must be noted that the language is not expected to be the whole solution: tool support is needed for language use, model checking, documentation and variant generation.

The language creation process is largely iterative by nature. The experience and intuition of the expert, combined with hints from the family, components and architecture are the real sources of clues. The following guidelines for constructing product family languages have been found useful in cases of family language creation[3]. Our intention is to put the instructions as "request for comment" during the workshop.

- Make the modelling concepts (metamodel) correspond to family concepts.
- Divide the product into parts to find relevant concepts and their aggregation structures.
- Prefer concepts already known and/or in use. Examine available specifications and models, but note that relevant concepts are not necessarily used in existing instances of models. Design principles used by experienced developers are a good source of information.
- In the beginning try to stick with family concepts; only later should you add concepts necessary for software production (generating code, configuration data, referring to code-components, etc.).
- Extract static and dynamic variability with different modelling languages. Often declarative languages support high-level customisation (like choosing available components) and functional languages are needed to specify dynamic aspects of variation. This reflects to the fact that often some variation is static in nature (choosing components and elements) whereas some variation is related to the behavior (e.g. among the selected components).
- Make a variant specification language to map available components. In other words, try to prevent developers from making specifications if reusable ones are available. Use scenarios to find out the constructs of modelling language that need to refer to components.
- Divide the modelling constructs into different modelling languages based on the required level of variability. Instead of having only one type of model to show all variability (which seldom is the best way) there can be separate languages for different "subdomains". Often one high-level static view is considered viable starting point, but it can also be functional view, like workflow describing the use of the product in some specific case.
- Try to stick with a few separate modelling languages, because integration and consistency among several types of models easily becomes difficult, if not impossible (e.g. as among different types of models in UML).
- Implement model layering (organization of model) based on types of reusable models or model elements. Components can be then reused as whole models or their parts.

[3] About the cases: up to 200 members in a family, up to 200 developers, up to 10 M instances (model elements), up to 500 types (metamodel elements), both declarative and functional languages.

- If you notice certain patterns of model elements occurring repeatedly in many models, it may be possible to define a separate modelling construct as shorthand for the pattern. This pattern can then be related with the generators to get required generated result and makes the modeling easier and faster.
- Move to generators all the functionality that variant developers do not need to consider. In ideal cases, all issues dealing with implementation language are "hidden" from the variant specification. In this way models describe only variant functionality, not how it's implemented.
- In large product families it may not be technically possible, but a separate language for the whole family (like a root model) can be useful for tracing. This also means that models should be organized into family, components, and individual variant products.
- As the last step, add constructs needed for consistency checking, review and documentation purposes.

4 CONCLUSIONS

Modelling languages can provide major benefits for product family development. They make a product family explicit, leverage the knowledge of the family to help developers, make variant creation fast (5-10 times faster) and can guarantee that the family approach is followed. These benefits are not easily, if at all, available for developers in other current product family approaches: reading textual manuals about the product family, browsing components in a library, or trying to follow a (hopefully) shared understanding of a common architecture or framework.

In this position paper we have argued that a modelling language (or better the corresponding metamodel) must map closely to the concepts and rules of product family. We investigated and proposed principles for constructing modelling languages for product families. Our work is based on the use of metamodelling to make families explicit, and metaCASE tools to instantly test and use the language. This shift to the 'metalevel' means that family concepts are defined by expert(s) on the type level (metamodel), avoiding the situation where each variant developer (mis-)uses family concepts freely on the model level. This leverages expert developers' abilities, to empower other developers in a team.

5 REFERENCES

Brinkkemper, S., Lyytinen, K., Welke, R., (1996) Method Engineering - Principles of method construction and tool support, Chapman & Hall
(for Method Engineering see also http://www.cs.jyu.fi/~jpt/doc/thesis/ime-3_2.html)

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

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
(see also http://www.sei.cmu.edu/publications/documents/90.reports/90.tr.021.html)

Seppänen, V., Kähkönen, A. -M., Oivo, M., Perunka, H., Isomursu, P., Pulli, P., (1996) *Strategic Needs and Future Trends of Embedded Software*. Technology Development Centre, Technology review 48/96, Sipoo, Finland

Weiss, D., Lai, C. T. R., (1999) *Software Product-line Engineering*, Addison Wesley Longman.
(see also http://www.research.avaya.com/~weiss/pubs/DefiningFamilies.ps)

White, S., (1996) Software Architecture Design Domain, *Proceedings of* Second Integrated Design and Process Technology Conf., Austin, TX., Dec. 1-4, 1: 283-90
(see also web-pick http://www.isso.uh.edu/publications/A9697/9697-14.html)