Product Derivation through Domain-Specific Modeling: Collected Experiences

Risto Pohjonen, Juha-Pekka Tolvanen

MetaCase, Ylistönmäentie 31
FIN-40500 Jyväskylä, Finland
{rise, jpt}@metacase.com
http://www.metacase.com

Abstract. Domain-Specific Modeling raises the level of abstraction beyond programming by specifying the solution directly using product domain concepts. The final products can be generated from these high-level specifications. This automation is possible because both the language and generators need fit the requirements of only one company and domain. This paper discussed Domain-Specific Modeling approach to product family development. It derives guidelines for DSM language and generator implementation by collecting the experiences gained from different cases of model-based product derivation.

1 Introduction

Modeling languages have been seen as an important mechanism to manage variation and guide development within product families [1], [8], [10]. However, traditional all-purpose modeling languages, like SA/SD or UML, provide little or no support for this matter. These languages are based on a fixed metamodels and therefore lack possibilities to explicitly bind product variation points to modeling constructs according to the domain or product family requirements. As a solution to this problem, domain-specific information is often included into the models informally with naming conventions, stereotypes, profiles or additional constraint languages. Unfortunately these kinds of language extensions do not solve the underlying problem, but add more overhead to the use of language, cause mistakes and make it difficult to achieve modeling and product derivation support.

Domain-Specific Modeling (DSM) addresses these issues directly on the level of the language itself. It suggests that the variation within the product line should be managed with well-focused modeling language specifically tailored for the product domain – which is opposite to the traditional modeling languages that try to be as general as possible. A DSM language is defined with natural domain and product concepts. Identifying the variation points and their parameters and binding them to the domain concepts then covers the variation space for the product family. Once applied for the application development, the models made with DSM language capture all static and behavioral parts of the family member. This enables developers to

generate the finished product automatically from these models. The language thereby shifts the abstraction level of designs to the product concept level, makes the product family explicit to developers and effectively sets legal variation space.

This paper discusses how to adopt DSM for product variation management and derivation – according to experiences collected from different industrially applied product families. As only industrial DSM deployment cases were taken into account, there are still too few cases to valid a statistical analysis of them[1]. The experiences were gathered with interviews and discussions with domain engineers, personnel responsible for the architecture and tooling and with consultants creating DSM languages for product family development. Some of these families, and related DSMs respectively, can be characterized rather stable whereas others are under frequent change. Also the main variation aspect is different between families: user interface, configuration, hardware settings, platform services, business rules, or communication mechanisms. Sizes of the families vary from a few to more than 200 members, and size of development teams ranges from 4 to more than 300 developers per family. Largest product families have over 10 million model elements and largest DSM languages have about 500 language constructs. Some languages were already used more than 7 years whereas some were used just for creating first variants within the family.

The organization of this paper is as follows. In the next section we present the architecture for a DSM environment. This 3-level architecture enables the domain engineers to find appropriate computational models for specifying variation inside a family and to allocate variant specification to application development environment. Sections 3 and 4 discuss the creation of the DSM language for variability management and the code generator support for variant derivation with an example. Section 5 then summarizes the experiences gathered from DSM implementation for product families.

2 Environment for Domain-Specific Modeling

A comprehensive DSM environment with full automatic generation of variants requires three things: a modeling tool with support for the domain-specific language, a code generator, and a domain-specific framework [10], [4], [9], [5]. This basic architecture is illustrated in Fig 1. The left side represents the entities relevant for creating the environment – a task carried out by domain engineers – while the right side illustrates the use of the environment by engineers developing family members.

[1] Due to confidentality all cases can be not illustrated in detail, but some cases are collected at www.metacase.com/dsm.html.

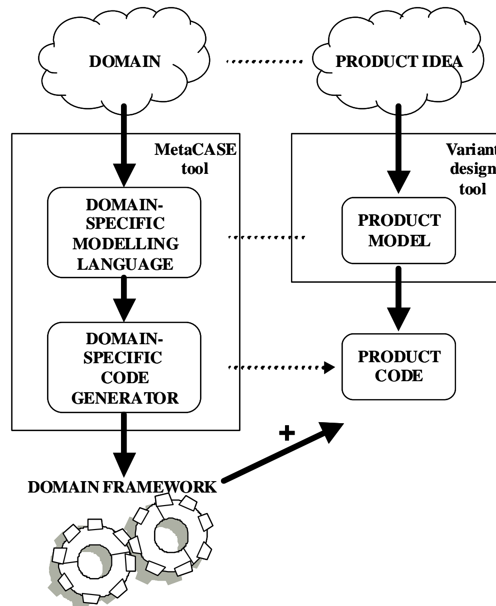

Fig. 1. Architecture for designing and using a DSM environment.

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 variation space for the product family. The language is formalized into a metamodel, from which all models describing family members are instantiated - this ensures that application developers follow the family approach de facto. The metamodel is also the key factor for the modeling tool support, which we will address shortly.

The code generator translates the specific model semantics into an output compatible with the domain-specific framework and provides variation for output formats. The domain-specific framework then provides 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.

Code generation also raises the question about the tool support for the DSM. Traditionally, there has been no cost-effective ways to implement full-blooded DSM environments with tools like editors and code generators. Over the 1990's, a new breed of customizable CASE tools, known as metaCASE tools, has emerged as a solution to this problem. A metaCASE tool is a development support environment with parameterized or configurable front- and back-end tools which runtime behavior is determined by the metamodel loaded into the environment. These kinds of metamodel-driven tools reduce the effort of building a DSM environment to a few man-weeks and change the focus of development mainly on specifying the metamodel for the language and building generators. The metaCASE tool then makes the language and generators automatically available for developers.

3 Variability Management with Modeling Language

In this and the next chapter we look at the process of building a DSM environment. We will begin with by examining how the variation within the product line is identified and then managed by incorporating it into the modeling language. As confidentiality prevents us from representing real world examples, we observe this process through a small but complete 'laboratory' example of wristwatch product family[2]. The basic idea in watch example is to consider a wristwatch as a set of small applications (like current time display, stopwatch, alarm, etc.) with user interface consisting of buttons and such display widgets as time zone or icon.

3.1 Domain Engineering

When we look at product development in the context of an overall product family, we need to define the product family in order to identify commonalities and differences among the related products. This process is known as domain engineering [10], [11]. The domain experts carry out the domain engineering and create DSM language and its support environment according to its results. Thus, the expert knowledge is leveraged to all developers, who can now concentrate on developing variants.

The starting point for language definition is the domain analysis that conceives the identification of domain concepts. The key strategy for finding domain concept is the commonality and variability analysis of the domain. The goal of this analysis is to identify the entities that are common for all products within the domain and find the variability between the products. However, it is important to understand that there are several ways to do this and that usually none of them alone can provide a complete coverage. Good results typically require concurrent use of a number of various strategies. In any case, the key success factor in finding the domain concepts is the domain expertise.

Once the commonalities and variabilities within the domain have been charted, the identified entities are categorized as either static or behavioral according to their nature. As an example, Table 1 summarizes the static and behavioral domain concepts of the watch example and their commonalities and variabilities, gathered as 2-by-2 matrix.

The next step of domain analysis is to adapt this rough presentation as a more formal definition of domain concepts. To do this, we have to analyze the relationships between the concepts and their possible variability attributes. The results of this analysis define the hierarchy of domain concepts from the top-level "whole product" to the low-level atomic elements. The identified variability attributes also partially set the variation space for the whole intended product range. The concepts for the example watch DSM environment and their variation space are presented in Table 2.

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

Table 1. The results of domain analysis for watch example

	Commonalities	Variabilities
Static	• display • button • zone • icon • application • action • time unit • alarm	• number of buttons • number of zones • number of icons • number of applications • combinations of displays and applications • combinations of actions
Behavioral	• applications are executed in certain order • application displays time units and icons • actions are triggered by user via buttons or by alarms • actions operate on time units, icons and/or alarms	• application execution order • time units and icons displayed vary depending on application • actions applied vary depending on application

Table 2. The watch concepts with variation attributes

Concept	Variability attributes
Watch	Logical Watch + Display
Logical Watch	Applications (0 – N) with execution order
Display	Buttons (2 – 4) + Zones (2 – 4) + Icons (0 – N)
Button	Up | Down | Mode | Set
Zone	TimeUnit
Icon	On | Off
Application	DisplayedTimes (0 - N) + Actions (0 – N)
DisplayedTime	Time
Time	TimeUnits (2 – 4)
Action	set Time | + Time | - Time | Icon on | Icon off | Alarm on | Alarm off
TimeUnit	Hours | Minutes | Seconds | Hundreds of seconds
Alarm	Time

The final task of domain engineering is to refine the results of analysis as the product reference architecture. This architecture can be derived fairly easily from the static commonalities and their aggregation structures. The watch example architecture is illustrated in Fig. 2.

It is worth to note that for the first deployable version of the DSM environment it may not be necessary to implement all concepts identified during domain analysis. It is possible to prioritize the concepts and implement of the DSM environment in incremental fashion.

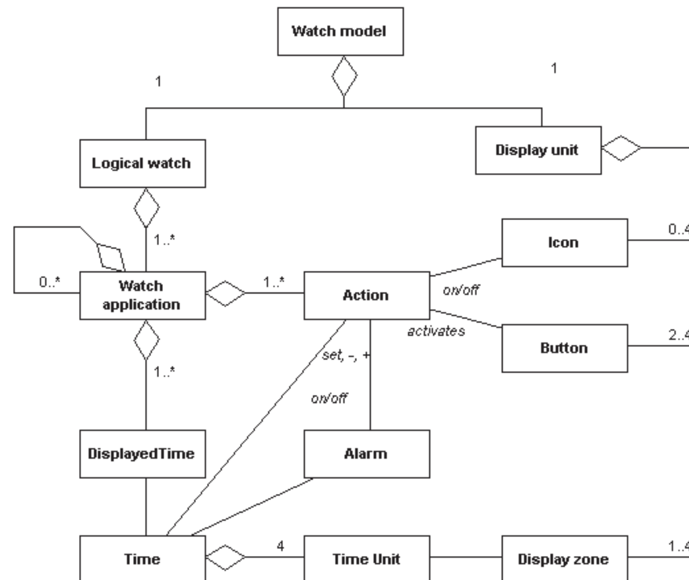

Fig. 2. The product architecture for watch example

3.2 Defining the Modeling Language

Domain engineering is strong on its main focus, finding and extracting domain concepts, 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 does not acknowledge behavioral variation, rules between different variation points or mapping to implementations. These aspects can be covered with additional techniques of method engineering and metamodeling. Method engineering is the discipline of designing, constructing and adopting development methods and tools for specific needs [3], [6]. In particular, it emphasizes the use of metamodels to specify concepts, terms and variation rules of product family domain. MetaCASE tools, as stated before, can then read these metamodel (i.e. product family) specifications to implement the tool support.

According to studied cases, there are two key requirements for DSM design. First is the computational model that is suitable for specifying the required variation. Another is the expected code generator output and its target platform and implementation language. These two requirements 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, mod-

eling languages, generator and domain-specific framework. The selection of computational model and underlying platform and programming language provide also additional information for successful distribution and allocation of domain concepts within the tree parts of DSM environment: the modeling language, code generator and domain-specific framework.

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. Based on our experience, language should be 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[3].

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.

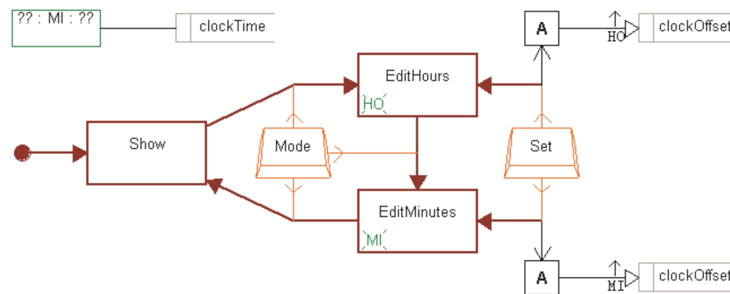

Fig. 3. State machine with watch domain extensions

Fig. 3 shows the example of modeling language for watch product family, presenting a simple application that displays and changes the current time. In this case we

[3] It seems not to be practice to design a DSM language by extending languages based on fixed metamodels. For example, pure UML profile mechanisms have limited capabilities to express product family concepts and their correctness constraints.

found it best to rely on the typical computational model used with embedded software, the state machine. We then enriched and narrowed the semantics of state machine to focus on the concepts of the watch domain. Basically, there are only two watch-specific extensions in our state machine. First, the transitions 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 (an example of more advanced variant of application shown in Fig. 3 is presented in Fig. 4). If further needs arise in the future, we can simply extend the set of possible operations or define new entity types to operate on.

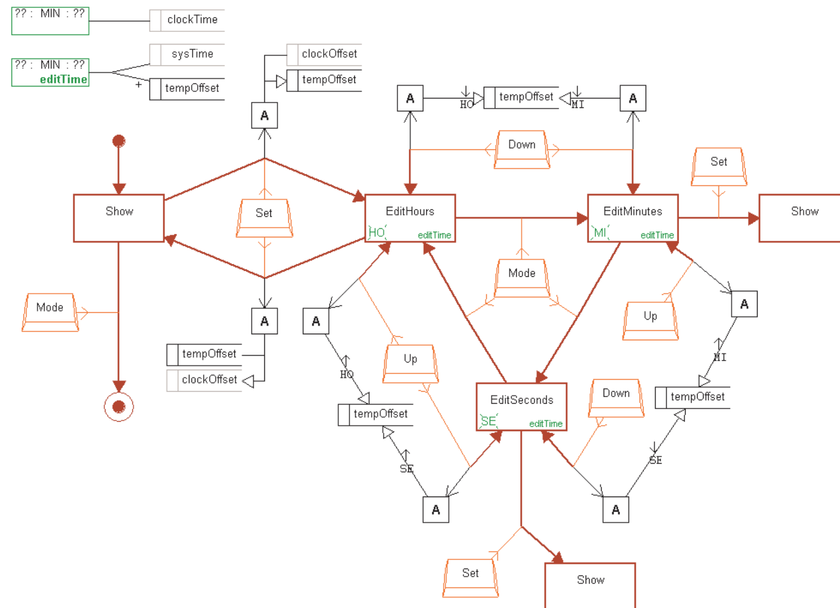

Fig. 4. A more complex variant of current time application

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.

4 Variant Derivation with Code Generator

In previous chapter we discussed how variation could be handled from within the DSM language. In this chapter we continue to familiarize ourselves how to derive the actual product variants from the models with code generator. Again, we use the wristwatch product line as an example.

4.1 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.

Listing 1 shows an example of code generated for the current time application shown in Fig. 2. The product derivation is complete in the manner that full code is generated from the modeler's point of view and manual rewriting of the code is not

needed. This completeness is crucial for model-based product development – it has been the cornerstone of other successful shifts made with programming languages. Moreover, domain-specific models describing the application functionality in code-independent manner gives possibility to use the same models to generate code for multiple platforms. As the example in Listing 2 shows, C code can be generated from the same designs: only the generator is different, not the product designs.

```
// All this code is generated directly from the model.
// Since no manual coding or editing is needed, it is
// not intended to be particularly human-readable

public class SimpleTime extends AbstractWatchApplication {

  // define unique numbers for each Action (a...) and DisplayFn (d...)
  static final int a22_1405   = +1; //+1+1
  static final int a22_2926   = +1+1; //+1
  static final int d22_977    = +1+1+1; //

  public SimpleTime(Master master) {
    super(master);
    // Transitions and their triggering buttons and actions
    // Arguments: From State, Button, Action, To State
    addTransition ("Start [Watch]", "", 0, "Show");
    addTransition ("Show", "Mode", 0, "EditHours");
    addTransition ("EditHours", "Set", a22_2926, "EditHours");
    addTransition ("EditHours", "Mode", 0, "EditMinutes");
    addTransition ("EditMinutes", "Set", a22_1405, "EditMinutes");
    addTransition ("EditMinutes", "Mode", 0, "Show");

    // What to display in each state
    // Arguments: State, blinking unit, central unit, DisplayFn
    addStateDisplay("Show", -1, METime.MINUTE, d22_977);
    addStateDisplay("EditHours",    METime.HOUR_OF_DAY,    METime.MINUTE,
d22_977);
    addStateDisplay("EditMinutes",     METime.MINUTE,     METime.MINUTE,
d22_977);
  };

  // Actions (return null) and DisplayFns (return time)
  public Object perform(int methodId)
  {
    switch (methodId) {
      case a22_2926:
        getclockOffset().roll(METime.HOUR_OF_DAY, true, displayTime());
        return null;
      case a22_1405:
        getclockOffset().roll(METime.MINUTE, true, displayTime());
        return null;
      case d22_977:
        return getclockTime();
    }
    return null;
  }
}
```

Listing 1. Java code generated for current time application.

```
typedef enum { Start, EditHours, EditMinutes, Show, Stop } States;
typedef enum { None, Mode, Set } Buttons;
```

```
int state = Start;
int button = None; /* pseudo-button for buttonless transitions */

void runWatch()
{
   while (state != Stop)
   {
      handleEvent();
      button = getButton(); /* waits and returns next button press */
   }
}

void handleEvent()
{
   switch (state)
   {
      case Start:
         switch (button)
         {
            case None:
               state = Show;
               break;
            default:
               break;
         }
      case EditHours:
         switch (button)
         {
            case Set:
               state = EditHours;
               break;
            case Mode:
               icon (Off,editHours);
               icon (On,editMinutes);
               state = EditMinutes;
               break;
            default:
               break;
         }
      case EditMinutes:
         switch (button)
         {
            case Mode:
               clockOffset = tempOffset;
               icon (Off,editMinutes);
               state = Show;
               break;
            case Set:
               state = EditMinutes;
               break;
            default:
               break;
         }
      case Show:
         switch (button)
         {
            case Mode:
               tempOffset = clockOffset;
               icon (On,editHours);
               state = EditHours;
               break;
            default:
               break;
```

```
        }
      default:
        break;
    }
    button = None;
    handleEvent(); /* follow transitions that do not require buttons */
}
```

Listing 2. C code generated for current time application.

4.2 Building Domain-Framework

In many cases the demarcation 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.

5 Conclusions

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. It is also worth to notice that experience reports on applying generators with languages targeted to specific domains have shown remarkably fewer errors (e.g. [7] reports 50% less). 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. DSM implementation seems

to require the extension of pure domain analysis by seeking computational models for describing variation with design models. Metamodeling is a viable technique to this kind of design of 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 with supporting tools, models can specify legal 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.

Implementation of DSM is not an extra investment in product family development. Rather, it saves development resources: traditionally all developers work with the family and variation concepts and map them to the implementation concepts manually. And among developers, there are big differences. Some do it better, but many not so well. So let the experienced developers define the concepts and mapping once, and others need not do it again. If an expert specifies the code generator, it produces applications with better quality than could be achieved by normal developers by hand. 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

1. Arango, G., Domain Analysis Methods, In: *Software Reusability*. Chichester, England: Ellis Horwood, (1994)
2. Batory, D., Chen, G., Robertson, E., Wang, T., Design Wizards and Visual Programming Environments for GenVoca Generators, *IEEE Transactions on Software Engineering*, Vol. 26, No. 5 (2000)
3. Brinkkemper, S., Lyytinen, K., Welke, R., *Method Engineering - Principles of method construction and tool support*, Chapman & Hall (1996)
4. Czarnecki, K., Eisenecker, U., *Generative Programming, Methods, Tools, and Applications*, Addison-Wesley (2000)
5. Greenfield, J., Short, K., Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools, John Wiley & Sons, to appear (2004).
6. Kelly, S., Tolvanen, J.-P., Visual domain-specific modeling: Benefits and experiences of using metaCASE tools, *International workshop on Model Engineering*, ECOOP 2000, (ed. J. Bezivin, J. Ernst) (2000)
7. Kieburtz, R. et al., A Software Engineering Experiment in Software Component Generation, Proceedings of 18th International Conference on Software Engineering, Berlin, IEEE Computer Society Press (1996)

8. Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
9. Pohjonen, R., Kelly, S., *Domain-Specific Modeling*, Dr. Dobb's Journal, Vol. 27, 8 (2002)
10. Weiss, D., Lai, C. T. R., *Software Product-line Engineering*, Addison Wesley Longman (1999)
11. White, S., Software Architecture Design Domain, *Proceedings of* Second Integrated Design and Process Technology Conf., Austin, TX., Dec. 1-4, 1: 283-90 (1996)