Companies defining families of related products would do well to reuse software design principles from one product to another, but modelling languages don't always support this concept, says **Juha-Pekka Tolvanen**

Keeping it in the family

REUSE HAS LONG BEEN A HOLY GRAIL IN software development, and reusing design principles is one of the most desirable achievements of them all. If you're a company developing a family of products, for example, there's no point reinventing the wheel—reusing software design elements across your product family seems like a no-brainer. Unfortunately, current modelling languages are based on the concepts of programming languages, making it difficult to map them to product family characteristics. Domain-specific modelling languages fitting to the product family provide a viable solution. In an ideal scenario, a modelling language (i.e., a metamodel) and variant generators are defined based on the family characteristics. The family-specific metamodel sets the variation space for possible models of variants and provides the basis for automated software production. Industrial applications of this approach show remarkable improvements in productivity—up to ten times faster.

Why code visualisation doesn't work

It is widely acknowledged that family and variant information should be separated from implementation[1, 2], but current modelling languages provide surprisingly little, if any, support for product family development. They are either based on the code world using the semantic concepts of programming languages (e.g. UML, SA/SD) 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, such as via naming conventions, stereotypes, changing the original language semantics and so on.

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, in a UML model, 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? Or does it follow the family architecture? Use of more powerful constraint languages attached to models (e.g. OCL) would not improve the situation at all. They would allow even more possibilities to specify variants 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. It would be like asking programmers to develop object-oriented programs in a non-OO language. Things don't become classes just by calling them classes!

Accordingly, family-specific modelling languages are needed to take full advantage of the product family development approach. These methods make product families explicit, shift the abstraction level of designs to the product concept level and allow fast and automated variant generation. Industrial experiences of this approach show major improvements in productivity, time-to-market responsiveness and training time[3, 4]. For example, Nokia states that in this way it now develops mobile phones up to 10 times faster, and Lucent reports that domain-specific languages improve their productivity by 3-10 times, depending on the product.

Domain engineering leads to family definition

When an organisation looks at product development in the context of an overall product family, it needs to define the product family in order to identify commonalities and differences among the related products. This process is known as Domain Engineering[1, 3]. A few experts conduct the domain engineering, and its results are used by the application developers. Thus, all the other developers have the opportunity to focus on developing variants. Domain engineering raises the level of abstraction by making common parts explicit. Similar upward shifts in abstraction have occurred in the past when programming languages have evolved towards a higher level of abstraction.

Most domain engineering approaches (e.g. DFR[5], FAST[3], FODA[6]) 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[7, 4]. In particular, it emphasises the use of metamodels to specify the concepts, terms and rules of a product family. Metamodel-based tools can then create modelling languages and generators based on the product family's metamodel. The ideal is that a developer would be able to develop the solution once only, as a model directly in product domain terms, from which the finished product variant could be automatically generated.

Recently, the use of metamodel-based technology has become widely accepted (e.g. via ISO and OMG frameworks). Similarly, industrial experiences of this approach show major improvements in productivity, time-to-market responsiveness and training time[3, 4].

What a domain-specific modelling language looks like

In a domain-specific modelling language, the models are made up of elements representing things that are part of the domain world, not the code world. The modelling language follows the domain abstractions and semantics, allowing modellers to work directly with domain concepts. This makes the product family explicit and can guarantee that the variants specified follow the product family. The tool support will then handle the generation of the final product. Note that not all knowledge about variants is necessarily in the modelling language: it can also be in generators or in the platform into which variant information is input.

Let's take an example. Suppose you manufacture a product family of digital wristwatches and your team develops the watch applications, such as stopwatch or world time, for different variants. Before any new variant features can be implemented, developers must design them in the watch domain. This process involves applying the terms and rules of the watch family, such as buttons, alarms, display icons and user actions. The domain-specific method applies these very same concepts directly in the modelling language. Using familiar terminology, the models become easier to remember, read and maintain.

The model in Fig 1 represents the time-setting feature: the actions a user can make by pressing buttons, the display elements blinking while being edited and the actions that increment or decrement the time. The model is based on modelling concepts specific to the family—all watches have buttons, and developers using these concepts specify variant data, so the time setting feature uses buttons called 'Set and 'Mode'. The modelling language also includes domain rules, preventing developers from making designs that are illegal in the family domain (such as connecting two watch buttons together).

As it can be see from the model, all the implementation concepts are hidden (in this case, using Java) from developers who can focus on finding the solution once only within the domain. As the descriptions in the domain level capture all required static and behavioural aspects of the application, it is possible to generate 100% of the code for a fully functional watch application from the models. Therefore, developers do not need to map the solution to implementation concepts in UML models or code.

When an organisation looks at product development in the context of an overall product family, it needs to define the product family in order to identify commonalities

Building a domain-specific modelling language

To get to a situation of domain modelling followed by full automatic code generation, three things are required: a modelling tool with support for the domain-specific modelling language, a code generator and a domain-specific component library. To understand how these things can be put together, let's walk through each development task using the watch modelling language as an example.

The first thing to do is to define what you want from the language. For example, the statement of purpose for the watch modelling language might read something like this: "The watch modelling language will enable us to model watch applications and generate Java code for a test environment executed in a web browser."

The next step is to analyse the family domain according to this statement of purpose and identify the domain concepts that capture the essence of the family. These can be found by studying the existing descriptions of the products, the domain terminology, the aggregate structure of the products, the domain knowledge and the domain 'folklore'. The key issue for finding these concepts is expertise provided by a domain expert or a small team of them. Typically, the expert is an experienced developer who has already developed several products in this family domain, developed the architecture behind the products, or has been responsible for forming the component library for the product family.

As the watch family is relatively simple, it is easy to define the domain concepts that form the family. There are physical product concepts like displays, buttons and widgets, logical product concepts like sub-applications or services, and behavioural concepts like setting alarms or editing a certain time unit. However, these alone do not make a modelling language yet. We need to apply the domain knowledge to put them together. For example, we know that we can use state machines to define how behavioural concepts can be combined to form sub-applications, and that the physical concepts provide the user interface. With this knowledge, we can now move on to define the required component library, modelling language and code generation by dividing the parts of the solution between these three entities.

Assembling the component library

While a component library is not necessarily needed in some cases, it usually makes the task of code generation development significantly easier. Furthermore, a component library often already exists from earlier development efforts and products. In such a case, the existing component library will most likely have already had its impact on the process of identifying the family concepts.

In the watch example, we need two base components: a template for the Java applet providing the user interface for the test environment, and a template for state machines defining the sub-applications. Both of these can be implemented as abstract classes that will be extended when the concrete classes are

inherited from them. For example, the abstract class for the state machine will contain the data structure for defining the state machine and the inherited concrete class will contain the code for loading the definition data. The concrete class also implements the methods that are executed during the state transitions.

Developing a modelling language involves three aspects: the domain concepts, the notation used to represent these in graphical models and the rules that guide the modelling process. This task also raises the question of tool support. Traditional CASE tools and code generators cannot provide sufficient support for modelling product family semantics and generating family members. Similarly, building your own CASE tool from scratch is both expensive and time-consuming. Fortunately, highly customisable metaCASE tools (such as MetaEdit+ and Ptech) now exist, providing a tailored environment for fast and easy implementation of the modelling language, as well as full tool support for that modelling language.

We can start by looking at the domain concepts identified during domain engineering. By allocating these concepts to the modelling language and refining them further, we can create the conceptual part of the modelling language. The goal here is to make the chosen concepts map accurately to the domain semantics. To provide the notation part of the modelling language, we also need to define symbols to be the graphical representations of these concepts to be used in models.

Concepts and symbols alone are not enough. A modelling language should also follow and enforce the rules that exist in the product family domain. The rules typically constrain the use of the language by defining what kinds of relationships are allowed between concepts and how certain concepts should be used. Once defined, the modelling language (enacted by the supporting tool) guarantees that all developers use and follow the same domain rules.

For the watch modelling language, we choose finite state machines as the basic model of computation. The familiar state and transition concepts can thus be used, providing us with perhaps 20% of our modelling language. In addition, the following semantic extensions are needed to make it suitable for our purposes:

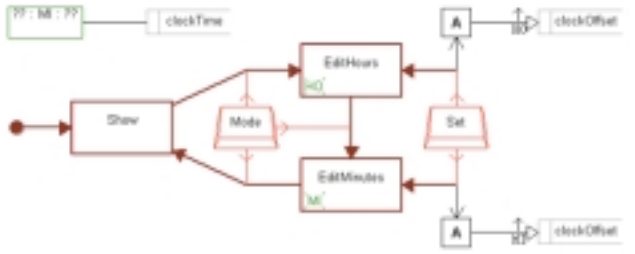
- The state transitions can be triggered only by button or alarm events.
- The state transition may include a set of actions that are performed during the transition. The language defines the legitimate actions—for instance, incrementing, decrementing or setting various time units that are either pre- or user-defined, and toggling alarms and icons on or off.
- Each state must be associated with a certain display function that defines the content of the display when the given state is active.

The extensions provide us with strong enough expressive power to make statically and behaviourally complete watch models that enable us to generate fully functional code from them.

Developing the code generator

Finally, we want to close the gap between the domain and code worlds by introducing a code generator. Building the code generation is basically a task of defining a set of rules that specify how domain-specific information is extracted from the models and transformed to the output that will compile and execute without any additional manual programming[8].

Figure 1. Modelling time setting feature



The key issue in building a code generator is how the model's concepts are mapped to code. The component library can make this task easier by raising the level of abstraction on the code side. For example, in the watch Java code, the alarms require a non-trivial thread-based implementation, but as the complexity of this implementation is hidden within the component library, the code generator needs to produce only a single method call to the service provided by the component interface. This is an example of how the mechanism of information hiding can be used to insulate the developers from complex or error-prone implementation aspects.

In our example, the state machine for each sub-application is generated from its model by inheriting the base functionality from the abstract class. We extend this functionality by loading the state machine definition data and defining the methods that are executed during the transitions and for the display functions. These methods themselves are actually dispatchers that make further calls to the services provided by the component library.

Conclusions

Domain-specific 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, substantially increase the speed of variant creation by between 500 and 1000%, and ensure that the family approach is followed de facto. These benefits are not easily available for developers in other current product family approaches, if at all. Such approaches include reading textual manuals about the product family, mapping family aspects to code or code visualisation notations, browsing components in a library or trying to follow a (hopefully) shared understanding of a common architecture or framework.

In this article we have argued that a modelling language (or, better, the corresponding metamodel) must map closely to the concepts and rules of the product family. Our work is based on the use of metamodelling to make a product family explicit: the family concepts and rules are captured in a metamodel that forms a modelling language. By instantiating the metamodel, models specify product variants within the family. This shift to the metalevel means that family concepts are defined by experts on the type level (metamodel). It avoids a situation in which each variant developer uses (or misuses) family concepts freely on the model level. This leverages expert developers' abilities to empower other developers in a team. ■

References

1. Jazayeri, M., Ran, A., van der Linden, F., *Software Architecture for Product Families*, Addison-Wesley, 2000.

2. Bosch, J., *Design and Use of Software Architectures*, Addison-Wesley, 2000.
3. Weiss, D., Lai, C. T. R., *Software Product-line Engineering*, Addison Wesley Longman, 1999.
4. Kelly, S., Tolvanen, J.-P., "Visual domain-specific modelling: Benefits and experiences of using metaCASE tools", *International workshop on Model Engineering*, ECOOP 2000, (ed. J. Bezivin, J. Ernst)
5. White, S., "Software Architecture Design Domain", *Proceedings of Second Integrated Design and Process Technology Conf.*, Austin, TX., Dec. 1-4 1996, 1: 283-90.
6. 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
7. Rinkkemper, S., Lyytinen, K., Welke, R., *Method Engineering - Principles of method construction and tool support*, Chapman & Hall, 1996
8. 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

Bibliography

- Czarnecki, K., Eisenecker, U., *Generative Programming, Methods, Tools, and Applications*, Addison-Wesley, 2000.
- Seppänen, V., Kähkönen, A. -M., Oivo, M., Perunka, H., Isomursu, P., Pulli, P., *Strategic Needs and Future Trends of Embedded Software*. Technology Development Centre, Technology review 48/96, Sipoo, Finland, 1996

Dr. Juha-Pekka Tolvanen is CEO of MetaCase (www.metacase.com), a company specialising in customisable modelling tools and code generators. He can be reached at jpt@metacase.com. Juha-Pekka's area of expertise is the engineering of software development methods, particularly for embedded product families. The material covered in this article is partly based on product family development research by the CAFÉ consortium, which includes Philips, Siemens, Nokia, Thales and MetaCase.

Listing 1: An example of code generated for the modelling time setting sub-application defined in Fig 1

```
Import java.util.*;

public class Simple extends AbstractWatchApplication
{
   public Simple(AbstractWatchApplet watchApplet)
   {
      super(watchApplet);
      addTransition ("Show", "Mode", "", "EditHours");
      addTransition ("EditMinutes", "Mode", "", "Show");
      addTransition ("EditHours", "Mode", "", "EditMinutes");
      addTransition ("EditMinutes", "Set", "a3_3210", "EditMinutes");
      addTransition ("EditHours", "Set", "a3_3207", "EditHours");
      addTransition ("Start [Watch]", "", "", "Show");

      addStateDisplay("EditHours", METime.HOUR_OF_DAY, METime.MINUTE, "d3_3134");
      addStateDisplay("EditMinutes", METime.MINUTE, METime.MINUTE, "d3_3134");
      addStateDisplay("Show", -1, METime.MINUTE, "d3_3134");
   };

   public Object a3_3207()
   {
      getclockOffset().roll(METime.HOUR_OF_DAY, true, displayTime());
      return null;
   }

   public Object a3_3210()
   {
      getclockOffset().roll(METime.MINUTE, true, displayTime());
      return null;
   }

   public Object d3_3134()
   {
      return getclockTime();
   }
}
```