# Boosting Embedded Systems Development with Domain-Specific Modeling

Domain-specific modeling promises order of magnitude productivity and quality improvements for embedded software development by leveraging the knowledge of domain experts into the generation of final code.

by Risto Pohjonen, MetaCase Consulting

In embedded software development, reuse is recognized as the key factor for better productivity and lower costs. In the ideal case, the product family approach makes it possible to apply reuse on a whole range of related products. The family members are based on a common architecture and rely heavily on reusable components, thus freeing the developers to concentrate only on the required variation between the products. This approach also enables the developers to focus on design instead of implementation details.

Traditionally, however, the product family approach has been hindered by the lack of adequate design tools. Most current design languages and tools provide little if any support for design level reuse and domain-specific design concepts, which are the key requirements for a successful product family solution.

## The Promise of DSM

Domain-specific modeling (DSM) is an emerging technology that addresses these issues. The basic idea of DSM is that the product family variants are described with a domain-specific modeling language that is based on the domain itself and the final products are automatically generated from these models. This is possible because the models capture all static and behavioral aspects of the product. Also, unlike the general-purpose code generators found in typical design tools, the DSM generators are specifically implemented to support the domain concepts found in the models.

DSM is an important mechanism for guiding development in a product family. The domain-specific modeling language and generators, created by a few domain experts, shift the abstraction level of designs to the product concept level, make the product family explicit to developers and effectively set legal variation space. Basically, this means that the domain knowledge of an expert is leveraged to the whole development team.

DSM is an important mechanism for guiding development in a product family.

Experiences from real-life cases of DSM uses have also reported more concrete benefits. The single most important promise of DSM is order of magnitude productivity improvements. For example, Nokia states that with DSM, it develops mobile phones up to 10 times faster than before. Similarly, Lucent reports that domain-specific languages improve its productivity by 3-10 times depending on the product. As the DSM

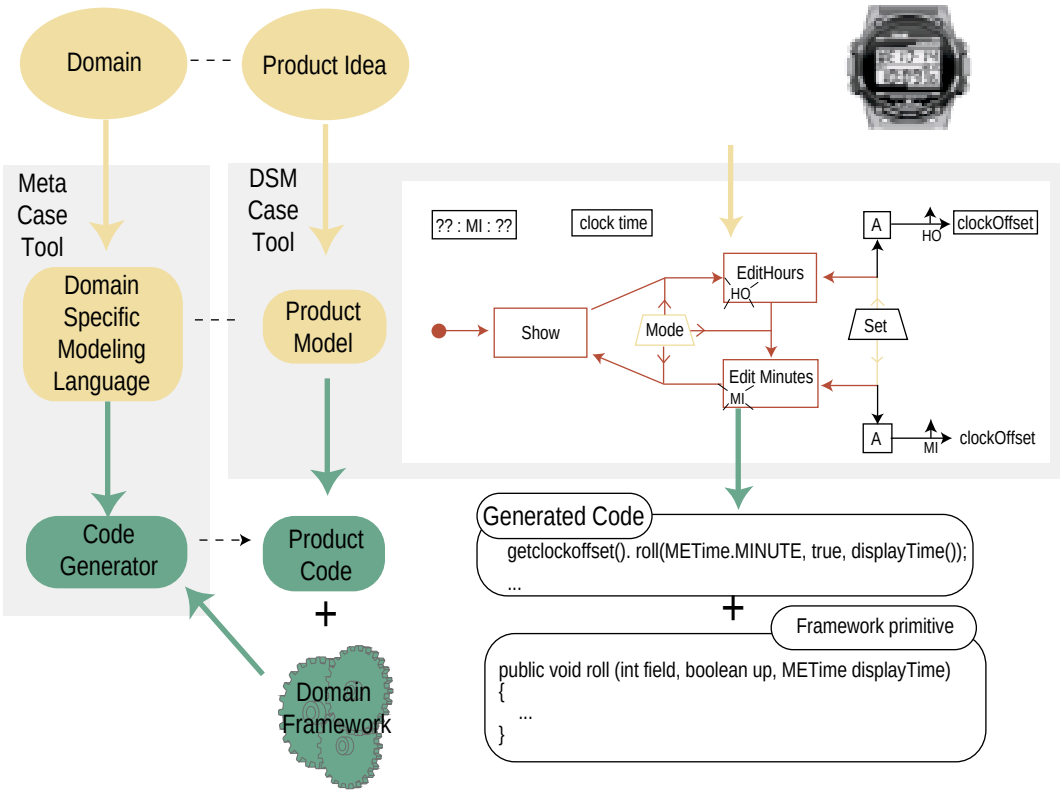


Figure 1 Architecture for designing and using a DSM environment.

solution is based on the domain and the terminology known and used by those working within the field, it is also easier and faster to introduce it to new people.

DSM also contributes to the improvement of the overall quality of the product. As the modeling language defines the variation space and the implementation is automatically generated, there are substantially fewer errors because the need for error-prone manual intervention is minimized.

## Architecture for a DSM Environment

For a complete DSM environment with 100% code generation, three things are required: a modeling tool with support for the domain-specific language, a code generator and a domain-specific framework. This basic architecture is presented on the left-hand side of Figure 1, with an example of a DSM environment for developing software for digital wristwatches on the right.

The role of the modeling language in a DSM environment is pivotal: as a representation of static and dynamic domain concepts and semantics, a modeling language defines maximum variation space for the product family. The language also ensures that developers follow the family approach and de facto standards. The task of the code generator is to translate the models into the required output format and to provide variation for output formats. The task of the domain-specific framework is then to provide the DSM environment with an interface for the target platform and programming language by providing the atomic implementations of commonalities and variabilities as framework-level primitive services and components.

```
// 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;
  }
}
```

Figure 2 Code generated from the example model in Figure 1.

These two aspects affect each other: sometimes the computational model may require a certain kind of generator output (e.g. a state machine as a computational model requires a state machine implementation or vice versa). 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.

The first part of a DSM environment to be implemented is the modeling language. As the language is the only part that is visible to the user and thereby provides the user interface for the development, it has to maintain control over all possible variations within the product family. The modeling language is also the main factor for increasing productivity. Therefore it should operate on the highest achievable level of abstraction and should be kept as independent from actual 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.

We have found that the best way to design a domain-specific modeling language is to base it on some well-known model of computation, like state machine, data model or flow model, and enrich this model with domain-specific concepts. For example, the modeling language for watch development (as illustrated in Figure 1), is based on the computational model of a state machine, which was complemented with the following domain-specific extensions:

- Transitions between states may be initiated by user actions only (hence the button symbol as a trigger).
- The actions during a state transition may operate on time units only, and there is only a limited set of legal operations.
- Each state is associated with a display function that defines how the time units are shown on display when the state is active.

## Building a DSM Environment

The starting point for building a DSM environment is to identify the commonalities and variabilities among possible family members and refine these variation points with static and dynamic variation attributes. These common elements and variation attributes are then allocated to the DSM environment architecture. In order to do this, two aspects need to be considered. First is the computational model that is suitable for specifying the required variation. The second aspect is the required code generator output and its target platform and implementation language.

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. Typically, efforts to develop a modeling language start with a flat model structure that has all concepts arranged on the same level without any serious attempts to reuse them. As the complexity of the model grows, 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 a behavioral computational model (like the 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.

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 domain framework should provide the required low-level interface with the target platform. As the translation process itself is complex enough, the generator should be kept as simple and straightforward as possible. It is also difficult to maintain variability factors within the generator—especially when the family domain and architecture evolve continuously. Thus, before including any variability aspect into the code generator, evaluate the nature of the variation carefully: if something seems difficult to support with the generator, consider raising it up to the modeling language or pushing it down to the framework. Also bear in mind that the developer should do all basic decision-making at the model level.

According to our experiences, the generator is a proper place for 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.

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

An example of generated code in Figure 2 illustrates how a domain framework can support the code generation. It is the

implementation of the state machine presented in Figure 1. As the watch domain framework provides primitives defining the state machine and operating on the time units, the required generator output remains reasonably simple and achievable.

Another issue related to the development of DSM environments is the tool support. Traditionally there have not been any cost-effective ways to implement complete DSM environments with the required editors and generators. More recently, however, tools with customizable modeling languages and code generators have emerged. These tools provide built-in support for both defining the DSM environments and applying them in product development. As MetaCase tools like MetaEdit+ or Ptech, the effort of building complete DSM environments is reduced to a few man-weeks.

DSM provides major benefits for product family development. 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.

As an investment, the creation of a DSM environment of course requires experts' time and resources, but 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.

MetaCase Consulting
Jyväskylä, Finland.
+358 14 4451 400.
[www.metacase.com.].