Into the domain of speed

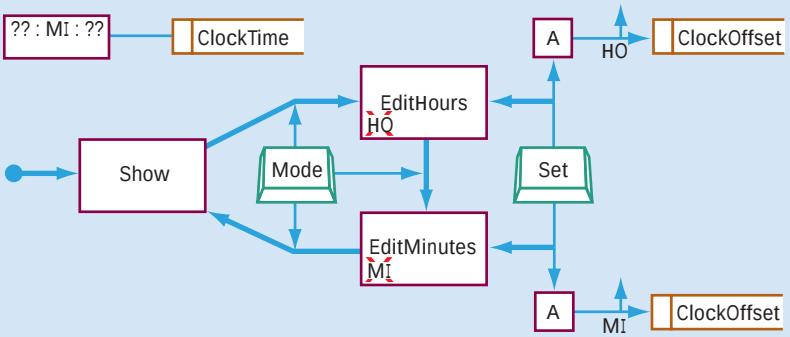
An upward shift in abstraction leads to a corresponding increase in productivity. In the past this has occurred when programming languages have evolved towards a higher level of abstraction. Today, domain-specific visual languages provide a viable solution for continuing to raise the level of abstraction beyond coding. **– BY JUHA-PEKKA TOLVANEN AND DAVID LARNER.**

Current modelling languages are based on the code world, leading to a poor mapping to an organisation's own domains and duplication of effort in problem-solving, design and coding. For example, traditional modelling languages such as UML and SDL do not relate directly to any application domain (eg, mobile phones, e-commerce platforms, point-of-sale systems etc) but to the implementation – basically they visualise the code. Therefore, developers have to leap straight from requirements into implementation concepts, and map back and forth between domain concepts, UML concepts, and program code. This takes a lot of time and resources and easily leads to errors.

Primarily in the embedded area, there have been moves towards a way of building software that removes this resource-intensive and error-prone mapping and shifts the emphasis to the domain, not its implementation in code. The ideal is that a developer would be able to develop the solution once only, as a model directly in domain terms, from which the finished product could be automatically generated. Industrial experiences of this approach show major improvements in productivity, time-to-market responsiveness and training time. For example, using this method Nokia now develops mobile phones up to 10 times faster, and Lucent has improved its productivity by between 3-10 times depending on the product.

To do the same, your team must move from modelling code to modelling products with domain-specific concepts. This article describes what domain-specific modelling is and how a domain-specific modelling language can be built for your company.

FIGURE: 1



What is a domain-specific modelling language

In a domain-specific modelling language, the models are made up of elements representing things that are part

FIGURE: 2

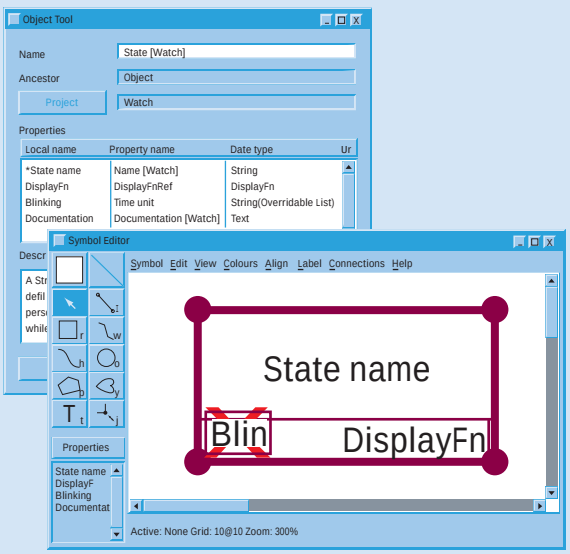


Figure 1 (right, top): Modelling time setting feature. Figure 2 (right): Conceptual and symbol definitions for 'State'

of the domain world, not the code world. The modelling language follows the domain abstractions and semantics, allowing modellers to perceive themselves as working directly with domain concepts. The tool support will then take care of the generation of the final product.

Let's take an example. Suppose you manufacture digital wristwatches and your team makes the watch applications, such as stopwatch or world time.

Before any new features can be implemented developers must design them in the watch domain.

This involves applying the terms and rules of the watch, such as buttons, alarms, display icons and user's actions. The domain-specific method applies these very same concepts directly in the modelling language. A simple example of such a modelling language is illustrated in figure 1.

The model 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. As you can see from the model all the implementation concepts are hidden (in this case Java) from developers and they can focus on finding the solution only once right in the domain. The ??:MI:?? element is a display function: it tells what variable value (ie, clock time) is shown in the watch display. In this case there is only one, telling that every state (Show, Edit Minutes, Edit Hours) shows clock time and the central point of display is minutes.

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 (eg, the model in figure 1) to implementation concepts in UML models or code.

The language also includes domain rules, preventing developers from making designs that are illegal in the domain (eg, connecting two watch buttons together) building a domain-specific modelling language.

Figure 4: The architecture of watch code generation

FIGURE: 3

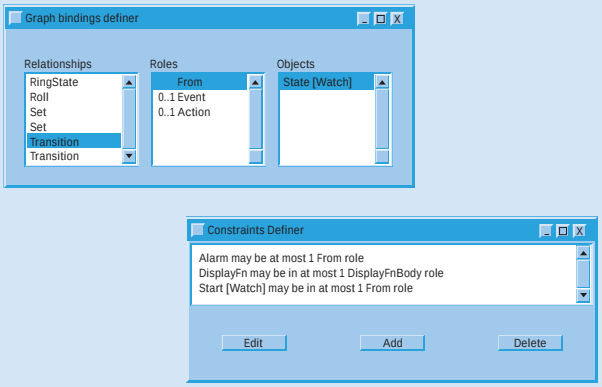


Figure 3: The definition of domain rules

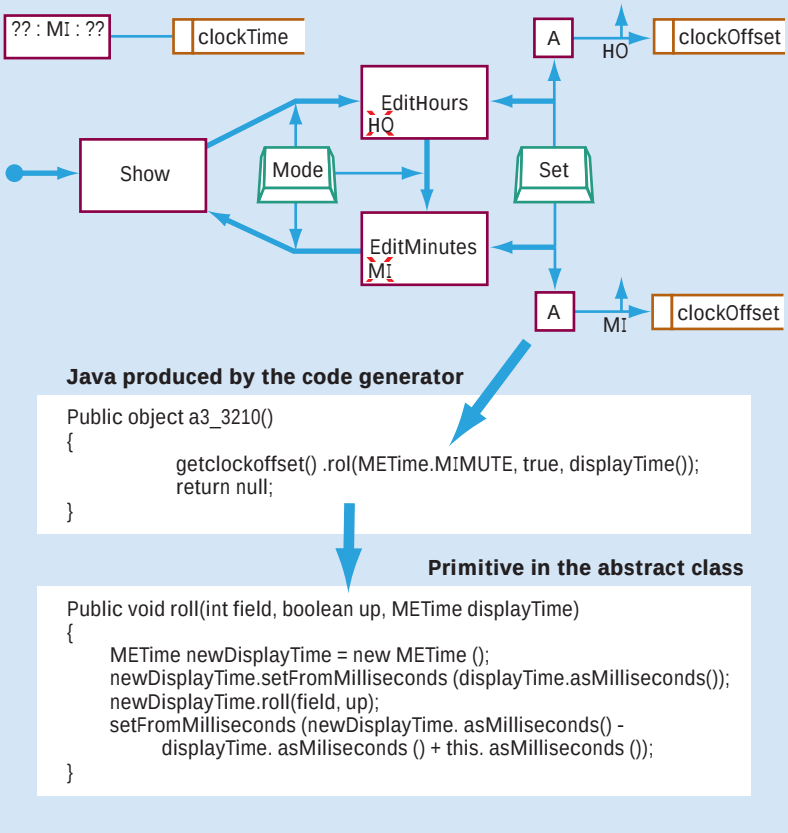
To get to a situation of domain modelling followed by full automatic code generation, three things are required: a modelling tool with support for the domain-specific modelling language, a code generator, and a domain-specific component library.

To understand how these things can be put together, let's walk through each development task using the watch modelling language as an example.

Defining the language solution for the domain

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

FIGURE: 4



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 domain according to this statement of purpose and identify the domain concepts that capture the essence of the domain. These can be found by studying the existing descriptions of the product, domain terminology, the aggregate structure of the product, domain knowledge and domain "folklore".

The key issue for finding these concepts is domain 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 domain, developed the architecture behind the product, or has been responsible for forming the component library for the product.

As the watch domain is relatively simple, it is easy to define the domain concepts. 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, but 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. In such a case the existing component library will most likely have already had its impact on the process of identifying the domain concepts.

In the watch example, we need basically 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 the modelling language

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 in-house modelling languages and code generators. Similarly, building your own CASE tool from scratch is quite an expensive effort. Fortunately, customisable tools like metaCASE and MetaEdit+ now exist, and can provide a tailored environment for fast and easy implementation of the modelling language, and full tool support for that modelling language.

We can start by looking at the domain concepts identified during the domain analysis. 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.

Figure 2 shows the conceptual and notational definition of the Watch State domain concept for the watch modelling language.

Concepts and symbols alone are not enough: a modelling language should also follow and enforce the rules that exist in the 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. The various domain rules implemented in the watch modelling language are presented in figure 3.

The window with “Relationships”, “Roles”, and “Objects” shows the concepts of the domain language. Because here the domain is watch the concepts are watch specific. Objects are the concepts that user can draw individually like button, display function, state.

Relationships are the connections be-

Figure 5: Example code produced from figure 1

FIGURE: 5

```
Import java.until.*;

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 ();
    }
}
```

tween objects. Like relationship 'Transition' can connect objects 'State' together. Roles are line ends (eg, arrowhead).

Therefore with combination of Object-Role-Relationship-Role-Object we can say that object 'State' (Show) has role 'From' in relationship 'Transition' [thick line] to role 'To' [arrow symbol] connecting to the object 'State' (EditHours).

In this way we can show different legal combinations preventing designers from creating illegal models (like connecting two buttons together).

The lower small window shows other constraints like in watch domain one alarm can only trigger one state. This means that designer cannot connect Alarm to trigger multiple states.

This example is again one particular domain rule. Of course it could be possible that triggering would go to different places, like alarm triggers timer and worldtime, etc.

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.

The key issue in building a code generator is how the models 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.

The general watch code generation architecture is presented in figure 4.

The state machine for each sub-application is generated from its model by inheriting the base functionality from the abstract class and extending it by loading the state machine definition data and by defining the methods that are executed during the transitions and for the display functions. These methods themselves are actually dispatchers that make the further calls to the services provided by the component library. An example of code generated for the sub-application defined in figure 1 is presented in figure 5.

Having fun with a super model

Developing an in-house, domain-specific method allows faster development, based on models of the product rather than on models of the code. The investment you need to make first is to build the domain language and related generators. This investment has been found small compared to the payback. For example, the implementation of the watch modelling language as presented above took eight days: five days to design the architecture and assemble the Java components (without any prior Java experience), two days to define the modelling language and one day to implement the code generator. This development effort also produced the first working watch model. From here on, new watch models can be implemented and tested in less than twenty minutes.

This example illustrates the most important benefit of domain-specific modelling languages: five to ten times faster development of products, leading to lower development costs and faster time to market. The key factors contributing to this are:

- The problem is solved only once at a high level of abstraction and the final code is generated straight from this solution.

- Complexity and implementation details are hidden and the familiar domain-specific terminology is emphasised, thus allowing the developer to concentrate on the required functionality, shifting the focus from the code to the design.

- Consistency of products and lower error-rates are achieved thanks to the better uniformity of the development environment and reduced switching between levels.

- The domain knowledge is made explicit for the developer team as it is captured into the modelling language and its tool support.

All these benefits have been proved by real-life cases from industry (eg, Lucent, Nokia, NASA, USAF). There is still, however, one more encouraging aspect of creating a domain-specific modelling language – it is great fun to do!

Juha-Pekka Tolvanen is CEO of MetaCase Consulting. He has acted as a consultant world-wide for method development and has published papers on this subject in several journals and in September this year gave a tutorial at CMP's Embedded Systems Conferences. Juha-Pekka Tolvanen can be reached at jpt@metacase.com.