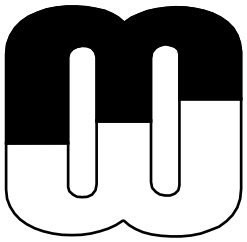


WHITE PAPER

DOMAIN-SPECIFIC MODELING WITH METAEDIT+: 10 TIMES FASTER THAN UML

MetaCase
Ylistönmäentie 31
FI–40500 Jyväskylä, Finland

Phone +358 400 648 606
Fax +358 420 648 606

E-mail: info@metacase.com
WWW: http://www.metacase.com

DOMAIN-SPECIFIC MODELING WITH METAEDIT+: 10 TIMES FASTER THAN UML

Abstract

Domain-Specific Modeling (DSM) raises the level of abstraction beyond programming by specifying the solution directly using domain concepts. The final products are 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. In this paper we describe why DSM is faster and how to build a DSM language and generator using MetaEdit+.

1 INTRODUCTION

It has been recognized for many years that there is a vital difference between an application's problem domain and its code (Jackson 1995). These are two different worlds, each with its own language, experts, ways of thinking etc. A finished application forms the intersection between these worlds. The difficult job of the software engineer is to build a bridge between these worlds, at the same time as solving problems in both worlds. In this article we will look at how that job can be made easier by focusing more on the problem domain (or just "domain" for brevity).

Figure 1 shows four different ways in which this bridge building has occurred: how developers have moved from an initial domain idea to a finished product. In all cases the problem has initially been expressed in the terms of the domain, and had to be solved in those terms: what had to be done rather than how it would be done. In the first two cases this solution was then mapped to the world of the implementation platform and implemented there. The invention of more powerful chips, assemblers, and compilers steadily narrowed the domain-product gap from the right inwards, allowing the final 'hand-made' artifacts of the designer to be at a higher and higher level of abstraction. These artifacts could then automatically be transformed into the finished product.

The introduction of modeling languages such as UML changed surprisingly little (Kelly & Tolvanen 2008, Greenfield et al. 2004). The problem must still be solved first in domain terms with little or no tool support. This is because UML does not relate directly to the application domain (e.g. mobile phones, car infotainment systems, medical devices, point-of-sale systems etc.) but to the implementation, i.e. it visualizes the code. Therefore, the domain solution must be mapped to the core UML models representing the implementation in code, from which in general a relatively small percentage of the finished code can be automatically generated. The developer must then

fill in the method bodies by hand: the largest part of the implementation. Thus the developer still has to solve the problem twice: once in domain terms (often in his head and on the backs of envelopes), and once in code terms. He still has to perform the mapping from the domain solution to the code solution, now with an extra 'stepping stone' of UML — planted firmly near the code side. In fact, in cases where UML models do not provide adequate mappings to code, the developer must now solve the problem three times!

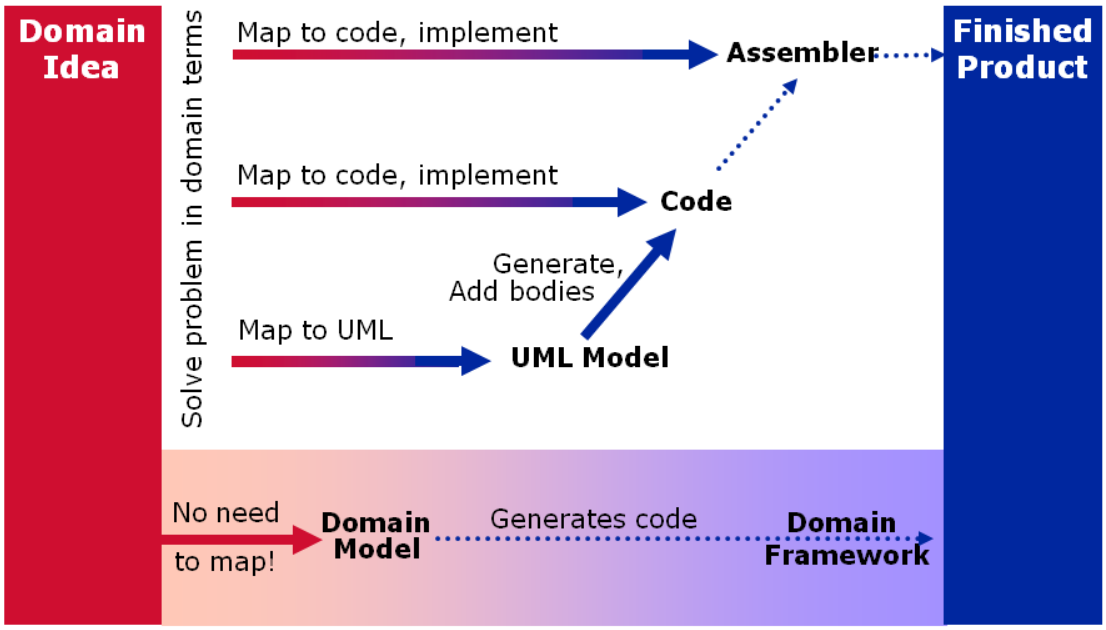


Figure 1. Moving from domain idea to finished product

Primarily from industry, there have been moves towards a way of building software that removes this resource-intensive and error-prone mapping and double (or even triple) problem solving. The ideal is that a developer would be able to **develop the solution once only**, as a model in domain terms, from which the finished product is automatically generated. Such an approach has already been seen to work very effectively in a range of situations, most notably in embedded systems and product families. In this paper we first describe how domain-specific methods work and then look at the main experiences from the users of these methods.

To achieve our ideal in a given domain, we must provide three things: a modeling language specific to that domain, a tool for building models in that language, and automatic code generation from models in that language to appropriate implementation code. In this paper we will discuss the support offered by MetaEdit+ in creating such languages and tools for visual modeling and code generation.

2 DOMAIN-SPECIFIC METHODS

In DSM, the model elements represent things in the domain world, not the code world. The modeling language follows the domain abstractions and semantics, allowing modelers to perceive themselves as working directly with domain concepts. The rules of the domain can be included into the language as constraints, ideally making it impossible to specify illegal or unwanted design models (Pohjonen & Kelly 2002).

Let's take a small example. Suppose you manufacture digital wristwatches and your developers make 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, states and user's actions. DSM applies these very same concepts directly in the modeling language. An example of a model in such a language is shown in Figure 2. The model represents the time setting feature: the actions a user can make by pressing buttons, the display elements blinking, and the actions changing the time.

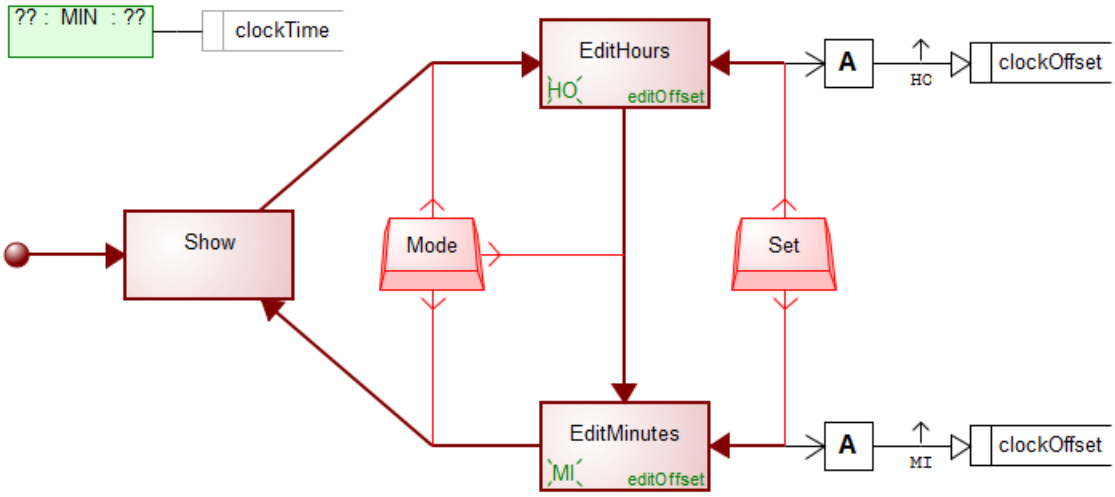
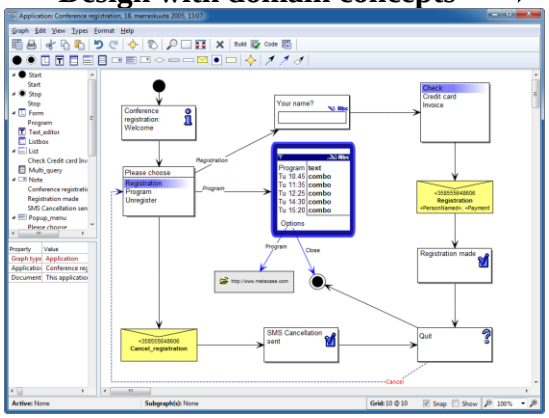


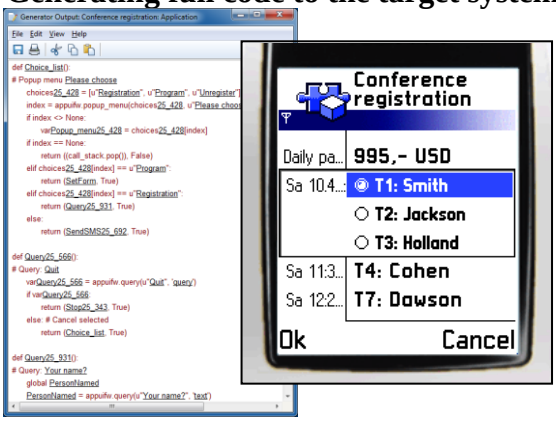
Figure 2. Modeling time setting feature

As Figure 2 shows all the relevant parts of the time setting feature are captured in the model, enabling the complete code to be generated. The language also directs developers to concentrate on the required aspects of the watch (by hiding irrelevant parts). A complete example of this modeling language with 100% code generators is included in the free MetaEdit+ evaluation version, available for download from http://www.metacase.com.

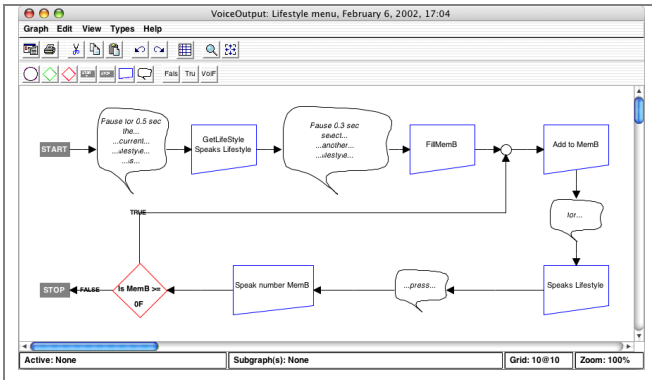
Every domain is different, and so every DSM example is different too. Below you will see other examples of DSM in different industry sectors, with an example model and full code generated directly from the model.

Design with domain concepts ➡ **Generating full code to the target system**

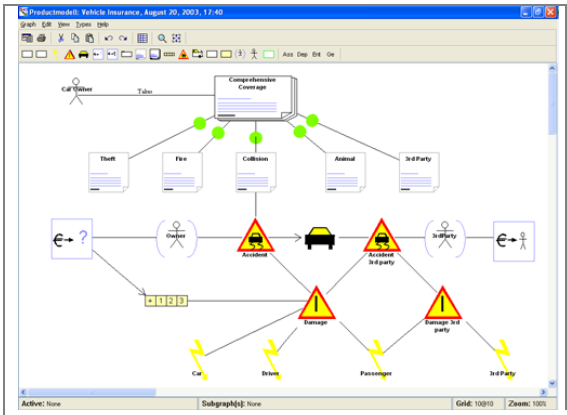


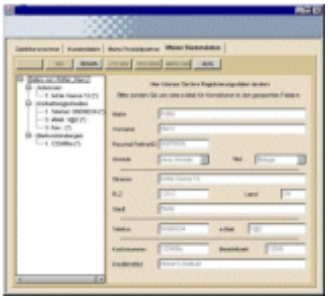


This example illustrates application development for mobile phones. DSM uses the widgets and services of the phone as modeling concepts, following the phone's UI programming model. The generator produces full code calling the phone's platform services and executes the result in an emulator or in the target device.



This example shows voice menu system development for an 8-bit microcontroller. The DSM shows the flow-like execution of the menu system. The generator produces assembler with the necessary functionality for memory addressing, calculation, etc.





This example shows specifying insurance and financial products. An insurance expert, non-programmer, draws models to define different insurance products, and then generators produce the required insurance code for a J2EE website.

Figure 3. Examples of DSM in different industry domains.

3 GETTING STARTED WITH DSM

To get the DSM benefits of improved productivity, quality and complexity hiding, we need to specify how the domain-specific language and generator should work (Pohjonen & Kelly 2002). In the past, we would also have needed to implement the supporting tool set. This was one of the main reasons holding DSM back: after all, implementing modeling tools is hardly a core competence for most organizations. Today, the work needed is reduced to just defining the language and generators, since MetaEdit+ provides the rest: diagramming editors, browsers, generators, multi-user and platform support etc.

To implement DSM you need an expert developer in that domain, or a small team of them. This would typically be 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.

Figure 4 shows the elements that must be made by the expert, along with how they will be used by the normal user. It is important to note that only the expert has to bridge the gap from the domain to the finished product. After that is done, other developers are freed of that burden and can concentrate on finding a solution in the domain.

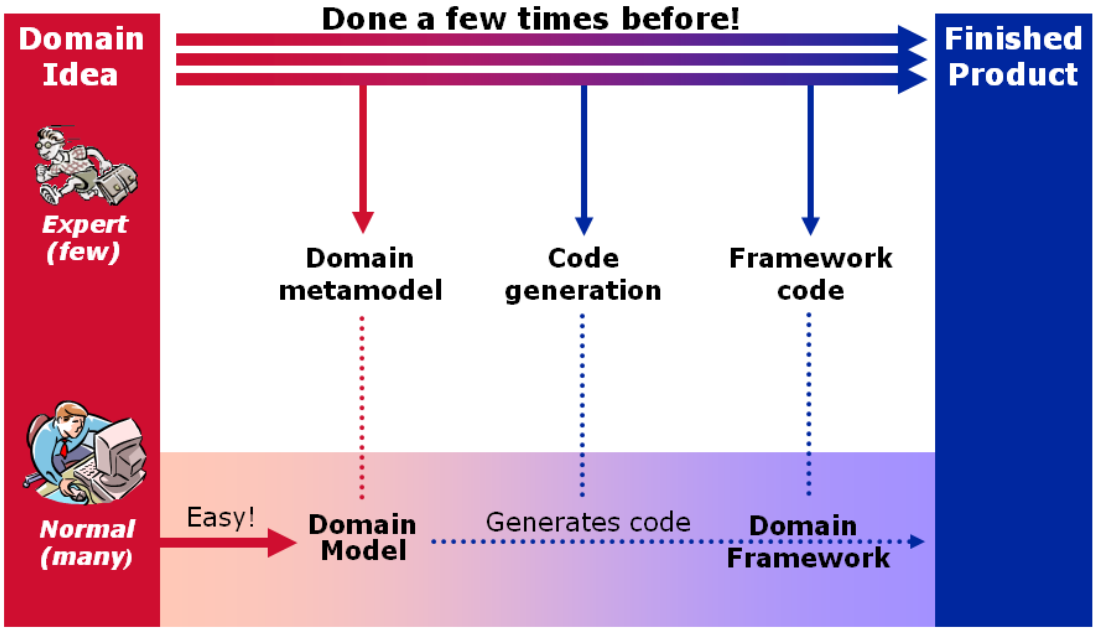


Figure 4. Leveraging experts to enable others

3.1 Assembling the domain framework

A domain framework provides the interface between the generated code and the underlying platform. In some cases, no extra framework code is needed: generated code can directly call the platform components and their services are enough. Often, though, it is good to define some extra framework utility code or components to make code

generation easier. Such components may already exist from earlier development efforts and products. Further developing these pieces of code into true framework code is a relatively easy task for the expert, requiring only normal developer programming tools and skills.

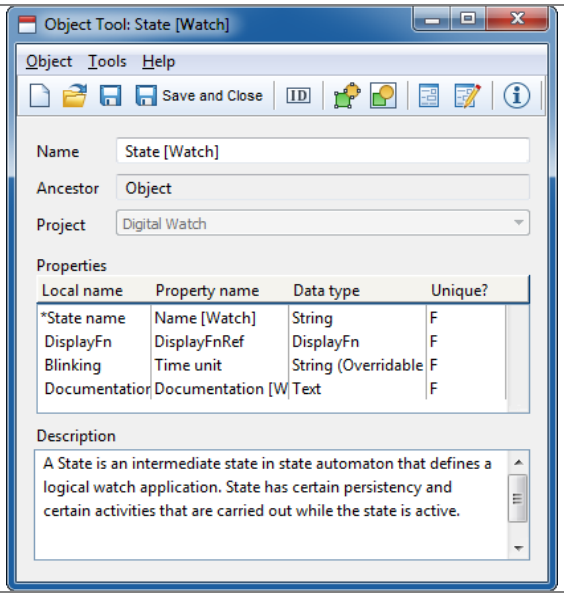
Whilst the component library here is thus nothing new, the fact that it will be intrinsically part of the development process ensures that the components there will actually be used (Greenfield et al. 2004, Kelly & Tolvanen 2008).

3.2 Developing the modeling language

Defining the modeling language deals with identifying the modeling concepts, the rules that constrain the use of language and enforce correctness of models, and the notation used to present these in models. These are usually best found from the domain terminology, system architecture, existing system descriptions, and component services.

For the language implementation, MetaEdit+ provides a metamodeling tool suite for entering the modeling concepts, their properties, associated rules and symbols (see figures below). Alternatively you may specify the metamodel using graphical metamodeling languages in MetaEdit+. The language definition is stored as a metamodel in the MetaEdit+ repository allowing future modifications, which reflect automatically to models and generators. The metamodel elements shown in the figures define parts of the watch-specific modeling language (see Figure 2).

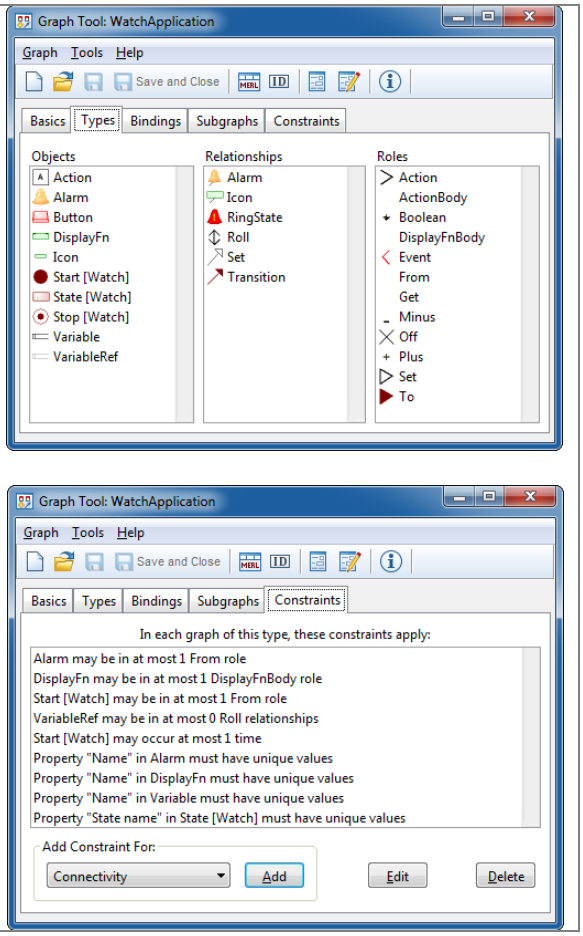
1) Define domain concepts
A DSM language should apply concepts that map accurately to the domain semantics. Using metamodeling tools you enter each domain concept and define its properties: what information can be stored with it. Modeling concepts required in software production (e.g. links to components) can also be added. The example shows the concept of watch state and its properties, such as a display function and blinking widget. An example of an instance of this concept is the 'Edit hours' state in Figure 2.



2) Choose domain rules

A DSM language should follow the rules as they exist in the domain. Once defined, the language (enacted by the supporting tool) guarantees that all developers use and follow the same domain rules. These rules are of different kinds and typically relate to connections between concepts, layering models, reusing designs, etc.

The figures show the different modeling concepts and a set of constraints, e.g. that only one 'From' role may leave each 'Start' object.



3) Draw notational symbols

A visual modeling language requires symbol definitions. The watch model in Figure 2 is one example. The notation should illustrate as well as possible the corresponding domain concepts' natural "visualization". End-users are often the best people to invent these symbols. The figure shows the symbol definition for the watch state; its shape, size, color, property values to be shown, etc.

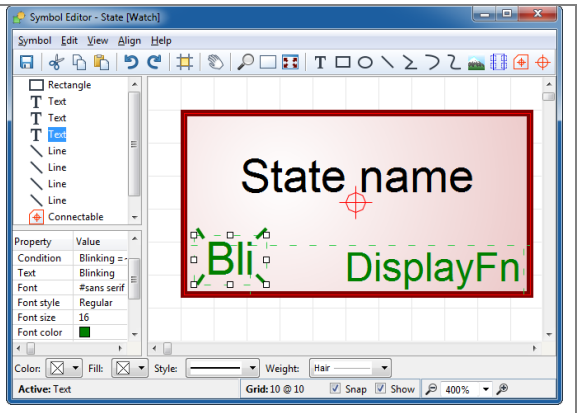


Figure 5. Steps for defining the domain metamodel

As the above example shows, good tools make DSM creation easy. With MetaEdit+ an expert can define the language (or just part of it) and instantly test it by making an example model. The expert can then concentrate on the challenge of developing the method. MetaEdit+ automatically provides the finished modeling and code generation environment with its editors, browsers, multi-user support etc. MetaEdit+ also delivers the language immediately to the developers and updates existing models instantly to reflect the changes.

3.3 Developing the generator

Finally, we want to close the gap between the model and code world by defining the code generator. The generator specifies how information is extracted from the models and transformed into code. This code will be linked with the framework and compiled to a finished executable without any additional manual effort (Kelly & Tolvanen 2008). The generated code is thus simply an intermediate by-product on the way to the finished product, like .o files in C compilation.

The common elements of all products made with this DSM language have already been abstracted out from what needs to be modeled per application, into the domain framework code shared by all applications. These common elements can range in size from whole components down to individual groups of programming language statements that occur commonly in code in this domain. The modeling language allows the capture of all the remaining information necessary to build a full product, hence all that is needed for fully working code can be found from the models.

The key issue in building a code generator is how the models concepts map to code. The domain framework makes this task easier by raising the level of abstraction on the code side. In the simplest cases, each modeling symbol produces certain fixed code, including the values entered into the symbol as arguments. The generator can also generate different code depending on the values in the symbol, the relationships it has with other symbols, or other information in the model.

MetaEdit+ provides the necessary functionality for creating and debugging generation scripts, and it guides the expert to access the concepts in the metamodel. Generators can be defined using the Generator Editor (see Figure 6), or alternatively you can use own generator and integrate it with MetaEdit+.

Make the generators

Generators insulate the modelers from implementation aspects: programming details, architecture, component use and even optimization and other compiler flags.

The example on the right shows part of the code generation definition: how watch models are used to generate 100% of the required code in Java. As an expert has specified the generator, it produces products with better quality than could be achieved by normal developers by hand.

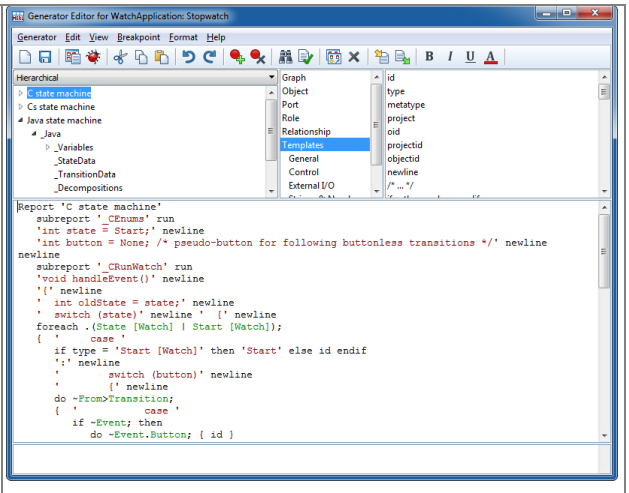


Figure 6. Defining generators

4 CONCLUSION

Domain-Specific Modeling allows faster development, based on models of the product rather than on models of the code. Industrial experiences of DSM show major improvements in productivity, lower development costs and better quality. For example, companies like EADS (MetaCase 2012), Panasonic (Safa 2007), Polar (Kärnä et al. 2009) and Elektrobit (Puolitaival et al. 2011) state that with DSM they can develop products up to 10 times faster. The key factors contributing to this are:

- The problem is solved only once at a high level of abstraction and the final code, configurations, tests, analysis etc. is generated straight from this solution.
- The focus of developers shifts from the code to the design, the problem itself. Complexity and implementation details can be hidden, and already familiar terminology is emphasized.
- Consistency of products and lower error-rates are achieved thanks to the better uniformity of the development environment and reduced switching between the levels of design and implementation.
- The domain knowledge is made explicit for the development team, being captured in the modeling language and its tool support.

DSM also provides a better role for expert developers. Rather than have them help others in fire-fighting problems in basic development tasks, or move them to a new area and lose their expertise, they can be put to work on a problem they will find interesting and rewarding, and which will best leverage their expertise.

Providing tool support for satisfactory modeling and generation has previously required several man-years of work. MetaEdit+ reduces the time needed down to the order of days or weeks. MetaEdit+ is tried and proven technology. It has been used to build hundreds of Domain-Specific Modeling languages providing a robust, higher-level, higher-quality way to build software.

REFERENCES

Greenfield, J., Short, K., Cook, S., Kent, S., Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley, 2004.

Jackson, M.A., *Software Requirement & Specifications* A lexicon of practice, principles and prejudices Addison Wesley, ACM Press, 1995.

Kelly, S., Tolvanen, J-P., Domain-Specific Modeling: Enabling full code generation, Wiley, 2008.

Kärnä, J., Tolvanen, J.-P, Kelly, S. Evaluating the use of domain-specific modeling in practice. Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling, 2009.

MetaCase, EADS Case Study, http://www.metacase.com/papers/

Pohjonen, R., Kelly, S., "Domain-Specific Modeling," Dr. Dobbs Journal, August 2002.

Puolitaival, O.-P., Kanstrén, T., Rytky, V.-M, Saarela, A. Utilizing Domain-Specific Modelling for Software Testing, The 3rd International Conference on Advances in System Testing and Validation Lifecycle, October 23-29, 2011, Barcelona, Spain, 2011.

Safa, L. The making of user-interface designer a proprietary DSM tool. In 7th OOPSLA workshop on domain-specific modeling (DSM), 2007