Building a Flexible Software Factory Using Partial Domain Specific Models

Jos Warmer[1], Anneke Kleppe[2][3]

[1]Ordina SI&D, The Netherlands
Jos.Warmer@ordina.nl
[2]University Twente, Netherlands
a.kleppe@utwente.nl

Abstract. This paper describes some experiences in building a software factory by defining multiple small domain specific languages (DSLs) and having multiple small models per DSL. This is in high contrast with traditional approaches using monolithic models, e.g. written in UML. In our approach, models behave like source code to a large extend, leading to an easy way to manage the model(s) of large systems.

1 Introduction

A new trend in software development is to use model driven techniques to develop software systems. Domain specific models (DSMs), domain specific languages (DSLs), and the transformations from the DSMs to code need to be carefully designed to make them really useable.

An obvious observation is that one single model (in a single file or single repository) will not suffice for describing a complete application. Such a model would be too large to handle; it would be unreadable and thus not understandable. Although obvious, this is something that has not been acknowledged in the modelling world. Companies that apply model driven development on a large scale are having problems in managing models that are sometimes over 100 MB size. We therefore argue for building smaller, partial models, each of which is part of a complete model. This is much like the way a code file for a class is part of the complete source code for the application. Each partial model may be written in either the same or a different DSL, thus using the advantage of the fact that a DSL is designed to solve one specific part of a problem as good as possible.

In this paper we show how partial models can be used to build large, complex applications. We also show the consequences of this approach on the DSL definitions and their accompanying model-to-code transformations. We will also show how managing models for large applications is simplified by using partial models.

This paper is based on the industrial experience of one of the authors with building the SMART-Microsoft Software Factory at Ordina, a model driven development software factory using the Microsoft DSL Tools. At the time of writing this software factory included four different DSLs. Typically several dozens of DSMs are created in a project that utilizes this software factory. Although the experience was gained using the Microsoft DSL Tools, the approach can be applied in other environments (like e.g. Eclipse GMF).

This paper is structured as follows. Section 2 explains in short the development process when using a model driven software factory. Section 3 introduces the concept of partial mod-

3. The author is employed in the GRASLAND project funded by the Dutch NWO (project number 612.063.408).

els, and section 4 explains our approach to references between partial models. Section 5 explains different forms of code generation from partial models.

2 The Software Development Process

The traditional development process, not using models, DSLs, or model transformations, can (simplified) be described as follows. Decide on the architecture of the application (1). Design the application (2). Write the code, compile it, and link it (3). Run the application (4).

The model driven software factory process, as introduced in [GSCK04], using DSLs and model transformations, works in a different way. First, the software factory itself is designed as follows:

1. Decide on the architecture of the application.
2. Design the DSLs for this architecture
3. Write the transformations for these DSLs

The system developer does not need to determine the architecture any more, but starts directly with modelling the application:

1. Model the application.
2. Transform the models.
3. Write additional code (if required).
4. Compile and link the code, and run the application

This process is often done iteratively, meaning that after running the application in step 4 you go back to step 1 and start modeling the next part of the application. The development of the software factory is also done iteratively, but in the context of this paper that is not relevant. Also note that a software factory is more than just a collection of DSLs, however this paper focuses on the DSL aspect, and what's more we focus on the first part of the process: how to build a collection of DSLs and their transformations.

3 Developing a Flexible Software Factory

The first step when developing a model driven software factory, is to determine the architecture of the applications that you are going to build with the software factory. Is it, for instance, a web-based, administrative application or is it a process control system? The answer to this question determines the architecture. From the architecture we derive which DSLs are to be defined for modelling the application.

The SMART-Microsoft Software Factory is targeting web-based, administrative applications, of which the architecture is shown in Figure 1. Based on this architecture we have defined four different DSLs, each of which corresponds to a part of the architecture. We recognise the following domains: the Web Scenario DSL for the Presentation layer, the Business Class DSL for the Business classes, the Services DSL for the Service Interface and Business Processes, and the Data Contract DSL for the Data Contract. There is no DSL corresponding to the Data layer, because this layer is completely generated from the Business Class DSL. A developer who wants to build a compete system will use all DSLs together.

The different DSL are mostly independent, therefore it is possible to use a subset of the DSLs provided. We can also combine the DSLs in a different way. For example, we are planning to develop a DSL for building Windows user interfaces, which can then be used instead of the current Web Scenario DSL. This allows us to flexibly evolve the software factory.

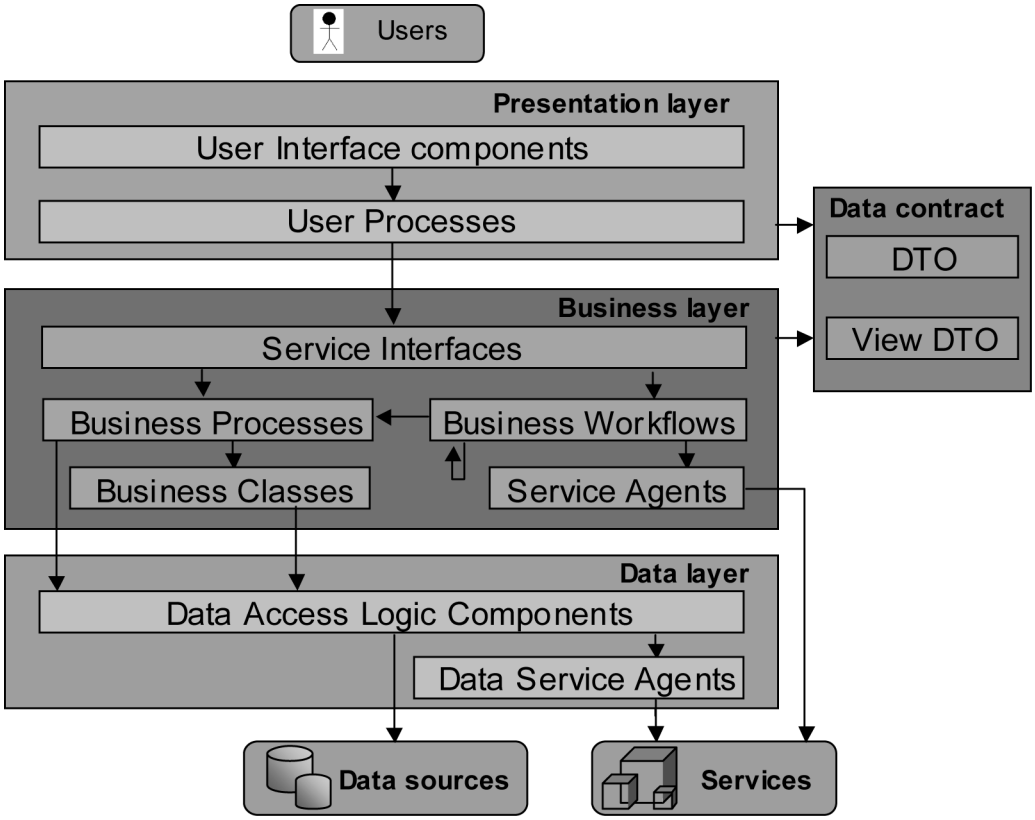

Fig. 1 The Web Application Service Architecture

3.1 Goals for Domain Specific Languages

1. A model is always executable in the sense that every aspect of a model is used to generate code. We do not consider models used purely for design or documentation, these can be built effectively with UML or tools like Visio.
2. A concept only becomes part of a DSL if it is easier or less work to model it than to code it. This keeps the number of concepts small and ensures that the DSL is very productive.
3. Models (or better said the code generated from the models) are meant to be extended by code.

3.2 Introducing Partial Models

When using the software factory to build an application, a developer determines the number and kind of DSMs that need to be built. One possibility, which we have seen used at several places, is to create one DSM per DSL. This would mean that we have four DSMs for each application. Still, for a large application this still does not scale up. For instance, if we have one DSM for the complete Web Scenario Domain, this will become an incredibly large model for any real application. The model would contain many Web Scenario elements, which each consists of a set of Web Pages and Actions. A model of such size is not readable, and certainly not understandable.

Working with one large model also introduces many practical problems relating to managing such a model in a multi-user environment. Experience with traditional UML tools has

learned us that this problem has not been solved by any of them. Even when a tool allows multiple persons to work simultaneously on a model, the model must usually be divided beforehand in non-overlapping slices and the developers must be very disciplined while working with the tools.

The solution to this problem that we present here is to use multiple DSMs per DSL. We call these models *partial models*, because they do not represent the complete system. Each partial DSM is stand alone and can be used in isolation. In the case of the Web Scenario DSL, the DSL has been designed such that each DSM contains not more than one Web Scenario. If an application needs e.g. twenty Web Scenarios, twenty Web Scenario DSMs will be created. As a direct consequence of this choice each partial DSM has some unique and useful properties:

- One partial DSM can be created and edited stand alone by one user.
- The partial DSM is the unit of version control, and when the DSM is stored on file, ordinary version control systems provide ample possibilities for version control.

Our approach fits very well with the structuring of the Microsoft DSL Tools that we have been using, in which one model is stored in one file. Also, in the Microsoft DSL Tools one model is identical to one diagram, and should therefore remain small. In the remainder of this paper all DSMs are partial models, the DSLs are designed to support this.

4 Combining Partial DSMs using References

Allowing partial DSMs has direct consequences for the way that a DSL is defined. One such consequence is that we need a way to define references between DSMs. This section describes the ins and outs of references.

4.1 References between Partial DSMs

A model element from a partial DSM may be referenced in another partial DSM just like classes and their operations may be referenced in a code file. To ensure that a DSM remains a stand alone artifact, references are always modelled explicitly and are always *by name*. There are no hard links between different DSMs, otherwise we would end up with one monolithic model again. To accommodate this we have introduced a metatype *Reference to ModelElement* for each modelelement that we want to refer to in each of our DSLs. This metaclass may be subclassed to create a reference to a particular type of modelelement. Thus, a model element in a DSM may be of type *Reference to BusinessClassDto*, holding the name (or path) of a business class in another DSM.

References may link DSMs written in the same DSL, e.g. a *Reference to WebScenario* in a Web Scenario DSM, or they may link DSMs written in different DSLs, e.g. a *Reference to BusinessClassDto* in a Web Scenario DSM, that refers to a modelelement in a Data Contract DSM. An example of the first can be found in DSM 1 in Figure 2, an example of the second can be found in DSM 2.

4.2 Checking References

In a complete application the references within the DSMs should all be valid, e.g. the referred *WebScenario* in Figure 2 must be defined in another DSM. For this purpose we have developed inter-DSM validation support. With one button, a user can do a cross-check on all references to check whether the referred elements exist. This validation is based on a small run-time component, which is populated from the DSMs in the developers workspace. This component is similar to a symbol table in a compiler and only holds the minimum information needed for validation purposes.

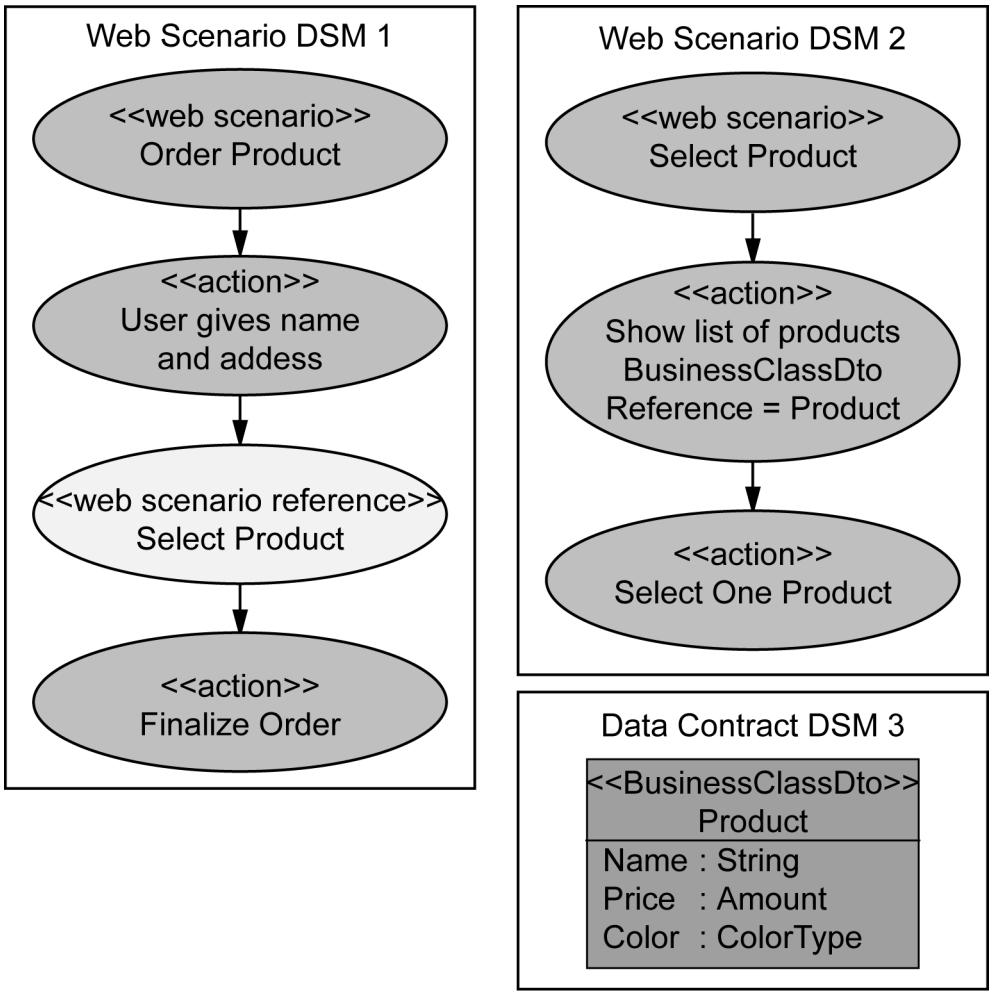

Fig. 2 Example of references between partial models

Note that a single DSM is still valid if a reference does not exist, but the collection of DSMs is not complete. The DSM with the unresolved reference can still be edited, checked in, and its model elements can be referred to by other DSMs, etc.

4.3 Dealing with Changes in References

A change in the name of a referred model element is allowed, but will make existing reference(s) dangling. This is an inherent feature, following directly from the way DSLs are designed. Tool support for coping with this kind of changes is not within the scope of language definition, instead it should be provided by the IDE. There are various options for dealing with dangling references:

- *No support*: the inter-DSM validation will result in an error message and the developer has to "repair" the dangling reference.
- *Refactoring support*: the user may explicitly perform a name change of the referred model element as a refactoring. The IDE will find all references, and change them to refer to the new name.
- *Automatic support*: when the user changes the name of a referred element, all references will change automatically.

Having no support at all does work, but is cumbersome. Automatic support has the problem that the developer does not know where automatic changes take place and he might therefore encounter unexpected results. In the Plato model driven environment that we have built in the

past, we found that automatic changes also results in the need to re-test the whole system, because the changes were not controlled.

The best option seems to be refactoring support. Note that in this case renaming a model element works exactly as renaming a class in C# or Java code. Either the user changes the name, which results in dangling references, or the user requests an explicit refactoring and is offered the possibility to review the resulting changes and apply them. In the SMART-Microsoft Software Factory we have chosen the option of using explicit refactoring. The run-time component for cross-reference validation holds all the information needed to execute this.

In both automatic and refactoring support the following problem may occur. Specially in large projects, there will be many dozens of DSMs, and each DSM can be edited simultaneously by a different user. To allow for automatic change propagation or refactoring the user performing the change needs to have change control over all affected DSMs. Because we do not have a model merging feature available in the Microsoft DSL Tools, this problem cannot currently be solved.

5 Code Generation

In this section we explain different forms of code generation from partial models. As our models are meant to generate code, this is an essential part of the DSL definition. We do not use our models for documentation purposes only.

5.1 Different Types of Generation

In model driven development [MSUW04, Fra03, KWB03] multiple layers of abstraction are used. Inhabitants of the lowest layer are called code, inhabitants of all layers above the lowest are called models. There is no restriction on the number of layers that may be used, as shown in [GSCK04].

The definition of a DSL includes the code generation for that DSL. Interestingly, it is also possible to generate another model instead of code, thus making use of multiple layers of abstraction. For DSLs defined at a higher level of abstraction, it is often more easy to generate a lower level DSM than code, because the generation target itself is at a higher level of abstraction. Therefore we distinguish the following two types of generation.

DSM to Code Generation. The first type of generation is code generation from a DSM. This is the most common way of generation. Template languages like T4 for Visual Studio or JET and Velocity for Java are often used for this purpose.

DSM to DSM Generation. The second type of generation is to generate another model from a DSM. This is possible when a DSM can be completely derived from another (higher level) DSM. Often the generated DSM takes the form of a partial model. The developer can add (by hand) other partial DSMs that refer to the generated DSM, thus extending or completing the model.

5.2 Linking Partial Models or Linking Partial Code?

Another distinction that needs to be made is the moment when the partial descriptions of the application are brought together. There are two possibilities:

1. Link all partial models together to form the complete model. Transform the complete model into code.
2. Transform a single partial model into (partial) code. Link the generated code.

Within the SMART-Microsoft Software Factory we have chosen to use option 2. The code is generated immediately (and automatically) whenever a DSM is saved. Our experience is that generating everything in one step from a complete model can become very time consuming, resulting in long waiting times to conclude the code generation process. Using option 2, we can perform the transformation process incrementally, re-generating only those parts that have been changed. Also, option 2 fits much better in our philosophy that we do not need a complete model at any point in time.

However, option 2 is not always feasible. When the information in two or more partial models needs to be transformed into a single code file, only option 1 will suffice. In our case we generate C# code, which offers the notion of partial classes, which are used extensively in the SMART-Microsoft Software Factory.

5.3 Regeneration and Manual Additions

When something - either code or another DSM - is generated from a DSM we need to be able to do at least two actions:

- Regenerate whenever the source DSM changes.
- Manually add something to the generated code or generated DSM.

Besides, we must at any time be able to use the two options independently. That is, when we have manually added code or DSMs, we must still be able to regenerate the generated parts while maintaining the handwritten code or DSMs. This is implemented as follows.

Regeneration of Code. When we generate C# code, partial classes and abstract / virtual methods are used to enable the user to add code without touching the file that was generated. This allows full regeneration without disturbing the handwritten code. For other types of artifacts, the situation is more complex, often handwritten code has to be in the same file as generated code (e.g. in XML configuration files). The handwritten code is then marked as a guarded block and retained when the file is regenerated.

Regeneration of DSM. When we generate a DSM from a higher level DSM, we use the same approach as when generating C# code. One partial DSM (in one file) is generated, and this file remains untouched by developers. Handwritten additions must be modelled in separate partial DSMs. *Reference* elements (see 4.1) may be used in the handwritten DSM to refer to model elements in the generated DSM.

6 Other Views on Modeling

In the UML world the term model is often used rather loosely both for a diagram, and for the underlying collection of interrelated modelelements. However, according to the UML language definition, there can be only one model - consisting of multiple diagrams - and each diagram is a specific view on part of the underlying model. The UML offers no way to define references between different models, it assumes that you always work with one (large) model.

In the agile modeling community there is a tendency to create small models. However, these models are typically used for documentation and design, but are rarely used for code generation. The difference between this type of models and the ones presented here is that the agile models usually cannot be processed automatically to generate working code. Although human readers might recognise references between agile models, tools will not. Also, what is considered to be a set of small models, often is a set of diagrams in one UML model.

The partial DSM as described in this paper always constitutes an executable model. Within a software development project these models have exactly the same status as source code.

They are the source from which the final system is generated. Before completely building the system all references in both the source code and the (partial) models must be resolved.

7 Conclusion

We have described the development of a model driven software factory using multiple DSLs. The approach takes a non-traditional view to modelling. Instead of having one monolithic model we use many smaller models of many different types. These models are called partial models or partial DSMs. In our approach one partial DSM has the same characteristics as one source code file, which clarifies many things and leads to a different way of thinking about models. The following characteristics of partial DSMs are identical to source code files.

- Storing a DSM in a file
- Code generation per DSM
- Version control per DSM
- References between DSMs always by name
- Refactoring in the IDE if requested by user
- Intellisense / code completion for *Reference* elements

While building the SMART-Microsoft Software Factory, we found more and more advantages of our approach. Although not discussed in this paper, each DSL can be used in isolation of other DSLs. This opens up possibilities to use a subset of the DSLs whenever applicable or, for example, replace one DSL by another one in the software factory. We also see opportunities to reuse both DSLs and DSMs in a relatively easy way.

We view the approach to building a software factory using DSLs as an approach to MDA. Although MDA is often related to UML, this is not mandatory. Using DSLs fits into this approach as well. Also, we deal with model-to-model transformations as well, although we have no fixed number of levels like the PIM - PSM - Code levels in MDA.

The SMART-Microsoft Software Factory also has strong connections with the idea of developing product lines [CE00]. The software factory is a product line for administrative web applications according to a defined architecture. We have ensured flexibility for building other product lines by keeping the DSLs as independent entities. Apart from the DSLs, a Software Factory also includes other things, like a specialized process and others. This paper focuses on the DSL aspect only.

References

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[Fra03] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories, Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[MSUW04] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled, Principles of Model_Driven Architecture*. Addison-Wesley, 2004.

[SMART06] http://www.ordinasoftwarefactory.nl/Default.asp/id,285/index.htm: SMART-Microsoft Website.