

Using a Domain-Specific Language and custom tools to model a multi-tier service-oriented application—experiences and challenges

Marek Vokáč¹ and Jens M. Glattetre²

¹ Simula Research Laboratory, P.O.Box 134, 1325 Lysaker, Norway,
marekv@simula.no,

WWW home page: <http://www.simula.no/>

² SuperOffice ASA / ICT Norway^{**}, Drammensveien 211, 0212 Oslo, Norway,
jens.2005@superoffice.com,

WWW home page: <http://www.superoffice.com>

Abstract. A commercial Customer Relationship Management application of approx. 1.5 MLOC of C++ code is being reimplemented, in stages, as a service-oriented, multi-tier application in C# on Microsoft .NET. We have chosen to use a domain-specific language both to model the external service-oriented interfaces, and to manage the transition to the internal, object-oriented implementation. Generic UML constructs such as class diagrams do not capture enough semantics to model these concepts. By defining a UML Profile that incorporates the concepts we wish to model, we have in effect created a Domain-Specific Language for our application. The models are edited using Rational XDE, but we have substituted our own code generator. This generator is a relatively generic text-substitution engine, which takes a template text and performs substitutions based on the model. The generator uses reflection to convert the UML and Profile concepts into substitution tags, which are in turn used in the template text. In this way, we can translate the semantics of the model into executable code, WSDL or other formats in a flexible way. We have successfully used this approach on a prototype scale, and are now transitioning to full-scale development.

1 Introduction and Problem Definition

Many companies are faced with a transition from an object-oriented programming model that implements a rich client, to a service-oriented architecture and an increasing emphasis on Web-based clients. A service-oriented architecture (SOA) requires application components to be structured in a way that is different from traditional, in-process object-oriented models.

Service-oriented architectures also prescribe a different approach than that of earlier Remote Object or Remote Procedure Call architectures, such as CORBA

^{**} Supported by the Norwegian Research Council ICT Programme “FAMILIER”, and “FAMILIES”, ITEA project ip02009 of the EU Eureka Σ! 2023 Programme.

or DCOM. One of the main tenets of SOA is to make boundaries between systems and services explicit, to promote interoperability and to encourage their proper use. Remote invocation is inherently orders of magnitude more expensive than local execution, and the architecture and granularity of the interfaces and messages must reflect this.

At the same time, the actual business and data access logic is generally implemented using object-oriented languages such as Java or C#. It may be desirable to reuse existing code, which typically represents a significant investment by the organization.

A coherent SOA requires modelling—it is not enough to simply go ahead and define services freely; this will result in a large set of disparate services that do not work well together. The design of a large object-oriented implementation will also benefit from modelling.

We are thus faced not only with the need to separately model a coherent SOA and an object-oriented implementation, but also to model the transition between the two—the connection between interfaces and their implementation. Ultimately, this set of models should result in the generation of executable code (including service definitions in WSDL or other appropriate description languages, and at least the skeletons of the implementations), as it will otherwise be hard to realize benefits that justify the investment in the modelling effort.

This experience report is written from the perspective of an industrial development project. We are looking for workable, pragmatic solutions that can be used in a full scale development project at the present time. Our approach to related work, tools and methods reflects this perspective.

We have conducted interviews with architects and developers in our organization to extract the knowledge presented here. The developers' experience with modelling ranges from minimal to extensive (more than 5 years), and their time with the company from 7 years down to just a few months. We are therefore able to present experience from a number of different viewpoints.

The rest of this experience report is organized as follows: section 2 summarizes the different approaches and tools we have considered. In section 3 we present our chosen solution, in the form of a Domain-Specific Language and its related Code Generator. Section 4 reports on our experience from using this approach for a modest, yet commercial and marketable, Collaborative CRM product. Finally, section 5 concludes and outlines our future work.

2 Tools and approaches to modelling SOA and OO

Several approaches to modelling at roughly the level needed for a Service-Oriented Architecture and its object-oriented implementation have been presented. A lot of effort has been spent on the Unified Modelling Language (UML), and Model-Driven Architecture (MDA) has been pushed as a concept and a trademark of the Object Management Group (OMG).

From the standpoint of a practitioner facing a choice of approach and a deadline, good tool support is perhaps the single most important factor. Manual,

paper-based modelling, or the use of prototype academic tools is not a sufficient basis for an industrial project of significant size and complexity. For instance, the MDA specification published by OMG (1) contains a bare three pages on the subject of transformations from the platform-independent to the platform-specific models.

Czarnecki (2) has proposed a taxonomy of transformations, as well as an overview of existing techniques. For our needs, a template-based approach seemed to provide the optimum balance between flexibility, power and readability (2, Ch. 3.1.2).

Our organization is a fairly small one, with six developers being considered a fairly large team. Contrary to the practice in many large (especially North American) companies of having strictly defined roles—such as architect, designer, developer, tester—most of our developers at one time or another assume almost every role, according to the stage of the development process and personal competence. This also influences our approach to modelling, since we do not have a division between modellers and implementers, or between domain and application engineers (3, Ch. 2).

Our evaluation of the state of the art therefore focused on available tools, either released or in a late beta stage, rather than on academic publications and methods therein. In practice, this restricted our choices to UML-based tools such as Rational XDE (4), Telelogic Tau (5), Borland Together (6), and Microsoft Visio (7). A further, important constraint is that our future development will be on the Microsoft .NET platform, using the C# language and Microsoft Visual Studio as the development environment.

A tempting alternative was to use tools designed for Software Factories (8), an approach where separate Domain-Specific Languages are used for different viewpoints within the total model. Transformations from model/viewpoint to C# code, SQL DDL or other artefacts can then be defined.

However, our search did not turn up any tools that we considered to be sufficiently advanced, robust and scalable to support the kind of modelling we wished to undertake. Microsoft's initiative on Software Factories and extensible modelers is interesting (9), but it is still at an early stage and not suitable for production.

Our need was (and still is) for a tool that we can use to span from a data dictionary, via simple object-oriented counterparts to relational tables, through composition and business logic up to a service-oriented set of interfaces that are not merely advanced CRUD operations; and to be able to generate the interfaces, descriptions and skeletons needed for both local and remote (e.g., via Web Services) invocations of the interfaces. It follows that reliance on the tools' built-in code generators would be too restrictive, as the transformations between the different viewpoints are not trivial.

It bears repetition that good tool support is absolutely essential in an industrial project; otherwise the model will quickly turn into more bureaucracy than help. Certainly, if the development process model uses concepts from the Agile/XP domain, with multiple, short iterations, we must expect the models

and the transformations to change. An inflexible tool would push the process towards a more strict waterfall pattern, which we do not desire.

3 A Domain-Specific Language and Code Generator

While UML has become a *de facto* standard for modelling OO software, the generic UML constructs such as classes, class diagrams and association have relatively low semantic content. Simultaneously, there are constraints on what one can model; for instance, an association cannot be set up between a specific attribute in one class and an attribute in another class.

For modelling an SOA, we need a modelling language that can capture concepts such as a *Data Contract*, a *Message Contract* and a *Service Contract*. We need to be able to group these concepts, inherit, extend and reuse them. Our organization has also made a considerable investment in its data dictionary, which describes (both at a table and an entity level) the data model underlying the whole application. When modelling services, we wish to leverage this investment.

At the same time, the service interfaces should not be mere repetitions of the underlying physical data model. This could very easily lead to a situation reminiscent of DCE, CORBA or DCOM, where the remoteness of a service call is hidden—the approach is to conceal the fact that a method invocation is actually on a remote object. This leads to unwanted dependencies, where internal behaviour (such as data types) is exposed, and often also to performance problems, since iteration over remote, low-level CRUD operations is easy to program but impossible to make fast and reliable.

Service-Oriented Architectures explicitly try to avoid sharing classes, since there are bound to be platform differences; instead, Data, Message and Service Contracts are used to specify messages and their associated content. The mapping from these types to the platform dependent types at each end is incidental and may not be predictable.

As an example, a data type specifying an integer with no upper limit may on some platforms be transformed into a string internally, if the platform lacks native unlimited-precision data types. Sharing class across platforms would require the correspondence to be known in advance, which cannot be taken for granted.

3.1 Using UML Profiles to define a Domain-Specific Language

UML is a modelling language for which there is quite extensive tool support. In UML it is possible to define *Profiles* that add semantic content to the generic mechanisms of UML itself, effectively making it possible to use it as a platform for developing Domain-Specific Languages (DSL). This is not the only possible approach; for instance, van Deursen (10) described a DSL for financial engineering that was used to generate several different kinds of artefacts (VSAM, CICS and COBOL items). It uses the MetaEnvironment tools (11) for the transformation/generation. However, UML and UML Profiles are open standards, and

therefore attractive by not locking the organization into a particular tool or tool vendor.

Of course, implementing a profile in a particular tool is dependent on the tool, but the concepts of the profile can in principle be transferred to another tool if needed, together with the models. The facilities provided by Rational XDE for defining profiles are fairly rudimentary, but have so far proven adequate to our needs.

In order to define our DSL, we have taken the minimum set of concepts needed to capture our modelling requirements, and translated them into a UML profile. These concepts cover both the service layer, the data dictionary, and the transition between the Service-Oriented and Object-Oriented worlds. It is critical to us that these two viewpoints are well integrated, since we will be implementing the services using object-oriented languages and tools. Figure 1 shows a simplified example of a service interface and how some of its data fields are derived from the data dictionary.

3.2 Generation of code and other artefacts

A model is useful in itself, as a design and documentation tool. However, its value is significantly increased if it can also be used to generate code, tests and embedded documentation. Such use is also a powerful incentive to keep the model up to date, as a working tool, and not just as a construct that was made early on in the development cycle and then quietly abandoned.

By definition, a model is an abstraction, and thus a simplification of the underlying reality. If a model contains enough information to fully generate the implementation, its complexity can easily become of the same order of magnitude as that of the implementation and its usefulness becomes doubtful. We therefore did not set out to find or create a tool that would generate the *content* of our implementations.

However, the structure of the services, the structure and skeleton of the implementation, and the “glue” logic required to technically define, deploy and put together services and their interfaces and implementations, are prime candidates for automated generation from models. The fact that these technologies change significantly over time, as new standards, tools and frameworks are adopted, provides a further powerful incentive for generating them.

Modelling and managing the transition between services and OO implementations is important, because best practices for design and grouping of them can be quite different. From our experience we believe that this is best done at the modelling level, and that generation of a skeleton for the implementation is extremely useful.

3.3 Code generation by text substitution

Commercial UML tools such as Borland Together or Rational XDE include code generators for several languages, such as C++, Java or C#. To a greater or lesser

degree, architects and developers can influence how the code generator works, i.e., what the emitted code looks like. Generally, however, what is a class in the UML diagram becomes a class or class-like construct in the code, and the adjustments one can make are more in the realm of coding style than semantics. One is also limited to generating the artefacts for which there is built-in support.

With the addition of semantic content through UML Profiles, this situation becomes untenable. The whole purpose of the profile is to capture semantics, that should then be reflected in the code. A UML “class” object that is assigned to a certain stereotype may not represent a class at all, but rather a service, a data contract, or a field with many descriptive attributes in a data dictionary. The standard code generators are not designed to handle this level of content.

We have therefore created a generic code generator that works by text substitution. It takes as its input a template text, and replaces recognized tags in

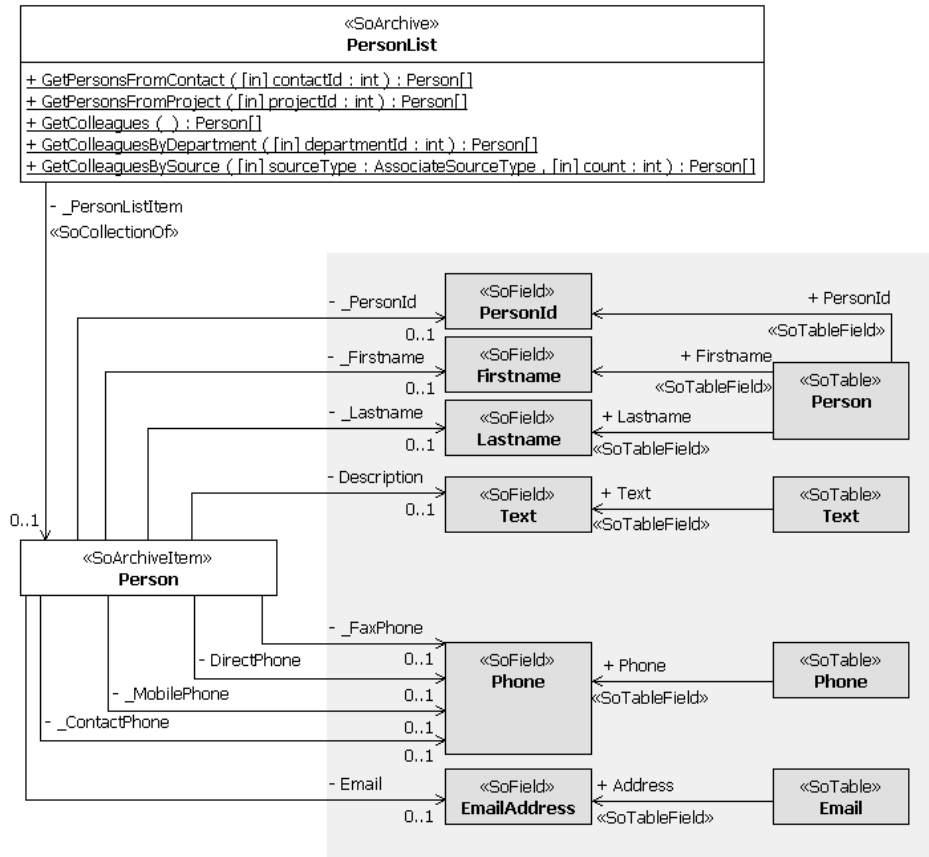
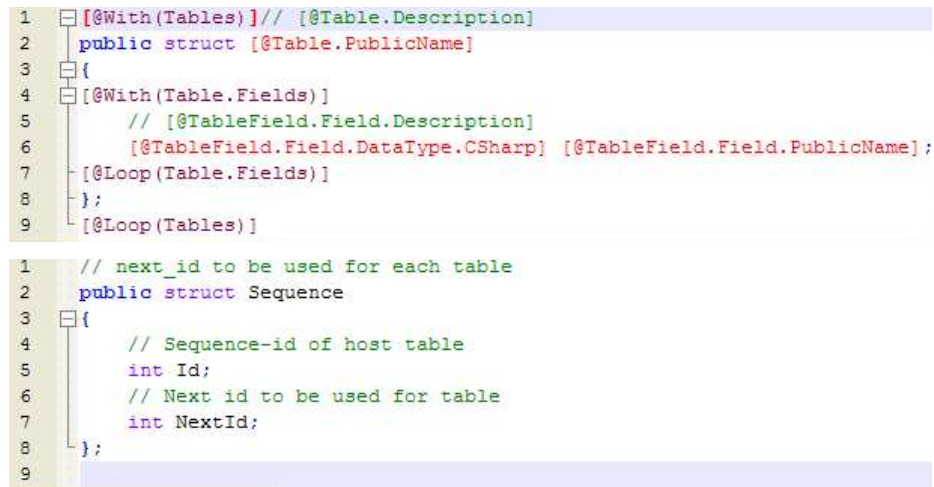


Fig. 1. Interface example: The shaded items come from the data dictionary model, while the clear items are service interfaces. The **«SoArchive»** stereotype denotes a list-like data set, while an **«SoArchiveItem»** is a single row in such a list.

the text with values from the model. The generator uses reflection on the UML profile and the UML tools' data model to define the tags; the (human) template author can then compose templates that translate into relevant, executable code. Simple looping and conditional constructs provide additional flexibility, while we do try to avoid the definition of an entire new programming language and environment within the code generator itself. Figure 2 shows a simple example, where we generate a C# object that corresponds to one database table and its fields; the lower half shows the result for a trivially simple table that has two fields. Syntax coloring is provided by a custom version of the Notepad++ editor (12).



```

1  [:@With(Tables)]// [:@Table.Description]
2  public struct [:@Table.PublicName]
3  {
4  [:@With(Table.Fields)]
5      // [:@TableField.Field.Description]
6      [:@TableField.Field.DataType.CSharp] [:@TableField.Field.PublicName];
7  [:@Loop(Table.Fields)]
8  };
9  [:@Loop(Tables)]

1  // next_id to be used for each table
2  public struct Sequence
3  {
4      // Sequence-id of host table
5      int Id;
6      // Next id to be used for table
7      int NextId;
8  };
9

```

Fig. 2. Simple data structure template, and generated result for a table with two fields

Our guiding principle is, as far as possible, to localize knowledge of the DSL semantics in the UML Profile and in the code templates. By keeping the actual code generator generic, we make it easier to extend and adapt the DSL as development proceeds: it is not realistic to assume that we will be able to define a DSL that fully supports all our needs early in the project, and then keep it constant for the duration.

Changes to the DSL imply changes to the UML Profile, possibly the existing models, and the code templates. They are therefore not to be undertaken lightly, but as long as we confine ourselves to extensions the cost is manageable. A breaking change to existing constructs would be costly; however, this is a problem common to all DSL tools that we know of—and most do not handle even simple extensions to the DSL.

As the problem domain is explored and the language matures, the rate of change over time decreases and languages become more stable. However, if a new aspect or a new domain needs to be modelled, we should expect to have to

make changes to the DSL. It is therefore critical that the tool chain supports at least extensions to the DSL in a straightforward manner.

3.4 Other uses for “code” generation

Having a text-based code generator that works by text substitution opens the possibility of generating other artifacts than executable code. The generator effectively becomes a simple transformation engine, and can be used to generate HTML documentation, WSDL service definitions, or deployment configuration files.

For instance, we can use the generator, together with a suitable template, to generate an HTML documentation file that contains service signatures, their descriptions (from documentation in the model), and cross-referencing tags that make it possible to seamlessly integrate the documentation into an existing development environment such as Visual Studio.

Another use is to generate WSDL service descriptions. Since the UML Profile contains concepts that make it possible to distinguish a public service from a private service from a simple RPC interface, it is relatively straightforward to use these attributes in the template text and generate WSDL only for the model elements that actually model services at the desired level.

A third use, illustrating the advantages of a template-based approach, is that we can also generate unit test skeletons from the same model—either in C#, or in some other language suited to the testing framework used.

3.5 Model transformation by code generation and reverse engineering

A conventional approach in Model-Driven Architecture (MDA) is to start with a platform-independent model (PIM), transform it through a set of rules to a platform-specific model (PSM) and from there to code. Examples can be found in (13; 14; 15), with some specifications in (1). However, the practical matter of setting up the transformation rules and tools to manage this in an automated manner is difficult—and if the transformation is performed manually, the overhead repeating it whenever the PIM or the transformation changes quickly becomes prohibitive.

For us, the SOA model, expressed in our DSL is the PIM, while its implementation in an object-oriented form in C#, and using Web Services (16) as a technical vehicle, is the corresponding PSM. That is, we define our DSL and external, service-oriented interface to be “platform-independent”. Note that at this level we hold no opinion as to the manner in which the service interfaces are to be accessed. This corresponds to the standard MDA view (1, Ch. 4.1.2).

Changing to use, for instance, Microsoft Indigo as the service access mechanism, would mean changing the transformation from the PIM to the PSM; since the underlying C# execution platform is the same, we would expect to be able to reuse the actual implementations with few changes. A more radical change,

for instance to a Java/Corba platform, would of course involve much more work, but we would still expect to generate the service definitions, interfaces and implementation skeletons.

Instead of performing a model-to-model transformation at a modelling-language level, we have chosen to perform the transformation directly from the PIM—which primarily models services—to the OO implementation (effectively the result of generating code from a PSM) directly.

The transformation rules are embedded in the code template that is half of the input to the code generation, the other half being the model and the semantics encoded in its use of the concepts from the UML Profile. Effectively, the template *is* a transformation rule—and it could have been a model-to-model transformation by setting up a template whose end product were valid XMI or some other, relevant metadata format. However, since our emphasis as a commercial development team is on creating a software product, we chose to concentrate on generating code.

Thus, our code templates actually combine two roles: transformation of the model from a platform-independent to a platform-dependent level; and transformation from a model to code (or similar-level artefacts such as WSDL). This combination is intentional, the main reason being efficiency.

Once the implementation code skeleton has been generated, it can be reverse-engineered using the standard functionality of the UML tool. The resulting UML model then becomes the result of applying the semantics of our UML Profile to our PIM, i.e., the PSM. Since it is reverse-engineered it always reflects the code, which is what is ultimately shipped to the customer. It therefore becomes a useful documentation and verification tool, rather than an intermediate step in the development process. A disadvantage of this approach is that the stereotypes from the PIM are lost, unless the generated code is somehow tagged, and the reverse-engineering mechanism recognizes the tags. Such recognition of extra tags is currently not available in Rational XDE.

4 Practical experience

Initially, the designer and user of the modelling tools and code generator was the same person (J. M. G.). As the project matured from a prototype / technology demonstration project to a full-scale development project with six developers, we have gathered more experience with both the technology and the organizational side effects.

We have conducted interviews with all the developers, ranging from the senior architect (M.V.) to recently hired developers with little modelling experience. In general, increased modelling experience correlates with an increased perception of the benefits of the approach.

4.1 Positive experiences

Perhaps the single most positive consequence of using a model is to raise the general consciousness level about the need for well-designed, thought through

interfaces. By making the separation between a service interface and its object-oriented implementation explicit, the developer is forced to take the difference into account.

Standardization is also an important benefit. Our code-generation templates contain and enforce a certain pattern for how a service, its messages and data should be related, and how they should be implemented for local and remote calls. Since all of this is generated, it will always be the same and consistent between services. “Standard” items, such as authentication tickets, are automatically added, again in a consistent way.

When new developers are added to an already established team, it may take some time to learn all the written and unwritten rules for design and coding styles. The combination of modelling and generation helps by codifying and enforcing the “standard” way of doing things.

Simultaneous generation of remote interfaces, local implementations, data and message contracts as well as unit test skeletons and documentation pages from a single model ensures that all of these artefacts are actually created. While we cannot force people to actually write good documentation or comprehensive tests, there is at least little excuse for not doing so—and empty tests or documentation pages are highly visible in code reviews. This increases the consistency of the work across developers

Generation also increases the visibility of “auxiliary” tasks such as documentation and testing. The importance of this rises with the approach of a deadline and the temptation to skip testing in order to finish in time.

While “local” changes—changes that affect just one or a few interfaces—do not benefit much from code generation, “global” changes that involve changes to how *all* interfaces or implementations are defined become much simpler to perform, usually by making changes to the template. Since they are applied equally to all relevant objects, consistency is easier to attain.

The fact that the code generator is an in-house tool is generally considered to be a positive factor. The tool quickly becomes central to the development process, and being dependent on a vendor’s release plan for fixes or changes could easily become a bottleneck. While the availability of the few developers who can update the tool can also become a limiting factor, it is at least under the team’s control. Open source tools are a possible alternative in this situation.

4.2 Challenges and pitfalls

While there are important benefits to be realized from modelling and generation, there are also costs and challenges involved. We have chosen to divide these into the purely technical, and those that are more cultural or organizational in character.

Technical challenges Currently, our model resides in a single file, and the code generator runs on the entire model every time. In practice, this means that only one developer at a time can have the model locked in the version control system

(merging of multiple versions is not practical). It also means that all target files are regenerated for every change. While the version control system will recognize and filter out submissions of files that have not actually changed, this still causes problems when scaling up to a team of six developers.

The problem is periodic in nature—typically, there is a period in each iteration where new services are designed and defined, followed by a period of actual implementation. The design period is one of high contention for the model, while the implementation and testing/debugging is independent of the model and therefore does not suffer.

The problem can mostly be solved by dividing the model into separately controlled packages, and by revising the code generator so it can be run on single packages, instead of the whole model. However, changes to templates or the UML profile will still force regeneration of the whole system.

To a certain degree, this problem is also related to the way our teams are organized. A team that has only a few designed architects/interface designers will have much less contention for the models, and can also afford to increase the amount of special knowledge and skills required to manipulate the models. Our teams are organized in the opposite direction, with most of the developers assuming most of the roles during one complete cycle.

A different challenge is posed by the size and complexity of the entities being modelled, and the way they are modelled. Since we are using UML as our basic modelling language, and the Association concept can only connect two Classes (as opposed to connecting specific attributes within or between classes), we ended up modelling service interface *attributes* as separate *classes*, with specific stereotypes. This gives us the flexibility and power we need, for instance by making it possible to reuse an attribute in multiple interfaces.

At the same time, this approach increases the visual complexity of the model and the screen real estate needed to contain it. When each attribute of an interface becomes a separate box with a line going to it, it is easy to run out of space on a screen. The model shown in Figure 1 was simplified for this paper, to make it a reasonable size—in reality, there about three times as many boxes of various kinds in the diagram. Even with dual 20-inch displays on each workstation, this can become an irritating problem. In the near term we do not see any easy solution; in the longer term, a modelling tool that is not based on UML, but is instead built to handle DSL's should provide a solution.

The template language is fairly straightforward, but since it is a proprietary language there is little tool support for it, in the form of syntax highlighting or word completion. We are currently looking at ways of adding these features to Visual Studio, to make template editing easier. Most of the suggestions for improvements—from the developers using the model—relate to details of the template language and tool support for editing templates.

Two of the developers have prior experience using XSLT expressions to transform models. Their perception is that the readability and traceability problems (which part of the template causes a certain output to appear) were larger in XSLT, and that they usually had to actually run a transformation or genera-

tion to see the output. With a template language based on simple substitution of textual tags, it is much easier to predict the output. This agrees with the experience of others, such as Czarnecki (2).

Tool support for transitioning from one version of the Profile to the next is largely nonexistent. We have created our own tool to bridge the gap, but serious use of any kind of DSL will be much hampered by the absence of such support. Ideally, a tool should provide an analysis of the consequences of a language change (such as an estimate of the number of modelled entities or associations affected or made invalid), as well as support for mapping concepts in the two language versions, and an application of that mapping to models made using the language.

Organizational challenges Even though we consider our template language to be simple, it still *is* a language, and it raises the learning threshold for new members of the team. It has to be learned, understood and worked with in order to be able to realize the full power of the approach. The alternative, where only one or two “master developers” understand the whole system, is both inefficient (they can become bottlenecks) and risky (in case they leave). The associated training costs are significant but acceptable.

The “extra” work involved in modelling an interface—using the Rational XDE GUI to draw interfaces, create or retrieve attributes and connect them, and attach the various stereotypes and parameters needed, may seem to be a drawback. A possible consequence is that this work is postponed, or that weaknesses in an interface are not corrected because the effort needed to do so is perceived as excessive. This may not actually be a big drawback: if it is very easy to change or create service interfaces, they will proliferate, likely with a decrease in the generality, stability and quality of each interface. Since our service interfaces will be used by partners, customers and consultants for many years, they have to be stable and of a high quality.

Generated code will often contain repetitions—the same template is used to generate skeletons and (partial) implementations for many objects in the model. At one level this may be considered a disadvantage; after all, refactoring to *avoid* repeating an algorithm in multiple places is a well established practice. If we view template as the “code”, it contains the algorithm only once, and changes to the algorithm are performed in the template, not the generated code. On the other hand, tools such as source browsers will show all the repetitions. Whether this is actually a problem on a significant scale remains to be seen, and we suspect it to be a matter of personal viewpoint and preference.

Costs and benefits Since we have not performed parallel development using models and code generation versus a more traditional approach, we do not have hard data on the costs and benefits. However, many of the features developed by this project have equivalents in existing code in the company, and those were implemented some time ago, mostly without modelling.

The costs are the most visible—it has taken one senior developer approximately two years to develop the Profile, the code generator and associated tools, and implement both a table- and entity-level database abstraction layer, plus a significant amount of infrastructure code. This is roughly comparable to the effort required for comparable development when the previous generation of the system was implemented.

The benefits, in terms of stability, error frequency or functionality, are harder to characterize. There is no doubt that the model-based approach encourages a much greater test coverage, and that it automatically leads to a degree of consistency that would otherwise require strict enforcement of the company styleguide and standards. We believe that the main benefits will accrue as we scale up the development both in complexity and in volume.

Since the costs associated with this approach are significant, we do believe that small or one-off projects are not likely to realize a net benefit. Our project will now scale up to six developers for approximately one year; we expect this to be large enough to realize a benefit, though we also expect the benefit to be more in terms of increased code quality and functionality, rather than reduced cost and time.

5 Conclusions and Future Work

Current modelling tools based on UML reflect the fact that UML semantics are informal, while specific enough to point clearly in the direction of an object-oriented target language. Since service-oriented architectures are not necessarily object-oriented, while their implementations often are, the standard code generators included in UML tools cannot be used for modelling both SOA interfaces and OO implementations directly.

At the same time, tools that support the creation of Domain-Specific Languages, are not yet ready for heavy industrial use. However, we view the DSL approach as perhaps the most promising to date and have adopted it for our development project.

Our solution has been to create a simple code generator based on text substitution, and to use a UML Profile to define the additional semantics we need in the modelling language. We thus transform our chosen UML tool (Rational XDE) into a simple DSL tool, albeit with limited functionality. The metadata that represent the model are reflected into the code generator, and the model-to-code transformations are provided by writing a code template that incorporates substitution tags matching the model metadata.

Our experience so far is that the approach works quite well for those lower layers of the application that express similar functionality repeatedly, such as Data Access Objects for individual tables. Here, “mass production” of functionality based on a template, repeated for each table, makes good sense.

When modelling more high-level services, the emphasis is more on the standardization of naming and behaviour, and the generation of skeletons rather than complete functionality. While the Data Access layer exhibits close to 80%

generated code, the service layer has less than 40% generated content—and this percentage may decrease as the implementation complexity increases.

However, it is the generated content that defines the interfaces and the implementation patterns, including the particular technology used to implement services. This is important, since the current emphasis on Web Services will surely be superseded by some other—hopefully compatible—technology within a time span that overlaps the lifetime of our product (typically 10 years). When this happens, regeneration of interfaces and “glue” logic should save a lot of effort, and provide a faster time to market for solutions compatible with new standards.

5.1 Future work

In the near future, our work will concentrate on making the template language more readable, as well as extending support for it into our development environment. Features such as syntax highlighting and auto-completion of reserved words, variables and other constructs is today taken for granted. The absence of such support makes editing of the templates unnecessarily tedious. Similarly, accessing the code generator from within the integrated development environment is desirable.

Further major development will probably wait for the availability of more sophisticated tools, for instance the Software Factory modellers announced by Microsoft. By being designed for customization and implementation of DSL's such tools should be more suitable than using profiles to force foreign semantics into existing UML tools.

We continually strive to find the correct balance between investment in in-house tools and dependence on external tools. In-house tools offer full control, at the price of full cost for the tools' development and maintenance. External tools reverse the equation, offering low cost but also a lower degree of control.

For a tool that is central to our development process, and using a technology that is not yet mature, we believe the in-house approach to be the correct one at this time. In the future, a switch to externally developed tools is quite probable, when sufficiently mature and well-supported tools are offered. Availability of the source code will probably be a distinct advantage, since it offers a “safety valve” in the case of problems that would otherwise threaten a development project.

Acknowledgements

Thanks are due to Guttorm Nielsen, Director of research & development of SuperOffice ASA for providing the time for writing papers in an otherwise hectic project timetable. Thomas Schjerpen, Martin Valland, Trond Nilsen and Jørund Myhre generously shared their insight and experience.

Our work has also been supported by the Norwegian Research Council ICT Programme “FAMILIER”, and participates in FAMILIES, ITEA project ip02009 of the EU Eureka $\Sigma!$ 2023 Programme.

Bibliography

- [1] Object Management Group: **Model Driven Architecture Home Page** (2004) <http://www.omg.org/mda/>.
- [2] Czarnecki, K., Helsen, S.: **Classification of Model Transformation Approaches**. In: **2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA**, Anaheim, USA (2003)
- [3] Czarnecki, K.: **Overview of Generative Software Development** (2005) <http://www.swen.uwaterloo.ca/kczarnec/gsdoverview.pdf>.
- [4] IBM: **Rational XDE** (2005) <http://www-306.ibm.com/software/awdtools/developer/rosexde/>.
- [5] Telelogic: **Telelogic Tau** (2005) <http://www.telelogic.com/products/tau/index.cfm>.
- [6] Borland Inc: **Together** (2005) <http://www.borland.com/together/>.
- [7] Microsoft Inc: **Visio 2003** (2005) <http://office.microsoft.com/en-gb/FX010857981033.aspx>.
- [8] Greenfield, J., Short, K., Cook, S., Kent, S.: **Software Factories**. Wiley, Indianapolis, USA (2004) ISBN: 0-471-20284-3.
- [9] Microsoft Inc: **Microsoft Grows Partner Ecosystem Around Visual Studio 2005 Team System** (2004) <http://www.microsoft.com/presspass/press/2004/oct04/10-26OOPSLAEcosystemPR.asp>.
- [10] Deursen, A.v.: **Using a Domain-Specific Language for Financial Engineering**. ERCIM News (1999)
- [11] CWI: **ASF+SDF MetaEnvironment** (2005) <http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment>.
- [12] Ho, D.: **Notepad ++, Version 2.8** (2004) <http://notepad-plus.sourceforge.net/>.
- [13] Judson, S.R., France, R.B., Carver, D.L.: **Specifying Model Transformations At the Metamodel Level**. In: **UML 2003 - Workshop in Software Model Engineering**, San Francisco, USA (2003)
- [14] Pires, L.A.F., Sinderen, M.v., Farias, C.A.R.G.d., Almeida, J.A.P.A.: **Use of Models and Modelling Techniques for Service Development**. In: **3rd IFIP International Conference on E-Commerce, E-Business and E-Government (I3E 2003)**, Guarajá, Brazil, Kluwer (2003) 441–456
- [15] Solberg, A., Oldevik, J., Agedal, J.A.: **A Framework for QoS-Aware Model Transformation, Using a Pattern-Based Approach**. In Meersman, R., Tari, Z., eds.: **On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE**. Volume 3291., Agia Napa, Cyprus, Publisher: Springer-Verlag GmbH (2004) 1190
- [16] W3C: **Web Services Activity** (2004) <http://www.w3.org/2002/ws/>.