# Patterns for Model-Driven Software-Development

Version 2.0, Dec 20, 2004



*Markus Völter*

*voelter@acm.org*
*völter – ingenieurbüro für softwaretechnologie*
*Heidenheim, Germany*
*www.voelter.de*

*Jorn Bettin*

*jorn.bettin@softmetaware.com*
*SoftMetaWare*
*Auckland, New Zealand*
*www.softmetaware.com*

# Status of this work

This paper presents a work-in-progress collection of patterns that occur in model-driven and asset-based software development. We really appreciate feedback!

The paper contains a couple of proto-patterns marked with a (P) after the pattern title. They describe ideas that may or may not evolve into full-blown patterns, and that are not (yet) described in proper pattern form. The proto-patterns have been included in the paper to hint at possible directions for extending the collection of patterns. We encourage people to contribute to the collection by suggesting additional proto-patterns. We are specifically seeking material in the area of versioning and testing in the context of MDSD.

# What is MDSD

Model-Driven Software Development is a software development approach that aims at developing software from domain-specific models. Domain analysis, meta modeling, model-driven generation, template languages, domain-driven framework design, and the principles for agile software development form the backbone of this approach, of which OMG's MDA is a specific flavor.

Here are a set of core values, which have been defined during a BOF session at OOPSLA 2003.

> *We prefer to validate software-under-construction over validating software requirements*
>
> *We work with domain-specific assets, which can be anything from models, components, frameworks, generators, to languages and techniques.*

*We strive to automate software construction from domain models; therefore we consciously distinguish between building software factories and building software applications*

*We support the emergence of supply chains for software development, which implies domain-specific specialization and enables mass customization*

Model-driven software development is about making models first class development artifact as opposed to "just pictures". Various aspects of a system are not programmed manually; rather they are specified using a suitable modelling language. These models are significantly more abstract than the implementation code that would have to be developed manually otherwise – they are specific to the domain for which the models are relevant. The modelling languages used to describe such models are called domain-specific languages (DSL).

Like any other (formal) language, a DSL has three constituent parts:

- A *metamodel* (also called *abstract syntax*) defines the building blocks of the language, and the rules how they might be combined to form legal models (sentences in the DSL)
- A *concrete syntax* defines the actual notation used to specify models (or sentences). A particular metamodel might have several concrete syntaxes; concrete syntax can be textual (in which case sentences are often called *specifications*) or graphical (where sentences are often termed *models*). We use both terms interchangeably.
- Finally, a DSLs needs to have *semantics*; the meaning of models has to be well-defined. We will return to the issue of semantics definition below.

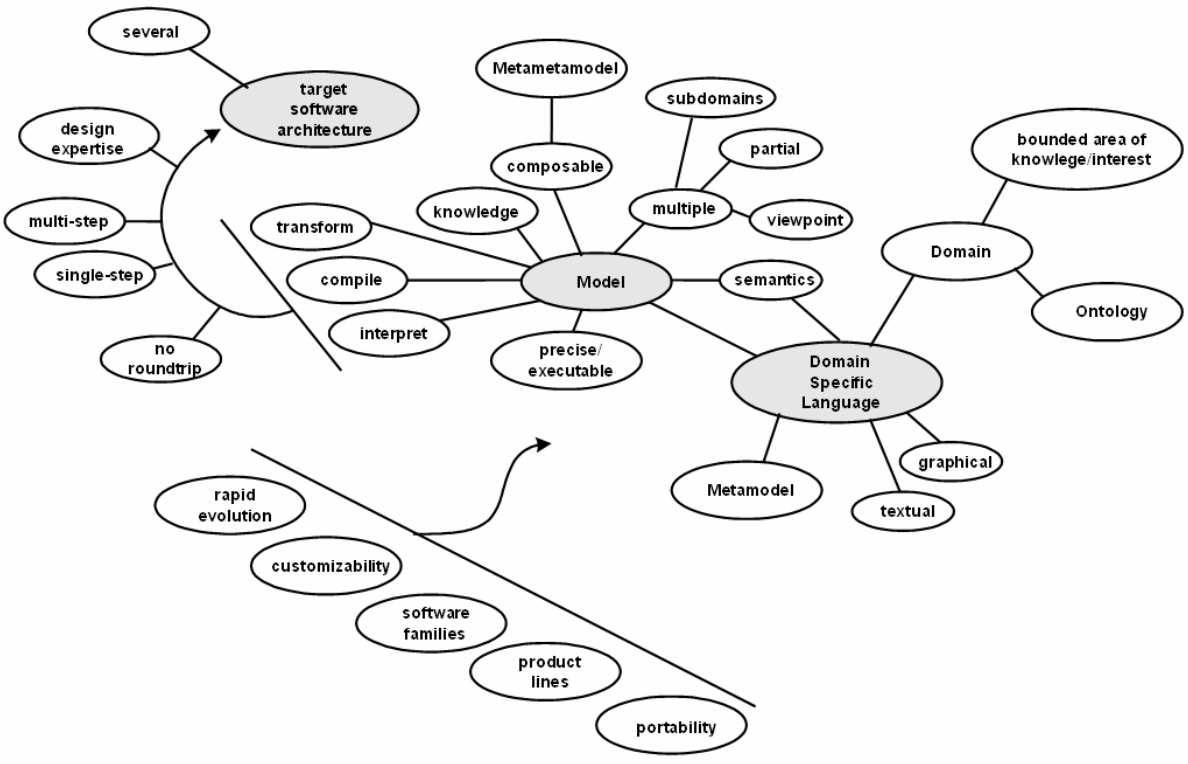
Models themselves are not useful in the final application. Rather, models have to be translated into executable code for a specific platform. Such a translation is implemented using model transformations. A model is transformed into another, typically more specific (less abstract) model; a series of such transformations results in executable code, since the last transformation is a model-to-code transformation. Because of today's somewhat limited tool support, many MDSD infrastructures use just one generation step, directly from the models to code. Model transformation tools using the latter approach are often referred to simply as model-driven code generators.

In addition to producing less abstract models (or implementation code), model transformations also serve the purpose of defining the semantics of the model they transform. By describing the rules of how a model is projected onto an implementation language, the meaning of the model is

defined. Although this is a rather pragmatic approach of defining semantics, it works well in practice[1].

Complex systems typically consist of a variety of concerns, such as components and their interfaces, the description of the deployment infrastructure (hardware) or timing and concurrency concerns. It is often not practical to use a single modelling language for all of these aspects; specifically, different concrete syntaxes are often useful. For example, the components and interfaces can be described using (stereotyped) UML, the hardware and the deployment using XML, and dynamic and concurrency aspects using a specific textual language. The generator must be able to understand all of these, and integrate the different partial models into a coherent whole. Note that some aspects of an application – typically the application logic – might be sensibly described using a 3GL programming language. It is perfectly ok to use a 3GL, nobody should feel forced to "invent" a DSL if the general purpose programming language is efficient for the particular task at hand.

The following illustration shows a mind map with the most important concepts.




[1] This definition of semantics has the disadvantage that you cannot formally ensure that if you have several sets of transformations which transform the models to different target languages (with well-known semantics) the defined semantics are the same. In practice you can use testing to make sure they *are* the same, but there is no formal way to ensure it.

# Pattern Form

The patterns are documented in Alexandrian form. Since this form is now widely used and well known, we refer readers to Christopher Alexander's original work on pattern languages[Alexander 1977] for further details.

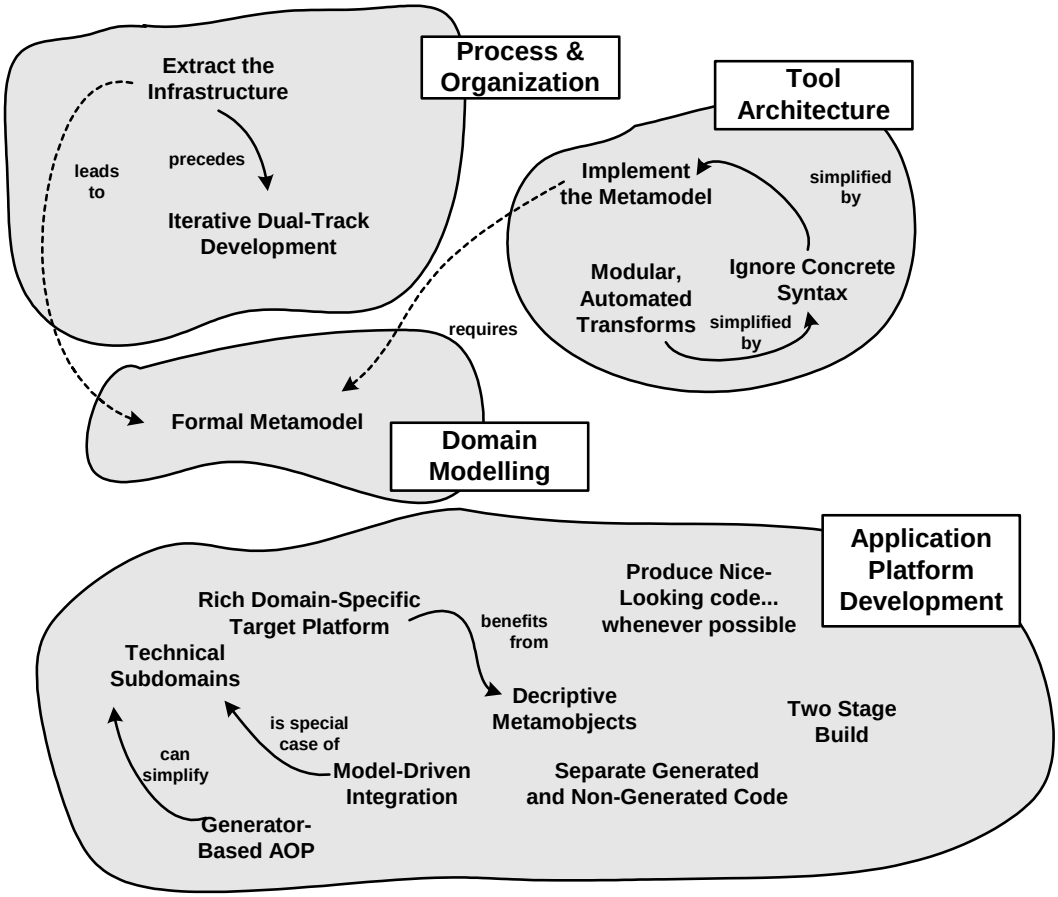
Note that, just as in Alexander's Pattern Language, we will qualify each pattern with no, one or two asterisks:

- No asterisk means that we are not very sure about the patterns content.
- One asterisk means that we think the patterns is valid, but we are not sure about details, formulations or all the forces.
- Two asterisks mean that the pattern is a fact.

The number and the quality of the known uses is also proportional to the number of asterisks.

# Overview

The patterns are structured into several groups: Domain Modeling, Process & Organization, Tool Architecture, and Application Platform Development. The following illustration shows the relationships among some of the patterns, as well as their association to the groups mentioned.


Process &
Organization
Extract the
Infrastructure
precedes
leads
to
Iterative Dual-Track
Development
Tool
Architecture
Implement
the Metamodel
simplified
by
Modular,
Automated
Transforms
Ignore Concrete
Syntax
simplified
by
requires
Formal Metamodel
Domain
Modelling
Application
Platform
Development
Produce Nice-
Looking code...
whenever possible
Rich Domain-Specific
Target Platform
benefits
from
Technical
Subdomains
Decriptive
Metamobjects
Two Stage
Build
is special
case of
can
simplify
Model-Driven
Integration
Separate Generated
and Non-Generated Code
Generator-
Based AOP

# The Patterns

## *Process & Organization*

### ITERATIVE DUAL-TRACK DEVELOPMENT **

You are developing a software system (family) using MDSD. One or more teams are working on one or more applications, and you need to develop a domain-specific infrastructure (application platform). You need to deliver iterations at fixed points in time, and the disruption caused by upgrading to new iterations of the infrastructure needs to be minimized.

☆ ☆ ☆

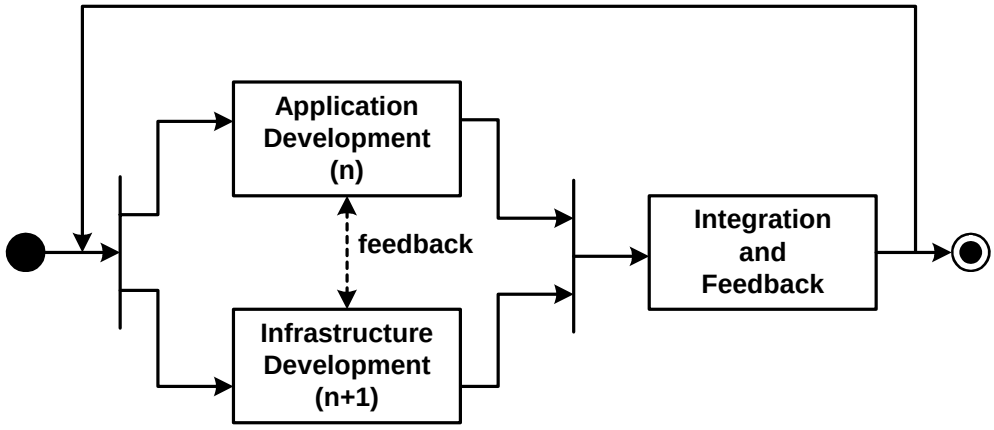
**When building a new software system family, you actually have to develop two things: a concrete application as well as the MDSD infrastructure that helps you build the applications based on the family. Development of an elaborate infrastructure in parallel with application functionality can compromise the stability of scope of application development iterations because of the repeated refactoring of application code to the updated platform.**

The MDSD infrastructure consists of transformation definition, the meta model, the concrete syntax definition as well as the target platform(s).

You cannot build applications based on the MDSD infrastructure unless the infrastructure is in place. Also, you cannot build the infrastructure if you don't have a solid understanding of the application domain, typically gained by developing a couple of applications in the domain.

Therefore:

**Develop the infrastructure as well as at least one application at the same time. Make sure infrastructure developers get feedback from the application developers immediately. Develop both parts incrementally and iteratively to achieve overall agility. To solve the chicken-and-egg problem, EXTRACT THE INFRASTRUCTURE from a running application. That means, in any particular iteration infrastructure development is one step ahead, and new releases of infrastructure are only introduced at the start of application development iterations.**




☆ ☆ ☆

In practice, to achieve sufficient agility, iterations should never be longer than four to six weeks and it is a good idea to use a fixed duration for all iterations.

Note that this incremental, iterative process based on synchronized timeboxes does not mean that you should not do some kind of domain analysis as described in [Cleaveland 2001] before starting development. A good understanding of the domain is a useful precondition for doing MDSD. Once development is under way, further domain analysis is performed iteratively as required as part of the infrastructure workflow.

An infrastructure team is at risk of leaping on interesting technologies and then hijacking the agenda to embark—with the best intentions—on bestowing the rest of the world with a new silver bullet. This risk can be managed by ensuring that the architecture group (i.e. consisting of representatives from the application development teams) is given the mandate to exercise SCOPE TRADING and VALIDATE ITERATIONS, so that the infrastructure being developed becomes a real asset from the perspective of application developers.

As a downside of this approach, it requires effective synchronization among the different sub-processes, and versioning can become an issue, specifically with today's MDSD tools. Also, updating (refactoring) the application models to comply to and utilize the new version of the infrastructure can be a non-trivial endeavor.

Note that:

- Once MDSD is well established in an organization, the technology infrastructure is highly standardized, and the focus of work shifts from standardizing use of technologies to building a domain-specific application platform, hence the term "application platform development" becomes a more accurate description over time.

- In the minimal case of a one-person project, this pattern collapses into the requirement to cleanly separate the code base of infrastructure (application platform) from the code base of individual applications.

☆ ☆ ☆

*The Family-Oriented Abstraction, Specification, and Translation (FAST) process [WL 1999] developed by AT&T has been used since 1992 and clearly differentiates between domain engineering and application engineering. FAST is based on experience over two decades of developing software families and has been evolving further at Lucent Technologies where it has been applied to over 25 domains.*

*One of the authors has been using iterative dual-track development since 1994, initially in conjunction with the programmable LANSA RUOM model-driven generator [LANSA], and later in several projects using different MDA tools.*

*Also, the b+m generative development process (GDP, see[GPD]) which has been used for a long time in the area of model driven development uses this principle as its basic foundation. It has proven to be essential to MDSD*

*Further concrete examples for the organization (team structure) of product line development are found in [Bosch 2000].*

## EXTRACT THE INFRASTRUCTURE **

You are developing a software system (family) using model driven development techniques. You do not yet have an MDSD infrastructure for the respective domain in available.

☆ ☆ ☆

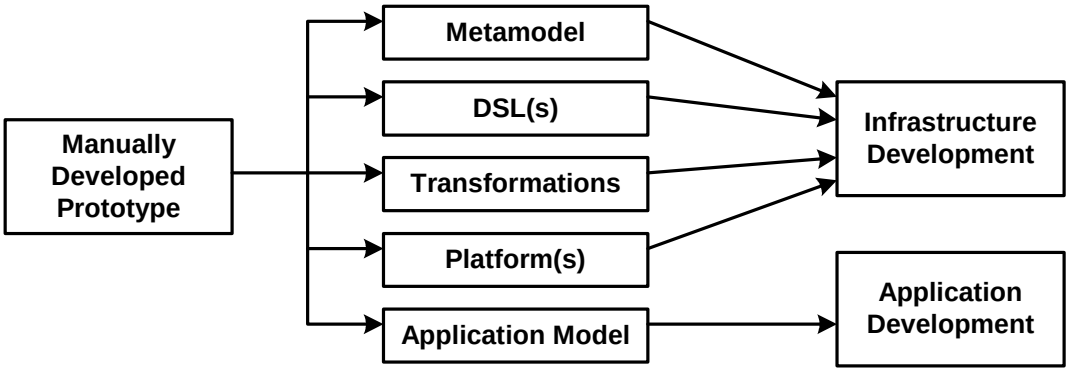
**At the beginning of an MDSD project you often don't know how to get started. You know, that you should use ITERATIVE DUAL-TRACK DEVELOPMENT, but how do you get started, anyway? You want to have at least one running application as fast as possible.**

Building a MDSD infrastructure requires you to think in terms of model transformations and meta models. You will have to scatter the implementation code over many transformation statements/rules (code generation templates, etc.). This is an approach many people are not use to.

Also, you want to make sure you don't have to debug the generated code forever in the early phases of the project. The generated code should have a certain minimum quality. You want to make sure it actually works.

Therefore:

**Extract the transformations from a running example application. Start by developing this prototype conventionally, then build up the MDSD infrastructure based on this running application. Start ITERATIVE DUAL-TRACK DEVELOPMENT after this initial stage.**




☆ ☆ ☆

The prototype application should be a typical application in the respective domain, not overly simply, but also not too complicated. If you are building a big and complex system, only use a subsystem of the whole system as an example.

There are two flavors of this pattern:

- In case you have been working on applications in the respective domain for a while and want to introduce a model-driven approach, you EXTRACT THE INFRASTRUCTURE from the previously developed applications.
- In case you start with a completely new software system family (green field), you should really develop a prototype in the sense of the word and EXTRACT THE INFRASTRUCTURE from it.

Note that it is important that you extract the infrastructure from an application that has high-quality architecture since this will be the basis for your software system family. So, even if you do have a set of legacy applications, it might be a good idea to write a *new* prototype with a new, improved, cleaned-up architecture.

Based on the experience of the authors as well as other practitioners, this infrastructure extraction (or "templatization") takes roughly 20-25% of the time it takes to develop the prototype.

This approach not only allows you to extract the transformations, it also helps you come up with a reasonable domain meta model as a basis for your DSL. Coming up with an expressive, small DSL also needs iterations as described in ITERATIVE DUAL-TRACK DEVELOPMENT.

Note also that some kinds of generators (specifically those using text templates) support the extraction of template code from running programs very well.

As a final remark we want to mention that the “templates” as mentioned above have nothing to do with C++ templates. Code generation templates are typically specific to a generator and allow you to navigate over the meta-model. They provide all the usual control logic constructs, nesting, etc. Typically, they are quite small and simple to use.

*In a project to develop a model-driven infrastructure for embedded systems, the communication core for the system family will be completely generated from models. The project will first develop a complete implementation of the core for one specific scenario manually and then extract the generative architecture from this prototype. This is done although the company does have experience in the domain.*

*This pattern was the motivation for the LANSA template language, and it was later used in 1993 as the foundation for LANSA RUOM, a customizable template language-based model-driven generator. One of the authors consistently used "templatization" of prototype code to automate pattern-based development in many projects from 1994 onwards, to build insurance systems, distribution systems, enterprise resource planning systems, and electricity trading systems. Although today's MDSD tools still use non-standardized template languages, the fundamental process is the same. The authors can confirm the validity of this pattern for software development with LANSA RUOM [LANSA], Codagen Architect [Codagen], eGen [Gentastic], GMT Fuut-je [Eclipse GMT], the b+m openGeneratorFramework [GenFW].*

# Domain Modeling

## FORMAL META MODEL **

You are developing a MDSD infrastructure for a software system family. You want to define your applications (family members) using a suitable Domain Specific Language (DSL).

**A domain always contains domain-specific abstractions, concepts and correctness constraints. These have to be available in the DSL used for developing applications in the domain. How do you make sure**

**your DSL and the application models defined with it are correct in the sense of the domain?**

Consider a DSL based on UML plus profile where you represent domain concepts as stereotyped UML classes. While UML allows you to define any kind of associations between arbitrary model classes, this might not make sense for your domain. Only certain kinds of associations might be allowed between certain kinds of concepts (stereotyped classes). In order to come up with valid models, you have to respect these domain-specific constraints.

Therefore:

**Use a formal means to describe your meta model. Describe it unambiguously and make it amenable for use by tools (IMPLEMENT THE META MODEL) to actually check your application models described using the DSL. TALK METAMODEL based on the FORMAL METAMODEL to verify it during its use.**

There are several useful notations for defining meta models. One very popular one, especially if you're using a UML-based DSL is MOF. If your DSL is based on extending the UML (e.g. using a Profile), make sure your meta model is as restrictive as possible and only allows the constructs you want to support explicitly – "disable" all the non-useful default UML features. Another useful meta modeling technique can be based on feature modeling – especially if you're mainly configuring applications with features.

It requires quite some domain-experience to come up with a good domain-specific meta model and DSL. In many organizations, however, a model-driven approach is used primarily to auto-generate the "glue code" required to run business logic on a given technical platform. In such as case, you may want to use an ARCHITECTURE-CENTRIC META MODEL.

If the latter approach is used, the main purpose of the meta-model is to enforce specific architectural constraints and to provide an efficient mechanism for designers to express specifications that is free from concrete syntax of implementation languages and from implementation-platform dependent design patterns.

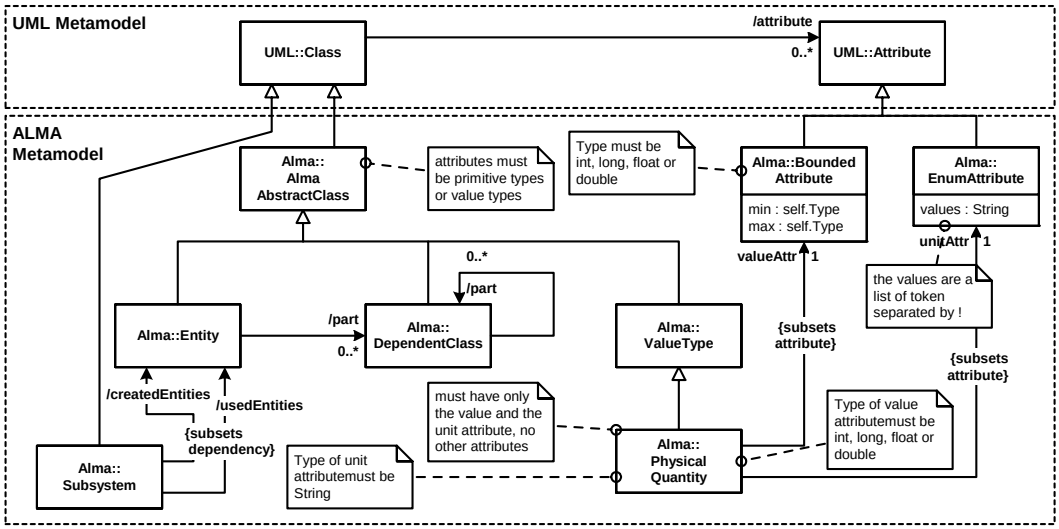
When building the meta model, make sure you understand your domain. Building a glossary or ontology as a first step can help. Of course, the meta model is defined incrementally, using ITERATIVE DUAL-TRACK DEVELOPMENT.

Note that a FORMAL METAMODEL is a very important precondition for coming up with a valid DSL and it is also the base for IMPLEMENTING THE METAMODEL – itself the basis for domain-specific tool support. However,

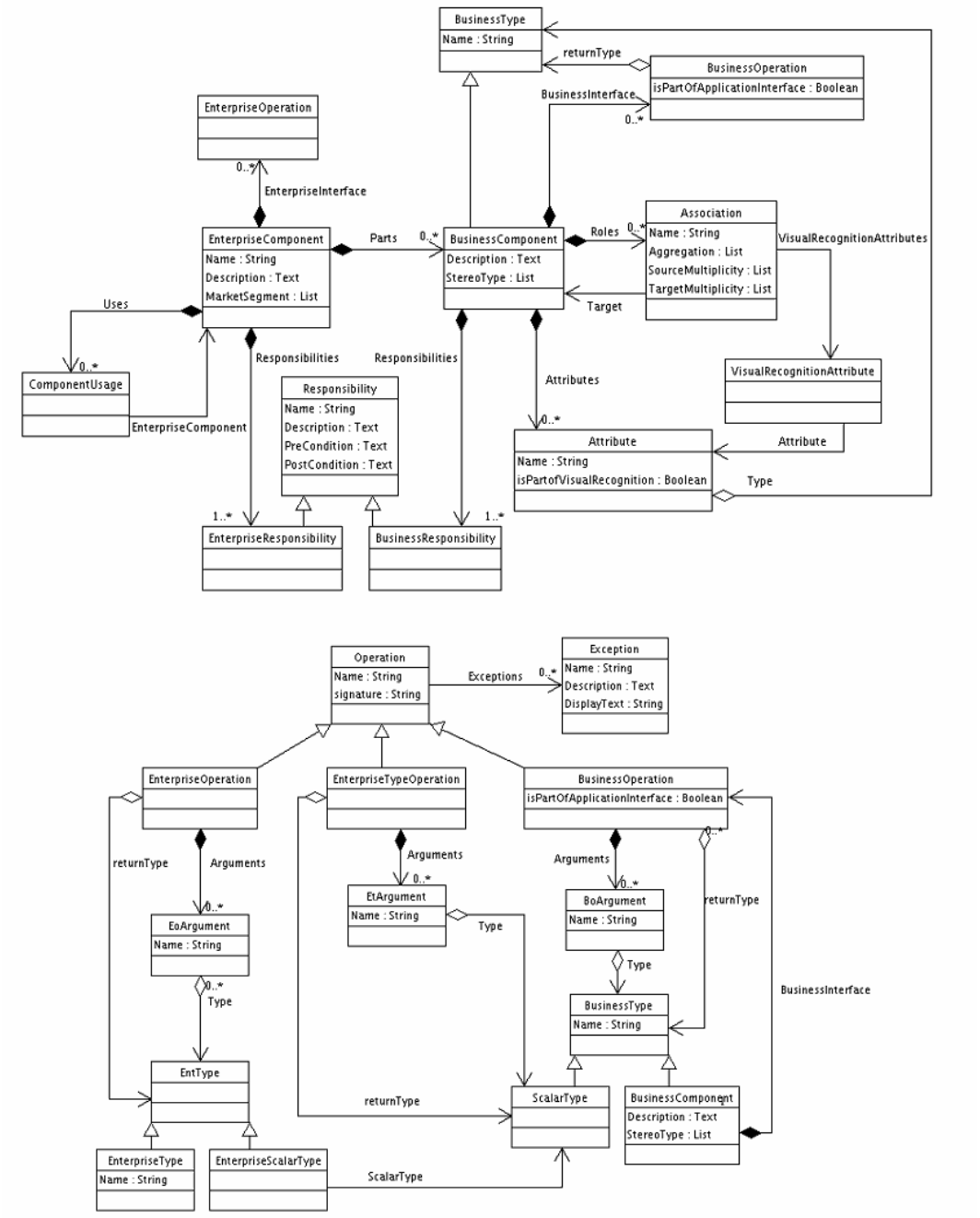
you still need to verify that the metamodel actually represents the real-world domain correctly; you may want to TALK META MODEL to do this.

☆ ☆ ☆

*The ALMA radio astronomy project uses the meta model defined below to define (parts of) its data model. The meta model has been iteratively developed and finally it has been formally documented in the form below. The meta model is also implemented for use by the code generator tool.*




*The following two diagrams show a somewhat more elaborate meta model that has been developed for large-scale distributed development of business applications, see [Bettin 2003] for more information.*




*Another situation where a formal meta model really helped was while writing a book on model-driven development. We wanted to sort out the commonalities and differences between MDSD, MDA, GP and architecture-centric MDSD. The breakthrough came when we came up with a* FORMAL META MODEL *for the domain of MDSD and its specializations.*

# Tool Architecture

## IMPLEMENT THE META MODEL **

You have a FORMAL META MODEL for your domain.

☆ ☆ ☆

**Having a formally defined meta model for your domain is a good thing; however, you need to use it efficiently when developing applications that are part of the defined family. The meta model will not be useful if it is only documented on paper somewhere – just as any paper-only artifact.**

Manually checking models against the underlying meta model is an error prone and tedious task. Relying on off-the-shelf modeling tools typically does not help, since they don't "understand" your meta model (this may change over time!). UML tools, even if they understood your meta model, could only check UML based models.

However, to make sure your generator can actually generate and configure your application, you have to make sure your model is "correct" in the sense of the meta model.

Therefore:

**Implement the meta model in some tool that can read a model and check it against the meta model. This check needs to include everything including declared constraints. Make sure the model is only transformed if the model has been validated against the meta model.**

☆ ☆ ☆

This approach is in line with MDSD, since you want to make sure your meta model is not „just a picture", but instead a useful asset in your MDSD process. You can generate the implementation for the meta model from the meta model itself using an MDSD approach, or implement it manually.

The meta model implementation is typically part of the transformation engine or code generator since a valid model is a precondition for successful transformation.

☆ ☆ ☆

*The b+m generator framework [GenFW] allows the implementation of the domain meta model using Java classes. Each meta model element is implemented as a Java class. When a model is read, the model is represented as instances of the meta model elements. Since all meta*

*model classes can implement a* CheckConstraints() *operation that is called by the framework, it is easy to implement constraint checking. The generator uses the UML meta model as a default, which can be extended or replaced by the developer.*

*LANSA RUOM was designed to not only provide an OO modeling capability, but also a limited degree of meta-modeling focussing on the definition of architectural constraints. LANSA RUOM allows users to define the allowable dependencies between different [user definable] types of components, and then prevents illegal dependencies from being defined in the RUOM modeling tool. This approach is distinctly different from the approach of standard UML modeling tools, where virtually no constraints are checked, and where only conformity with UML syntax is checked. See [Bettin 2001] for an example of a situation where conformity with standard UML syntax gets in the way of visually expressing architectural structure. The RUOM meta modeling capability has proved useful for many organisations.*

*The eGen generator from Gentastic allows visual meta-modeling, and generates a design portal that enforces the constraints consistent with the cardinalities in the meta-model. This approach is ideal for the definition of domain-specific meta models. The main drawback in this particular example is the quality of the generated design portal.*

*The current version of the GMT Fuutje tool allows limited soft-coded meta-modeling along the lines of UML tagged values, i.e. meta model elements can be extended with additional attributes. More extensive meta-model changes need to be realized in the form of Java code, probably somewhat similar to the approach taken in the b+m generator.*

*The Codagen Architect generator is tied to the UML meta-model, and relies on tagged values to be passed from standard UML tools to the generator. This approach does not allow for any constraint checking at design time in the UML tool, and any "invalid" tagged values in UML models are detected only at generation time.*

## IGNORE CONCRETE SYNTAX * *

You want to transform a model or generate code from a model.

☆ ☆ ☆

**Every model must be represented in some concrete syntax (e.g. XMI for MOF-based models). However, defining the transformations in**

**terms of the concrete syntax of the model makes your transformations clumsy and cluttered with concrete syntax detail when you really want to transform instances of your meta model. How can you make sure your transformations do not depend on concrete syntax?**

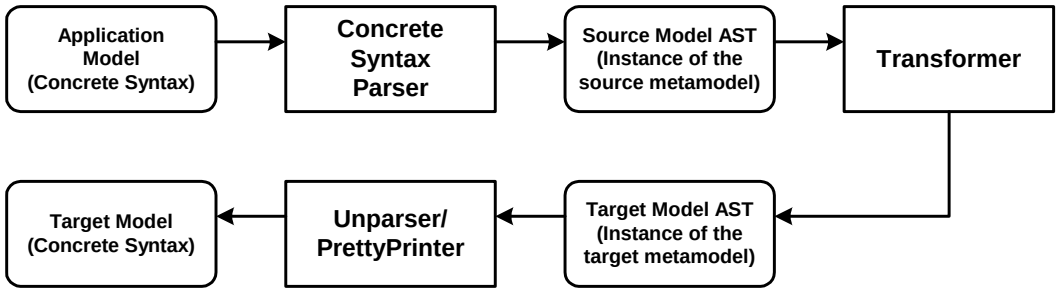
Definition of transformations based on the concrete syntax is typically a very error-prone an inefficient task. Consider using XMI. XMI is a very complicated syntax. Defining transformations based on XMI (and maybe XSLT) is not very efficient.

Also, in many cases several concrete syntaxes are useful for the same meta model, for example if you are using different DSLs for different TECHNICAL SUBDOMAINS. Defining the transformations based on concrete syntax unnecessarily binds the transformation to one specific concrete syntax.

Therefore:

**Define transformations based on the source and target meta models. Make sure the transformer uses a three phase approach:**

- **first parse the input model into some in-memory representation of the meta model (typically an object structure),**
- **then transforms the input model to the output model (still as an object structure)**
- **and finally unparse the target model to a concrete syntax**




☆ ☆ ☆

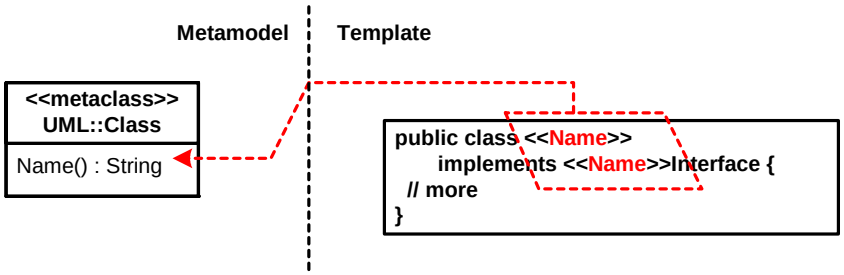
This approach results in a much more efficient and productive way of specifying transformations. It also makes the transformer much more flexible, because it can work with any kind of concrete syntax. This is particularly important in case of XMI-based concrete syntax, because the details of XMI vary between UML tools. You don't want to bind you transformation to one specific tool (and maybe even tool version).

Code generators (transforming a model to code) often do not use the full three phase approach, but directly generate textual output from the input model instance; creating an output AST instance would be overly complicated. Instead, templates are used that access the source meta model.

Note that this approach fits together neatly with the IMPLEMENT THE META MODEL pattern. If done right, the same implementation can be used for both

purposes. The templates can then access the meta objects directly; properties of the meta objects can be used to provide data for template evaluation as shown in the following illustration.

It is also worth pointing out that compilers have been using this approach for a long time. They are structured into several phases, the first one parsing the concrete syntax and building an abstract syntax tree in memory, on which subsequent phases operate.




☆ ☆ ☆

*The diagram above is representative of most template language based generators/transformers.*

*Revisiting the b+m generator framework, we already saw that it represented the applicable meta model as Java classes. The transformations are template-based (since it generates code directly). These templates can contain statements that reference the meta model and its properties. You do not see anything of the concrete syntax of the model. A front-end is responsible for parsing the concrete syntax and instantiating the meta model elements.*

*Just as the B+m generator, Fuutje GMT, Codagen Architect, eGen, and LANSA RUOM all use a template language to shield the user from the concrete syntax of the model. In LANSA RUOM the template language is fairly weak, which is compensated by allowing fully user definable pre-template processors, which perform the role of translating between concrete model syntax and template variables at generation time. In eGen the available template variables and navigation are a direct reflection of the user definable meta model.*

*As far as we know, most of today's MDA tools rely on non-standard template languages for the mapping of models to textual artifacts such as code. Limitations of these template languages and issues with proposed alternative approaches are sketched in [Bettin 2003b].*

## TWO STAGE BUILD *

You are working in the context of a software system family and need to design a model driven generator.

☆ ☆ ☆

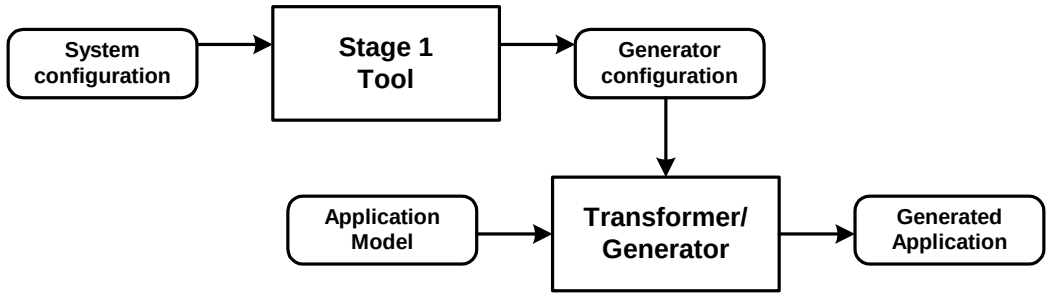
**It is often very complex to incorporate all product configuration steps into one transformation run. Features might have dependencies among each other. Different parts of the system are typically specified using different means. How can you build a simple, maintainable, and adaptable transformation process?**

Consider the selection of a target platform. Depending on the platform, different transformations must be executed. The selection of the platform thus determines which transformations to execute. It is very hard to incorporate all alternatives into one set of transformations that takes care of all possibilities.

Other such issues are the selection of certain libraries, or typical cross-cutting concerns.

Therefore:

**Separate the generation run into two stages: the first stage reads some kind of configuration and prepares the actual generator for the core transformation. The second stage is the execution of the transformer and uses the preparations done in the first stage.**




☆ ☆ ☆

In many cases, the first stage uses a different tool (such as a batch file or an ant script) to prepare the generator itself. Also, while the model for the second phase often describes application functionality or structure, the specification for the first step is actually more of a tool configuration activity.

As a consequence of the fact that there is no well-proven paradigm for transformation/template code management, each tool has its own idiosyncrasies. Usually the approach taken is driven much more by the tool architecture than the structure of the domain. Since many tools use a file-based approach, the example given below is representative.

Note that this approach is also used in open source distributions, where *make install* is used to prepare the makefile that in a second step builds the application.

*In the small components project, an XML-based specification is used to define the target platform, etc. Based on this, the ant tool is used to prepare the environment in which the generator operates; specifically, it copies the applicable set of template files to the locations from where the generator will load them. In a second stage, the generator itself processes the templates and thus generates code from the application model, this one being a combination of UML and XML.*

*As indicated earlier, in LANSA RUOM pre-template processors read the model, and set up the template execution environment for each template. The pre-template processors are also the place for model-driven integration, as they may access meta-information about pre-existing systems as required, and make this information available to RUOM templates. Another interesting feature of LANSA RUOM is the existence of post-template processors, which allow the replacement of user definable tokens (post-template processor commands) with arbitrary code. The tokens need not always be present in the template, but may be part of the generated code—resulting from computations with template variable content. Thus in LANSA RUOM the build consists of three main stages. This feature would not be required in a template language that fully supports recursive template execution.*

*In general there are significant differences in the way tools prepare and coordinate template execution. In eGen for example, code that links templates is physically separated from template code. In Codagen Architect, the template execution sequence and relationships between templates are indirectly specified via a classification scheme of templates and via the ordering in the list of templates. The common theme through all the tools is a "multi-stage" build.*

# SEPARATE GENERATED AND NON-GENERATED CODE **

You are generating large portions of your application, but you still have to program some aspects manually.

☆ ☆ ☆

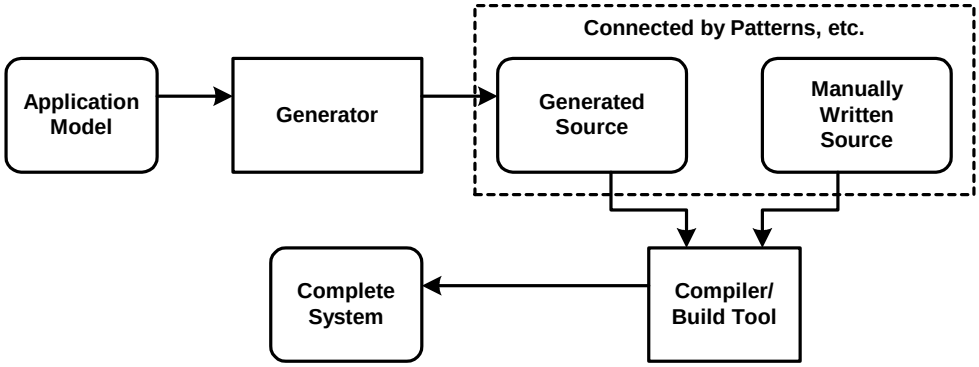
**If only parts of the application is generated, "blanks" must be filled-in by manual programming. However, modifying generated files by adding non-generated code creates problems in the areas of consistency, build management, versioning and overwriting of manually written code when regenerating.**

If generated code is never ever modified, the whole generation result can simply be deleted and regenerated if necessary. If the code is modified, there must be special protected areas that the generator does not delete when regenerating code; this requires the generator to actually re-read the generated code before regeneration and to preserve the protected areas. Consistency problems can arise (when the model is changed in ways that make the non-generated parts incompatible).

Also, versioning is more complicated, since the manually written code and the code generated from the model are in the same file, although they should be versioned independently.

Therefore:

**Keep generated and non-generated code in separate files. Never modify generated code. Design an architecture that clearly defined which artifacts are generated, and which are not. Use suitable design approaches to "join" generated and non-generated code. Interfaces as well as design patterns such as factory, strategy, bridge, or template method are good starting points (see [GHJV95]).**




☆ ☆ ☆

As a consequence of using this pattern, the application is forced to have a good design that clearly distinguishes different aspects. Generated code can be considered a throwaway artifact that need not even be versioned in the version control system. Consistency problems thus cannot arise.

There is another reason why this pattern is critical: Often the hand-crafted code (that is not practical to generate) is also the code that needs to be adapted when implementing a new variant of a product or a family member of a product line. Thus separating generated from non-generated code is critical for effective management of variants and helps to identify points of variation.

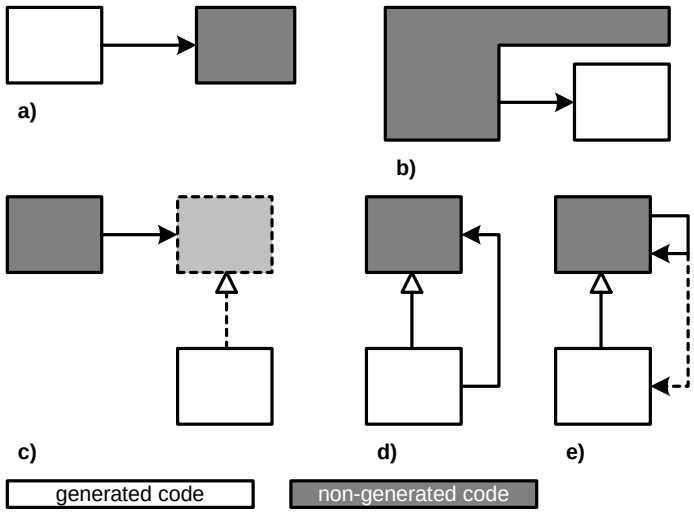
On the downside, this approach sometimes requires a bit more elaborate design or some more (manual) programming.

Sometimes, for performance reasons, there are situations when direct insertion of manually written code into generated code is unavoidable, this makes the introduction of protected areas mandatory.

This pattern can be generalized in the sense that in many cases, you have several generators generating different parts of the overall system, for example in the context of TECHNICAL SUBDOMAINS. Manually written code can be seen as only a very special kind of generator (the programmer ☺). The architecture clearly has to cater for these different aspects.

☆ ☆ ☆

*The following diagram shows, how generated and non-generated code could be combined, using some of the patterns mentioned above.*




*First of all, generated code can call non-generated code contained in libraries (case (a)). This is an important use, as it basically tells you to generate as few code as possible and rely on pre-implemented components that are used by the generated code. As shown in (b), the*

*opposite is of course also possible. A non-generated framework can call generated parts. To make this more practicable, non-generated source can be programmed against abstract classes or interfaces which the generated code implements. Factories can be used to „plug-in“ the generated building blocks, (c) illustrates this.*

*Generated classes can also subclass non-generated classes. These non-generated base classes can contain useful generic methods that can be called from within the generated subclasses (shown in (d)). The base class can also contain abstract methods that it calls, they are implemented by the generated subclasses (template method pattern, shown in (e)). Again, factories are useful to plug-in instances.*

## RICH DOMAIN-SPECIFIC PLATFORM **

You are generating code from domain-specific models.

☆ ☆ ☆

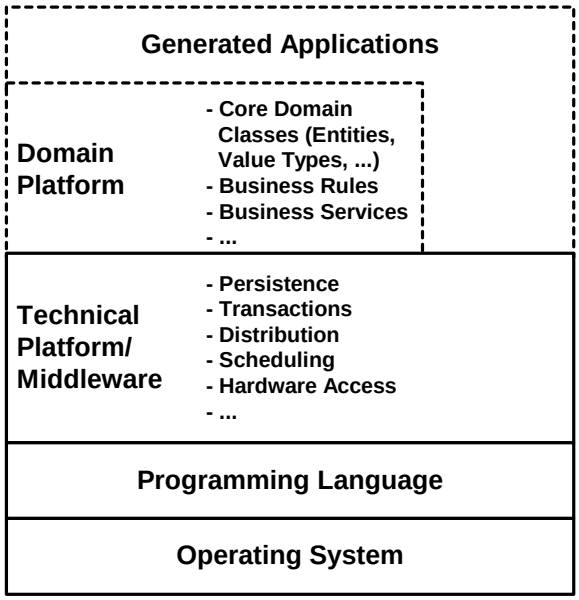
**In the end, application models must be transformed to a certain target platform to be executed. The bigger the difference between the domain concepts and the target platform, the more complex the transformations have to be. With today's tools this can become a problem.**

Transformations should be as simple and straightforward as feasible. This is mainly because of the fact that today's development environments (IDEs, Wizards, Debuggers, etc.) are much more elaborate for "traditional" development. The more work can be done "the normal way", the better.

A good example for the problem described here is object-relational mapping tools. The impedance mismatch between the OO philosophy and the relational data model is a major problem that is only now being solved really well, although the problem has been around for a while.

Therefore:

**Define a rich domain-specific application platform consisting of libraries, frameworks, base classes, interpreters, etc. The transformations will "generate code" for this domain-specific application platform.**




☆ ☆ ☆

The code generated will not just consist of "real code", but also of configuration files, deployment information and other artifacts that can be used by the DOMAIN-SPECIFIC PLATFORM. Incrementally grow the power of your DOMAIN-SPECIFIC PLATFORM (frameworks, libraries) as the depth of your understanding of the domain increases. This reduces the size and the complexity of the "framework completion code" that needs to be generated (and sometimes even hand-crafted). Transformations become less complex, which is desirable, given the limitations of today's tools.

Note that this pattern must not be overused, otherwise we are back to normal development. You should still model your business logic as far as possible with suitable DSLs and generate the implementation. Also, any kind of glue code or configuration data that is specific to the modeled application should be generated; LEVERAGE THE MODEL!

Once the application platform grows near enough to the concepts in the DSLs, the complexity of the transformations will decrease. The generator can be limited to generating repetitive "glue code". As long as tool support is still limited, this approach is very practical. In the end, this approach will allow you to use an ARCHITECTURE-CENTRIC META MODEL.

The key to application platform design is the iterative, incremental approach in the context of ITERATIVE DUAL-TRACK DEVELOPMENT. Designing elaborate frameworks up-front consistently leads to failure. Instead, small frameworks combined with code generation provide a solid base for iterative improvement. When generation gets hard to implement, usually the answer lies in improving the frameworks. Conversely when implementing framework features gets too hard, often generative techniques can provide an elegant solution. Depending on your deepening of the understanding about

the system you will refactor back and forth between DSL/generator and platform.

Note that as a consequence, you can use the core concepts of your application platform in the domain meta model. In general, a DSL and a framework/platform can be considered as the two sides of the same coin: the framework provides core concepts and functionality, whereas the DSL is used to "use" these concepts in an application-specific sense.

*The Time Conscious Objects (TCO) toolkit from SoftMetaWare [BH 2003] is a good example of how a framework and generative techniques complement each other. In this case the framework provides support for the concept of "time", and the meta model enables "time conscious classes" to be tagged at a high level of abstraction with the appropriate level of time consciousness.*

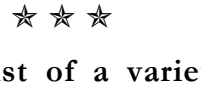
*Architecture centric MDSD as advertised by b+m and several other pragmatic MDSD people explicitly aims at representing the core concepts of the platform's architecture in the domain meta model.*

*An example of a domain-specific language that heavily depends on a rich domain-specific framework is provided in [Bettin 2002]. In this example the DSL is a visual notation for the specification of behavior in object-oriented user interfaces.*

*On the "Zemindar" project one of the authors used a DSL to enable end-user programming of complex arithmetic and statistical functions at run-time. In this case the DSL did not have to be invented, and a third-party off-the-shelf Java spreadsheet component was used as the DSL. The same project also used a model-driven generator, but it would have been completely impractical to re-implement a framework for spreadsheet functionality from scratch—or even worse, to attempt to "generate" such functionality.*

## TECHNICAL SUBDOMAINS **

You are building a large and complex software system family using MDSD



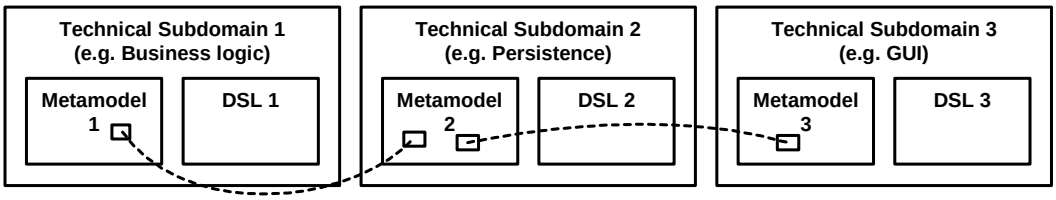
**Large systems typically consist of a variety of aspects they cover. Describing all of these in a comprehensive model is a very complex and daunting task. The model will become complicated and full of detail for a variety of aspects. Also, the DSL used for one aspect might not be suitable to describe some of the other aspects.**

Consider using a UML-based DSL to describe the application functionality (business logic). Now in addition you also have to describe persistence aspects as well as GUI design and layout. You will have to indicate persistent items in the DSL, such that appropriate code and table structures can be generated for persistence. It is very hard to put all that into the same model. A UML-based language is typically not suitable for those aspects. Trying to model GUI layout with UML is practically impossible.

Also, having it all in the same model makes maintenance complicated and prevents the efficient separation of work packages for different teams.

Therefore:

**Structure your system into several technical subdomains. Each subdomain should have its own meta model, and specifically, its own suitable DSL. Define a small number of GATEWAY META CLASSES, i.e. meta model elements that occur in several meta models to help you join the different aspects together.**




☆ ☆ ☆

This pattern is especially useful if you IGNORE THE CONCRETE SYNTAX in your transformation engine since it allows you to represent the gateway meta model elements (those that occur in several subdomain meta models) using the different concrete syntaxes of the different domains, while representing them in the same way inside the transformer (and thus providing a natural integration of the different meta models).

Note that this pattern deals with the partitioning of the system into several technical subdomains, not with structuring the whole system into different functional packages. The latter is of course also useful and should be done too.

A very specific TECHNICAL SUBDOMAIN is MODEL DRIVEN INTEGRATION. Mapping and wrapping rules can be very nicely specified using a suitable DSL. GENERATOR-BASED AOP can also be a way to handle cross-cutting TECHNICAL SUBDOMAINS.

☆ ☆ ☆

*The Time Conscious Objects (TCO) toolkit from SoftMetaWare is explicitly designed to unobtrusively fit into existing architectures as a technical subdomain. I.e. TCO assumes that a pre-existing system may*

*be based on an arbitrary object-oriented modeling language (which could be UML or plain old Java code), and the very simple DSL of TCO allows users to annotate the model with information about the level of time consciousness of objects.*

*The small components project uses a UML based DSL for specifying interfaces, dependencies, operations and components. It uses a completely different DSL based on a suitable XML DTD to define system configuration, component instance location and technical aspects configuration such as remoting middleware.*

*The DSL for specification of behavior in object-oriented user interfaces [Bettin 2002] can easily be used in combination with other modeling languages such as standard UML to specify object structure.*

## MODEL-DRIVEN INTEGRATION *

You need to integrate your MDSD developed software with existing systems and infrastructure.

☆ ☆ ☆

**Green field software development projects are rare, mostly new software is developed in the context of one or more existing systems that will still be around for a while. Additionally, often there is a desire to phase out some of the legacy systems over time, and to incrementally replace them with an implementation that better addresses business needs and that is based on a current technology stack. Integration among different – new and legacy – systems is thus part of many projects, model-driven or not.**

Depending on the integration strategy the code may need to be generated in the context of the current technology stack and/or the relevant technology stack of the system to be integrated with. Generated artifacts may also include appropriate data-conversion scripts for one-off use. Typically, integration revolves around mapping of APIs using a systematic approach, including necessary data conversions.

Therefore:

**Extend the model-driven software development paradigm to the domain of integration among software systems. Mapping information between systems is most valuable when captured in a model. Approach integration as part of MDSD, not outside of MDSD. Define a TECHNICAL SUBDOMAIN for MODEL-DRIVEN INTEGRATION. If it gets complex, consider using one TECHNICAL SUBDOMAIN per system. Define the DSLs in these domains that enable you to express the**

**mapping of relevant elements in your business domain model and the existing legacy systems. Use automation to ensure that "switching-off" of legacy systems is possible even after you've left the project.**

☆ ☆ ☆

Integration with exiting systems is a strength and not—as sometimes alleged—a weakness of a model-driven approach.

In case of integration between two separate model-driven systems, it may be beneficial to split the integration code generation between both systems such that the knowledge about the different technology stacks does not have to be duplicated in template definitions etc.

For simple integration issues a TECHNICAL SUBDOMAIN may be overkill, and it may be sufficient to use UML tagged-values or an equivalent concept in a DSL to capture the mapping between relevant elements in your business domain model and elements in existing systems. Only take this approach if this information does not clutter up and detract from the domain model, and only if the integration is between systems/sub-systems that are not legacy systems that are due to be phased out.

In particular if a legacy is planned to be phased out, ensure that integration code can easily be removed once it is no longer needed, otherwise dead code leads to architectural degradation over time. Make use of an *Anticorruption Layer* as described in [Evans]. Specify the mapping using EXTERNAL MODEL MARKINGS.

Consider automating the gradual "switching-off" of legacy systems to the degree where it amounts to identifying switched-off parts using the DSL in the relevant subdomain model. Be a good citizen and make life easy for coming generations—remember that the people who may be switching-off the last parts of a legacy system in three years may know very little about the integration code.

☆ ☆ ☆

*One of the authors has used this pattern many years ago in conjunction with LANSA RUOM to integrate for example with legacy infrastructure for security. The RUOM feature of pre-and post-template processors was essential in this context.*

## GENERATOR-BASED AOP *

You are developing a software system (family) using MDSD techniques. You generate implementation code using some kind of code generator.

☆ ☆ ☆

**In many applications, cross-cutting concerns must be handled consistently and in a well-localized manner. Programming languages do not provide support to modularize these concerns; adding another tool (i.e. an aspect weaver) is often not possible because of insufficient support, tool availability or developer skills. How can you still handle cross-cutting concerns in a consistent way?**

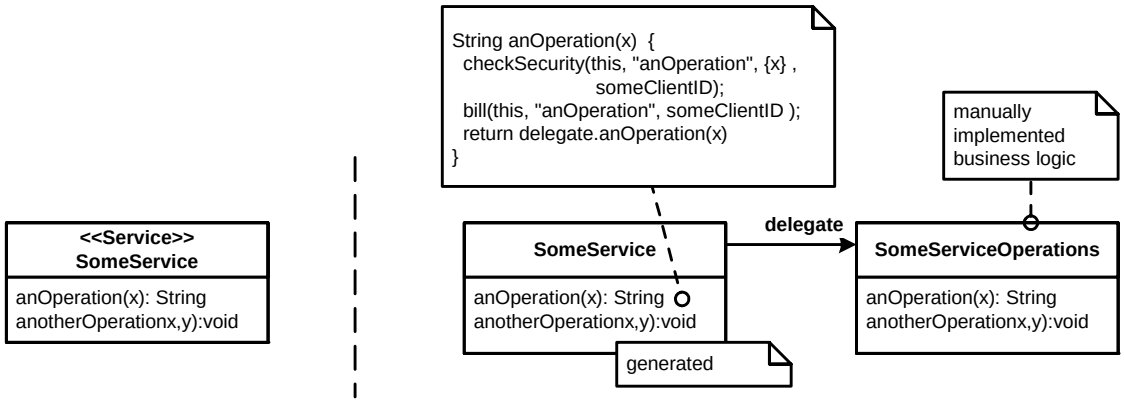
In the case of a business application that should be made available to several clients, it is often required to bill each client's use to the respective client. It is thus necessary to log the execution of each operation and determine the cost associated with the invocation.

Another cross-cutting aspect that needs to be handled in this scenario—as well as in many other scenarios—is authorization (checking whether a client has the right to access specific functionality or to read, modify, or delete specific information). A client may only be allowed to see data it "owns", and it may also be restricted to usage of a subset of the overall application functionality.

To ensure consistency, you want to make sure these aspects need not be manually handled by application developers – rather, some form of AOP should be used to handle these cross-cutting aspects in a centralized manner.

Therefore:

**Implement the handling of cross-cutting concerns with the help of the generator. You can either take advantage of the generator's integral features (e.g. consider that it generates many instances of a meta model element with the help of one transformation/template) or use the generator to implement proxies, interceptors and other AOP-addressing design patterns in the generated system. Consider the cross-cutting concern a TECHNICAL SUBDOMAIN and provide a suitable DSL for it.**




☆ ☆ ☆

As a consequence of applying this pattern, you don't have to use an additional tool (the aspect weaver) while still being able to handle cross-cutting concerns. Of course it is not possible to address all kinds of cross-cutting concerns; aspect-weavers that operate on language level such as AspectJ are much more powerful and of more general-purpose applicability. However, in many circumstances, the generator-based approach is sufficient. In addition, you always have the freedom to adapt the structure of the generator (and maybe of the application platform architecture) to allow handling of the aspects you require.

☆ ☆ ☆

*In an EJB project this approach was used to generate exactly the kind of proxy mentioned above. Dynamic security checks were implemented, as was very expressive auditing and logging.*

## PRODUCE NICE-LOOKING CODE … WHEREVER POSSIBLE **

You generate application code from models.

☆ ☆ ☆

**In many cases, the idea that developers never see generated code is unrealistic. While developers never modify generated code, they will probably see the generated code when debugging the application or when verifying the transformation engine configuration. How can you make sure developers actually understand generated code and are not afraid of working with it?**

The prejudice that "you cannot read/work with/debug generated code" is a well established one. In some settings this is even the reason why code generation, and model-driven development is not used at all. Fighting this prejudice is thus crucial.

Therefore:

**PRODUCE NICE-LOOKING CODE … WHEREVER POSSIBLE! When designing your code generation templates, also keep the developer in mind who has to – at least to some extent – work with the generated code.**

☆ ☆ ☆

There are several things you can do to make your code look nice:

- You can generate comments; in the templates, you have most if not all information available to add meaningful comments. Typically you

can even adapt comments to the generated code by templatizing comments.

- Because of typically unsufficient “whitespace management support” in many tools you have to decide whether you want to make your templates look nice, or whether the generated code should look nice. A good approach is to make sure the templates look nice and use a pretty printer/formatter tool to reformat the generated code after it has been generated. Such pretty printers are available basically for every programming language, as well as for XML, etc.
- A third very useful aspect is to include a so-called “location string” to the code generated by a particular template/transformation. This describes the model element(s) from which the particular section of code has been generated. It is good practice, especially for debugging purposes, also to include the name of the template/transformation and the "last changed" timestamp of the template/transformation used to generate the code. An example could be *GENERATED FROM TEMPLATE SomeOperationStereotype [2003-10-04 17:05:36] FROM MODEL ELEMENT aPackage::aClass::SomeOperation()*.

Using this pattern can make a big difference. It basically says that you should stick to coding conventions and style guides also in generated code. Especially, useful indenting is crucial!

Also note that if you templatize a "quality" prototype, you should already have all the comments at hand.

This pattern should really apply to generated and to hand-crafted code. In practice, all too often hand-crafted code is very messy. There is a small caveat regarding the generation of optimized code that may be required in some cases, where the results won't look nice. These cases should be explicitly identified and described – and the respective code should be separated from the rest.

*Is there a non-trivial example somewhere that says more than just “yes, we also did it?”*

## DESCRIPTIVE META OBJECTS **

You are developing a software system (family) using model driven development techniques. You generate implementation code using some kind of code generator.

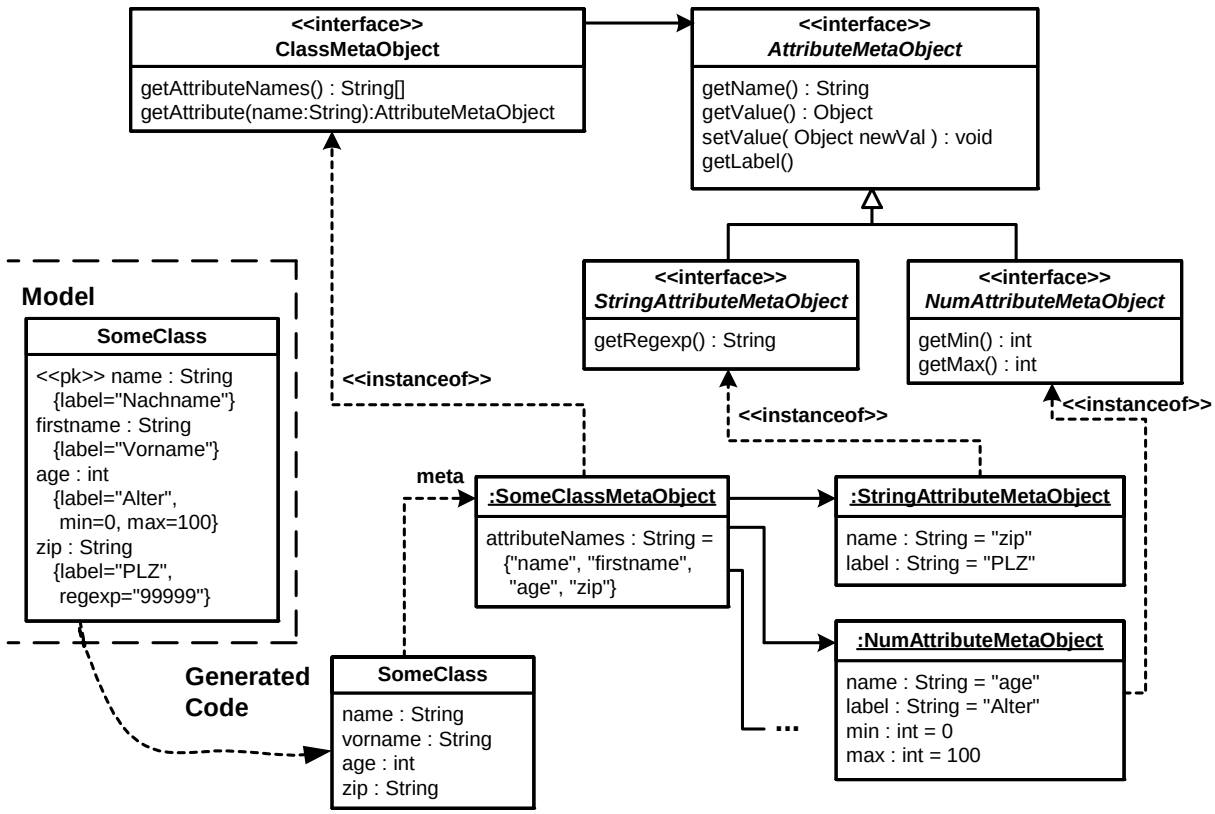
**When using a RICH DOMAIN-SPECIFIC PLATFORM for your model-driven development, the application often needs information about some model elements at run time to control different aspects of the application platform. How can you make model information available at run time and associate it with generated artifacts? How can you build the bridge between generated code and framework parts?**

Consider you want to build an application that needs to provide domain-specific logging mechanisms. The application will need to output the value of the attributes of a generated class into the log file. To make this possible, the logger needs to know the names and values of all attributes of a class. Especially in languages that don't feature reflection, you cannot easily implement such a mechanism generically.

Another problem could be that you annotate object attributes with additional information, such as a nice label, a regular expression for contents checking or min/max values for number attributes. At run time you need to be able to access this information e.g. to build a GUI dynamically. You cannot easily embed this information in programming-language native classes.

Therefore:

**Use the information available at generation time to code-generate meta objects that describe the generated artifacts. Provide a means to associate a generated artifact with its meta object. Make sure the meta objects have a generic interface that can be accessed by the RICH DOMAIN-SPECIFIC PLATFORM.**




☆ ☆ ☆

This pattern makes selected parts of the model available in the application in a native, efficient way. Another (theoretical) alternative would be to store parts of the model with the application – however, access to a complex model is typically slow, and therefore this approach is not feasible.

There are different ways of how a meta object can be associated with its generated artifacts. If the artifact is completely generated, you add a *getMeta object()* operation directly to the generated artifact. If this is not feasible (e.g. if you want to keep your artifacts free of these things) you can also use a central registry that provides a lookup function *MetaRegistry.getMeta objectFor(anArtefact)*. The implementation (i.e. the mapping) for the operations will be generated, too.

The meta objects cannot just be used for describing a program element, but also to work with it. This leads to a GENERATED META OBJECT PROTOCOL.

☆ ☆ ☆

*One of the most well-known examples of this approach is JavaBeans, where the BeanInfo class describes the Bean itself in the way described above, mainly for use by GUI tools. The only difference is that JavaBeans are note typically generated, but manually written.*

*An early OR-Mapping framework called LPF has generated meta objects that described generated relational table structures. These were used at run time to manage persistence.*

*The LANSA 4GL environment, which goes back to 1987, is based on an "active repository" that provides access to extensive meta-information about LANSA objects. The LANSA RUOM model-driven generator makes extensive use of the LANSA repository.*

*Another example is in the context of the Small Components project. This is based on C++, and the descriptive meta objects are used to "emulate" reflection. Note that direct model access would not be possible since the infrastructure is intended for embedded systems where performance and code size is critical.*

# Acknowledgements

We would like to thank our shepherd, Juha Parssinen for the useful comments on our paper, as well as the participants of the EuroPLoP '04 writers workshop where this paper was workshopped.

Thanks to Tom Stahl of b+m AG for inspiring some of the patterns and providing known uses. Thanks to Joe Schwarz and Gianni Raffi for allowing us to use the ALMA data model example.

# References

[Alexander 1977] Christopher Alexander, 1977, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press

[Beck 2000] Kent Beck, 2000, *Extreme Programming Explained: Embrace Change*, Addison-Wesley

[Bettin 2003] Jorn Bettin, 2003, *Best Practices for Component-Based Development and Model-Driven Architecture,* http://www.softmetaware.com/best-practices-for-cbd-and-mda.pdf.

[Bettin 2003b] Jorn Bettin, 2003, *Ideas for a Concrete Visual Syntax for Model-to-Model Transformations,* http://www.softmetaware.com/oopsla2003/bettin.pdf and http://www.softmetaware.com/oopsla2003/bettin.ppt.

[Bettin 2002] Jorn Bettin, 2002, *Measuring the Potential of Domain-Specific Modeling Techniques,* http://www.cis.uab.edu/info/OOPSLA-DSVL2/Papers/Bettin.pdf.

[Bettin 2001] Jorn Bettin, 2001, *A Language to describe software texture in abstract design models and implementation,* http://www.isis.vanderbilt.edu/OOPSLA2K1/Papers/Bettin.pdf.

[BH 2003] Jorn Bettin, Jeff Hoare, 2003, *Time Conscious Objects: A Domain-Specific Framework and Generator,* http://www.softmetaware.com/oopsla2003/pos06-bettin.pdf.

[Bosch 2000] Jan Bosch, 2000, *Design & Use of Software Architectures, Adopting and Evolving a Produt-Line Approach,* Addison-Wesley

[Cleaveland 2001] Craig Cleaveland, 2001*, Program Generators with XML and Java*, Prentice Hall

[CN 2002] Paul Clements, Linda Northrop, 2002*, Software Product Lines, Practices and Patterns*, Addison Wesley

[Cockburn 1998] Alistair Cockburn, 1998, *Surviving Object-Oriented Projects,* Addison-Wesley

[Codagen] *Codagen Architect*, http://www.codagen.com

[Eclipse GMT] *Generative Model Transformer project*, http://www.eclipse.org/gmt/

[Evans 2000] Eric Evans, 2003, *Domain-Driven Design*, Addison-Wesley

[GDP] b+m AG, *Generatice Development Process*, http://www.architectureware.de/download/b+m_Generative_Development_Process.pdf

[GenFW] Sourceforge.net, *openArchitectureWare*, http://architecturware.sourceforge.net

[Gentastic] Gentastic, *eGen*, http://www.gentastic.com

[GHJV95] Gamma, Helm, Johnson, Vlissdes, *Design Patterns*, Addison-Wesley 1995

[Kiczales et. al., 1991] Gregor Kiczales, *The Art of the Metaobject Protocol*, MIT Press, 1991

[Koschmann, 1990] Timothy D. Koschmann, *The Common Lisp Companion*, Wiley, 1990

[LANSA] *LANSA Rapid User Object Method*, http://www.lansa.com/downloads/support/docs/v10/lansa060.zip

[OMG MDA] *Model-Driven Architecture*, http://www.omg.org/mda/

[WL 1999] D. M. Weiss, C.T.R. Lai*: Software Product Line Engineering, A Family-Based Software Development Process*, Addison-Wesley