Patterns for Model-Driven Software-Development

Version 1.4, May 10, 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.

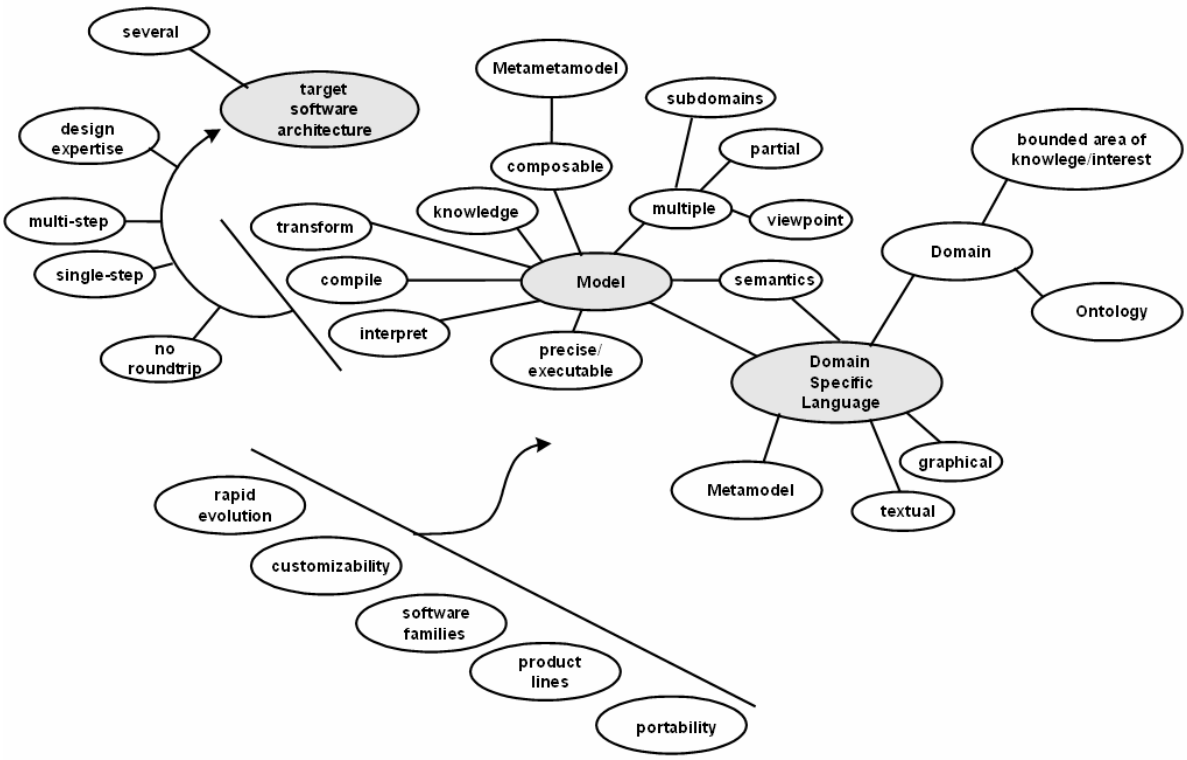
What is MDSD

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*

Until we come up with some more elaborate introduction, the following mind map should suffice to give a rough impression.



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.

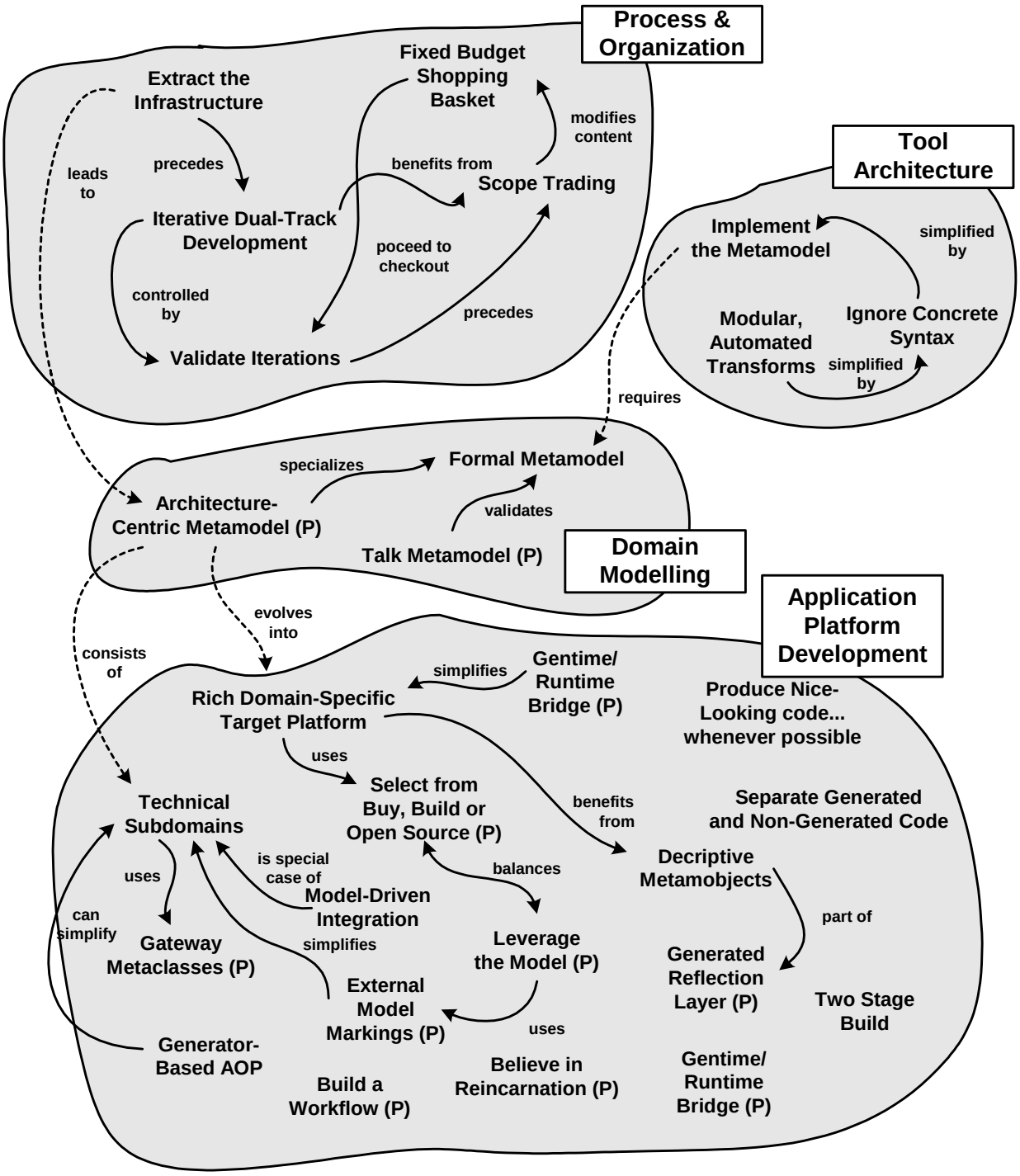
Note that, just as in Alexander's Pattern Language, we will qualify each pattern with no, one or two asterisks:

- 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.



The Patterns

Process & Organization

Model Driven Software Development is based on a clear distinction between domain engineering, i.e. designing and building an application platform and application engineering, i.e. designing and building individual applications. This separation of concerns has a long track record in companies practicing software product line engineering [CN 2002].

The relationship between application development teams and the infrastructure team that develops the application needs to be managed in the same way as the relationship between customer and application development. The infrastructure team needs on-site customers in the form of application development team representatives, otherwise there is a significant risk that the functionality delivered by the infrastructure team will not be acceptable to the application development teams. That does not mean that each application development team needs to have a permanent ambassador in the infrastructure team, it rather means that each of the application development teams need a technically competent designer who participates in a cross-team architecture group that determines the requirements for the infrastructure team. The cross-team architecture group is certainly not a novel concept, the novelty is rather in insisting on having competent designers in the application development teams. This counter-acts the tendency of all competent designers gravitating to the infrastructure team. Effective skill transfer can be achieved by short-term secondments (one iteration) between application and infrastructure teams in both directions.

As experience shows, infrastructure teams are 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 (consisting of representatives from application development teams) is given the mandate to exercise SCOPE TRADING and to VALIDATE ITERATIONS.

The architecture group representatives in the application development teams are primarily accountable to their team, and all requirements for the infrastructure need to be traceable to functional or non-functional requirements for the overall product/project.

Secondments can be used to distribute domain knowledge and technical knowledge.

This organisational structure is highly important for managing larger projects. In a small project the infrastructure team may boil down to one or two people, and the architecture group operates very informally, however is still makes sense to formally record the results of SCOPE TRADING.

The described structure of an infrastructure team and one or more application development teams is compatible with Alistair Cockburn's Crystal Orange methodology [Cockburn 1998], although Crystal Orange does not consider the specific case of a model-driven approach.

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.

☆ ☆ ☆

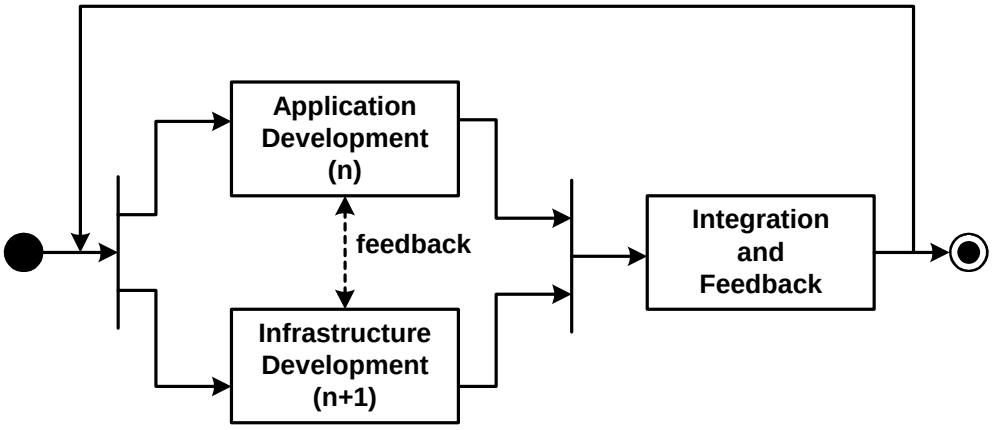
When building a new software system family, you actually have to develop two things: a concrete application as well as the MDSD infrastructure that helps you build the applications based on the family. Development of an elaborate infrastructure in parallel with application functionality can compromise 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].

FIXED BUDGET SHOPPING BASKET **

You are iteratively developing software and need to ensure that a fixed amount of money is spent wisely to build a product that meets customers' needs.

☆ ☆ ☆

Practical experience shows that large software development initiatives usually result in high risks for the customer and the development organization. In anticipation of the risks both parties attempt to mitigate the impact, the customer by insisting on a fixed price, and the vendor by building contingency into the fixed price. How do you ensure that the customer gets a working system for his fixed budget?

These simplistic mitigation strategies don't work, as the impact of the risks of large-scale software development is hard—if not impossible—to measure accurately in advance. Paying a premium for a fixed price contract does not guarantee that the delivered system will meet user needs at the time of deployment. Similarly adding significant contingency to the estimated effort does not guarantee that a fully featured system can be delivered on time and within budget.

The abstract nature of software prevents contractual details from capturing every aspect that needs to be considered in software design and software implementation to lead to a user-friendly system, i.e. a system that optimally supports users in their work. Software requirements are suitable to provide guidance in terms of scope, but they are not sufficient to guarantee a product that is acceptable to the customer.

Therefore:

Split the fixed budget over a series of iterations to determine the available resource capacity. Use timeboxed iterations of constant length to provide shippable code at least every three months. VALIDATE ITERATIONS provides the "checkout" procedure to confirm the items that meet expectations, and to put unsatisfactory items back on the shelf of open requirements. Subsequently SCOPE TRADING is used by business stakeholders in collaboration with selected end users to spend their fixed "iteration budget" to fill the FIXED BUDGET SHOPPING BASKET for the next iteration.

☆ ☆ ☆

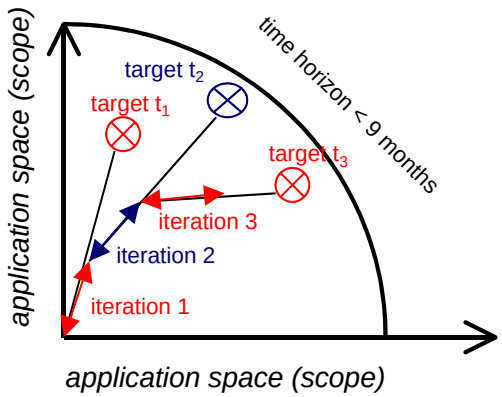
Rather than performing time consuming, repetitive reviews of requirements relating to customer needs as perceived at the inception of the project, much better results are achieved by regularly performing VALIDATE ITERATIONS to test-drive the software product under construction.

Requirements that have failed validation are annotated with clarifications, and are listed alongside any new requirements that may have emerged during the last iteration. Subsequently the development team calculates realistic prices for all open requirements to enable SCOPE TRADING.

The work of two or three iterations can be packaged and made available in the form of an intermediary release. The short release cycle makes SCOPE TRADING practical, i.e. the addition of important new requirements delays the implementation of less important features by a couple of months, and not by longer periods. The quality of a release has to be such that it is shippable, i.e. it must have passed all phases of testing. For this reason the last iteration in a release focuses on bug fixes, testing, and packaging.

The following set of rules summarize the management of releases and iterations:

- Use timeboxed iterations that are shorter than six weeks, validated by users/customers.
- Produce shippable code at least every three months
- Ideally, deploy into production every three months to get "live" feedback
- For development of new business applications, "go live" within nine months, don't risk losing the team (mother) or the application (baby) as illustrated in the diagram below.



Each point in the "application space" represents a potential system, with richness of functionality represented as the distance from the origin. In well-run iterative, and incremental development, each iteration should lead to an increase in functional richness.

☆ ☆ ☆

One of the authors started using this pattern in 1994, and has since then refined this pattern into its current form, and applied it in a large number of software development projects in various industries and involving a wide range of technology stacks. This pattern achieves an appropriate distribution of rights and responsibilities between the customer and the software developer. All projects where this pattern has been used—both MDSD and traditional projects—were highly successful. The pattern has even been applied to fixed-price projects such as the "Zemindar" project, which effectively became fixed-budget, variable-scope projects, which becomes evident.

SCOPE TRADING **

You need to develop a (project) plan for the next iteration consisting of a fully prioritized list of requirements that are scoped for the next iteration.

☆ ☆ ☆

The content of each iteration needs to reflect the priorities of requirements in terms of relevance to the business, and also needs to include items that are of critical architectural significance. Each iteration needs to deliver functionality that can be validated by end users and stakeholders in a meaningful way, and at the same time architecturally significant items cannot be overlooked. If new requirements are allowed to be raised during iterative development, how do you manage scope and prevent scope creep?

The primary intention of an iterative approach is to enable validation and allow early feedback to influence the direction of the project. It makes little sense to measure progress against a fixed list of requirements that were

drawn up at the start of the project. Instead, VALIDATE ITERATIONS determines how close the project is to meeting the needs of users and stakeholders at regular intervals.

It is not uncommon to see "iterative" development without direct stakeholder feedback. Although it still allows to eliminate architectural risks as early as possible, it means that iterative development falls far short of its potential, and—from the perspective of the customer—it is much like classical waterfall development, with a big surprise at the end.

Therefore:

New requirements raised are not allowed to enter a "running" timebox. At the beginning of each iteration, use a scope trading workshop to agree the scope of each iteration and capture the outcome in an iteration plan. Ensure that not only end users but also other relevant stakeholders are present at the workshop so that priorities can be agreed. Formally document results of the workshop, and then proceed within the timebox based on the priorities defined in the scope trading workshop to ensure that estimation errors don't affect the most important requirements and items of critical architectural significance.

☆ ☆ ☆

The duration of a scope trading workshop can vary, usually it takes 1 to 4 hours, depending on size of the project. The scope trading workshop occurs after VALIDATE ITERATION, as soon as indicative estimates of the effort for the remaining open requirements have been compiled by the development team based on velocity figures from the last iteration. Agile methodologies such as Extreme Programming explain how to pragmatically measure velocity and estimate requirements. Alternatively there are similarly pragmatic approaches based on use case points. We refer to [Beck 2000] for the details of estimating the effort associated with individual requirements.

The scope-trading workshop involves the project manager, the architect, key stakeholders, and selected end users. The objectives for the workshop are simple:

- Assigning a strict sequence of priorities to requirements items (stories, use case scenarios, features—whatever is used in the methodology used in the project) by the user/stakeholder community.
- Identification of all requirements that are of architectural significance for the next iteration(s), that need to be included in the coming iteration.

As a guideline, no iteration should be filled entirely by architecturally significant items, unless some of these items coincide with those that are at the top of the list of user priorities. At the beginning of a project, EXTRACT THE INFRASTRUCTURE means that significant amounts of architectural work need to be done, which goes hand in hand with the well-known practice to build a “technical prototype” at the beginning of a project. Over the course

of several iterations, the number of architecturally significant items should decrease, such that the stakeholder/user community drives the direction of development. Once a development organization has a good MDSD infrastructure for a certain domain, new projects will start with user requirements right away.

All items that don't fit within the timebox are moved onto the list of open requirements for following iterations as candidate scope items. At this point, end users and stakeholders have a last chance to trade scope for the coming iteration; however changes should be minimal if the preceding prioritization exercise has been performed properly.

If end users can't agree on a strict sequence, the stakeholder(s) can determine an arbitrary sequence. This sets clear expectations of what would get dropped off in next timebox in case of changes in scope or resource limitations.

Scope trading is scalable to product development involving multiple customers with different agendas, where every customer is contributing different amounts to the available product development budget. In this case an open and fair voting process can be established by giving every customer a set of "priority points" in proportion to their financial contribution, and then allowing every customer to allocate "priority points" to the items on the product roadmap. Arbitration of priorities can occur in an anonymous format facilitated by the product vendor, or even an independent third party. The facilitator's role is restricted to iteratively allowing customers to revise their priority point allocations, until no customer desires further adjustments. The concept can be accelerated or timeboxed by the provision of electronic collaboration tools similar to the tools used to implement electronic auctions.

One of the authors has been applying this pattern consistently over many years, from small three-month web application development projects to large-scale product development in the enterprise applications market, where tier-1 products are sold for several million dollars. The pattern gives stakeholders the ability of introducing new requirements and priorities at the end of each iteration, thus providing them with a very effective tool to steer the project. For some stakeholders it comes as a surprise that identifying the items at the bottom of the list of priorities is just as important as identifying the items at the top of the list of priorities. In the "Zemindar" project and in several other projects this pattern allowed the project to deliver a working system by a fixed date.

VALIDATE ITERATIONS **

You want regular confirmation that the project is on track, and confirmation of the items that meet customer expectations, and items that need further refinement or rework.

☆ ☆ ☆

End users need the ability to test drive software-under-construction, and users as well as stakeholders need the ability to provide useful feedback to the development team. Although *on-site customer*[1] ensures that requirements can be clarified on a daily basis, and that selected end users test-drive software-under-construction continuously, scope within a timebox needs to remain fixed, progress needs to be tracked at regular intervals and stakeholders need to confirm the accuracy of the direction proposed by end users.

End users and business experts may identify and articulate new requirements at any point in a project, but the scope of a timebox needs to be kept stable. Although changed requirements necessitate a re-estimation of the effort by the development team the development team should not be distracted during an iteration. Additionally, priorities need to be realigned on a regular basis without causing churn in the development process.

Overall, end-of-iteration activities that follow the timebox, including re-estimation of effort should be packaged into the shortest possible schedule, so that further development can continue in a new timebox with new priorities as soon as possible.

Therefore:

A timebox is concluded with a formal iteration validation workshop to confirm progress and to document the parts of the software that are acceptable to users and stakeholders. Let an end user that acted as the on-site customer drive the demonstration of implemented features. Explicitly communicate to the end user and stakeholder community that new requirements can be brought up at any point during the workshop. Encourage exploration of "what-if" scenarios—stakeholders may develop a new idea while watching the demonstration, and similarly the architect may want to use the opportunity to raise issues that may have escaped the requirements elicitation process and that have been uncovered by the development team.

☆ ☆ ☆

[1] *On-site customer* is one of the core Extreme Programming practices that relates to the agile principle of *Business people and developers must work together daily throughout the project.*

VALIDATE ITERATIONS and *on-site customer* are highly complementary, if one of the two is not done, then there is a significant risk of failure of iterative development. *On-site customer* is about enabling iterative, incremental user acceptance testing, and VALIDATE ITERATIONS is about formally confirming user acceptance test results.

Results of the validation workshop are documented in an Iteration Assessment that consists of

- A list of features that the customer has accepted
- A list of outstanding requirements
- A list of re-work items

After a validation workshop, and prior to SCOPE TRADING, high-level requirements specifications need to be updated and revised such that requirements size estimation is possible.

The total duration of the end-of-iteration activities including SCOPE TRADING varies from less than a day of up to five days in the extreme case—the less time is used the better, however the level of discipline cannot be compromised. The amount of effort involved depends on the amount of requirements covered, complexity of the functionality under validation, and on how quickly consensus between stakeholders and amongst the end users can be reached.

Ideally an end user drives the presentation of the system, explaining new functionality to his peers. Alternatively a business analyst may drive the presentation.

Independent of the main presenter, a facilitator familiar with the software-under-construction is required to reach agreement on whether specific implemented features satisfy requirements or require modification to be acceptable to the customers. The facilitator also acts as the scribe and documents all features and parts of features that the end users accept as presented. Features may be accepted subject to minor modifications that need to be documented in the workshop. The deficiencies of features not acceptable to the end users or to other business experts in the audience need to be documented by the facilitator.

In distributed projects that span time zones and geographies, validation workshops may need to be conducted using electronic means such as video/voice conferences coupled with web-based tools that allow the sharing an application across several sites. In the scenario of a large number of stakeholders (customers), validation workshops may need to be performed involving representatives from a product user group and selected customer representatives. Additionally, there may be a requirement to conduct more than one workshop with different audiences—in which case care has to be taken not to hold up further product development by dragging out the end-of iteration activities.

One of the authors has been applying this pattern consistently over many years in many projects. The pattern has been introduced very successfully into several organizations that were initially skeptical of the value of timeboxed iterative software development.

Based on practical project experience, the first validation workshop can be somewhat of an anti-climax for users and stakeholders for two reasons. Firstly the validation workshop uncovers misunderstandings, which raises questions about the capability of the software development team and the software development process. Secondly, not much visible functionality is available at the end of the first iteration, as much of the initial effort is used to eliminate architectural risk and to build "behind the scenes" infrastructure. It is the second iteration workshop that usually wins the customer's confidence, where the customer sees that feedback provided in the previous workshop has been fully addressed.

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.

☆ ☆ ☆

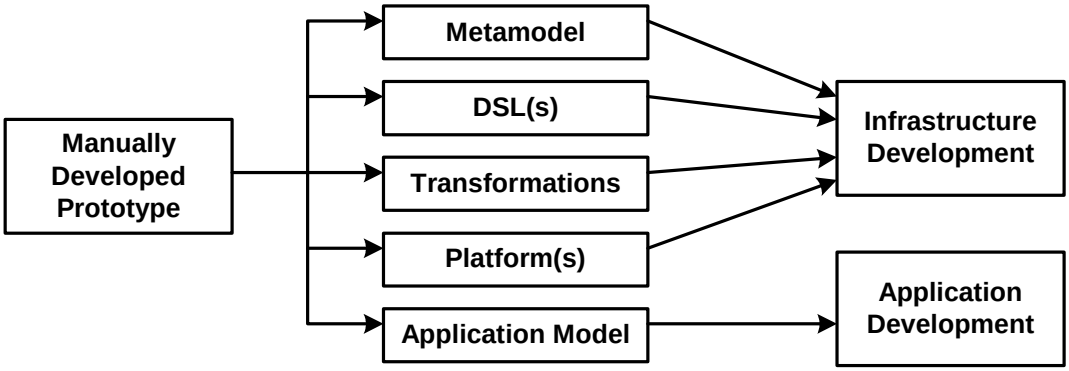
At the beginning of an MDSD project you often don't know how to get started. You know, that you should use ITERATIVE DUAL-TRACK DEVELOPMENT, but how do you get started, anyway? You want to have at least one running application as fast as possible.

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].

BUILD A WORKFLOW (P)

Formalize the development process, for which workflow is a good paradigm. Make it changeable and agile. Having a documented workflow process helps to build team routines. Nothing helps more in keeping schedules and deliver quality than having ingrained routines.

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.

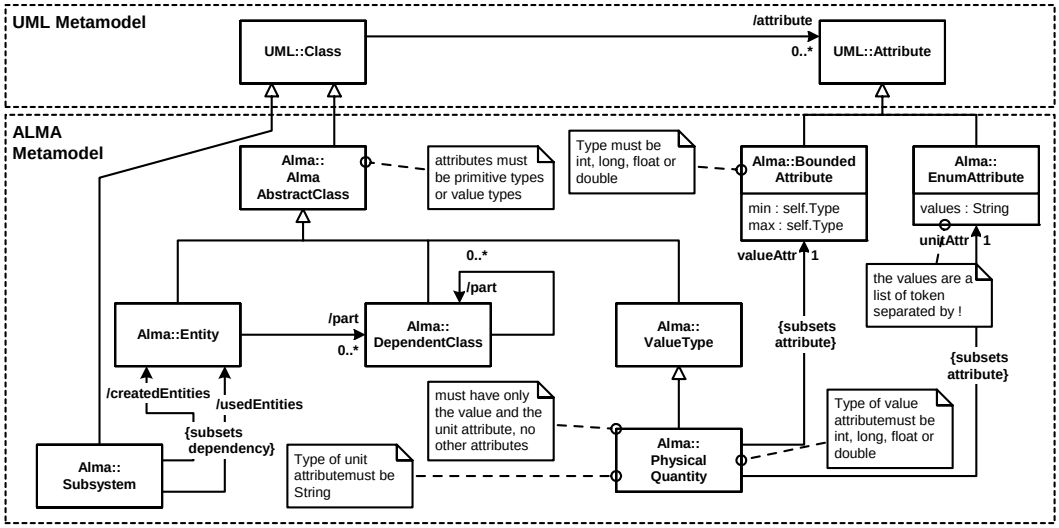
When building the meta model, make sure you understand your domain. Building a glossary or ontology as a first step can help. Of course, the meta model is defined incrementally, using ITERATIVE DUAL-TRACK DEVELOPMENT.

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

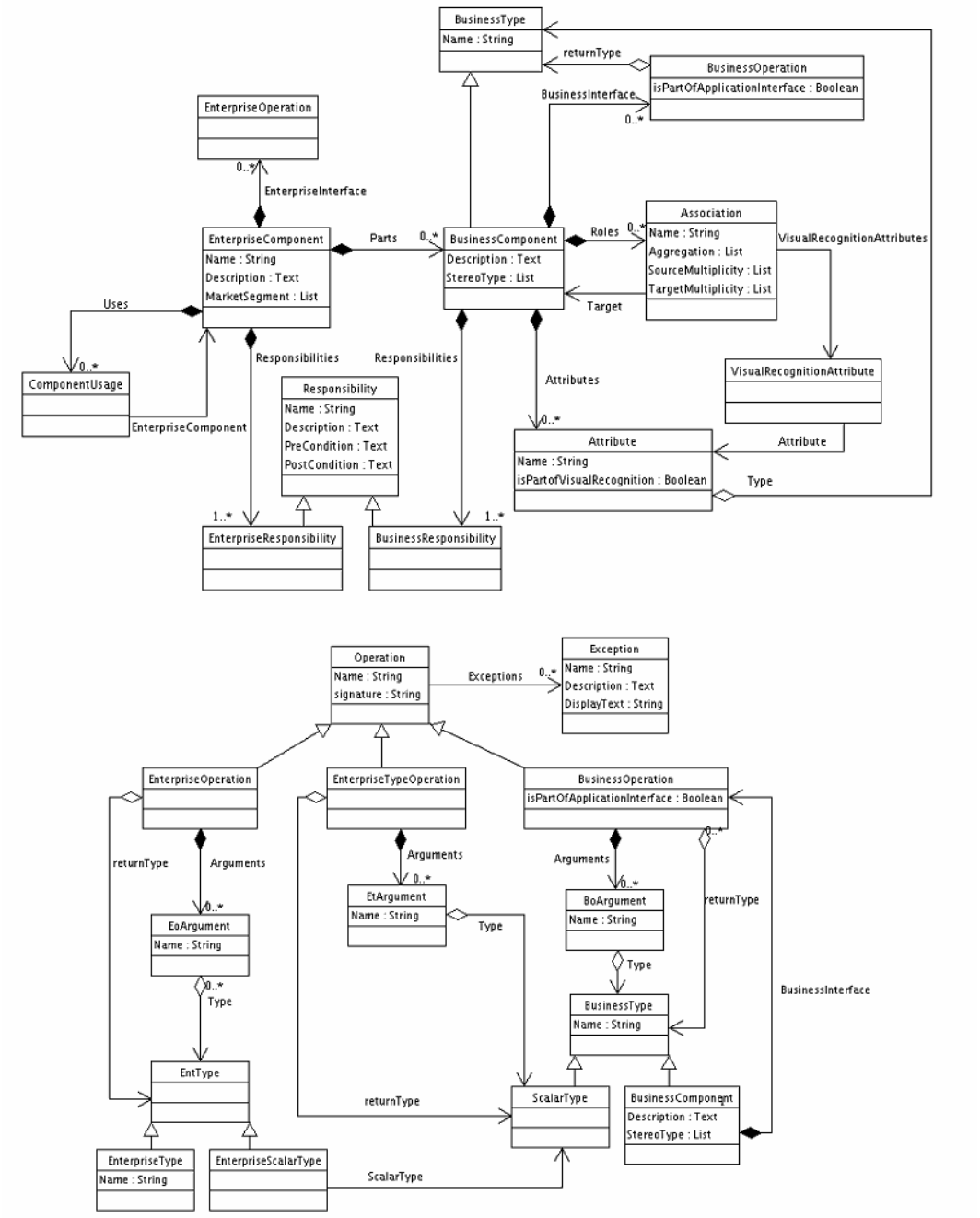
METAMODEL – itself the basis for domain-specific tool support. However, you still need to verify that the metamodel actually represents the real-world domain correctly; you may want to TALK META MODEL to do this.

☆ ☆ ☆

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.

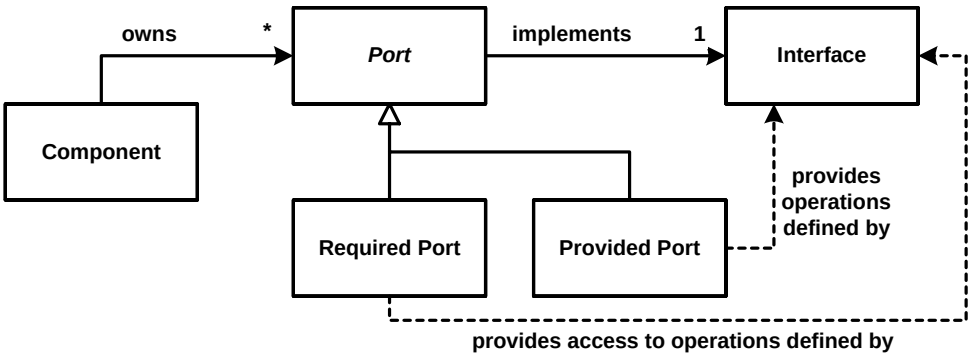


Another situation where a formal meta model really helped was while writing a book on model-driven development. We wanted to sort out the commonalities and differences between MDSD, MDA, GP and architecture-centric MDSD. The breakthrough came when we came up with a FORMAL META MODEL *for the domain of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

TALK META MODEL (P)

It is important to continuously improve and validate the FORMAL META MODEL for a domain. It has to be exercised with domain experts as well as

by the development team. In order to achieve this, it is a good idea to use it during discussions with stakeholders by formulating sentences using the concepts in the meta model.



Consider the example meta model above. When talking to stakeholders, use sentences like

- A component owns any number of ports.
- Each port implements exactly one interface.
- There are two kinds of ports: required ports and provided ports.
- A provided port provides the operations defined by its interface.
- A required port provides access to operations defined by its interface.

As soon as you find that you cannot express something using sentences based on the meta model, either a) you have to reformulate the sentence, b) the sentence's statement is just wrong, or c) you have to update the meta model.

This technique is a well-known technique in domain modeling and relates to the *ubiquitous language* pattern in Eric Evan's book [Evans 2000]. The book contains many other useful techniques – after all, meta-modeling is also some kind of modeling.

ARCHITECTURE-CENTRIC META MODEL (P)

In case you cannot come up with a sophisticated, domain-specific meta model you can represent the concepts of the technical platform directly in the meta model. This results in architecture-centric model-driven development and in relatively simple transformations. Over time, more and more domain-specific concepts can be added to the meta model, making models more expressive, less dependent on the technical platform, but requiring more complex transformations.

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?

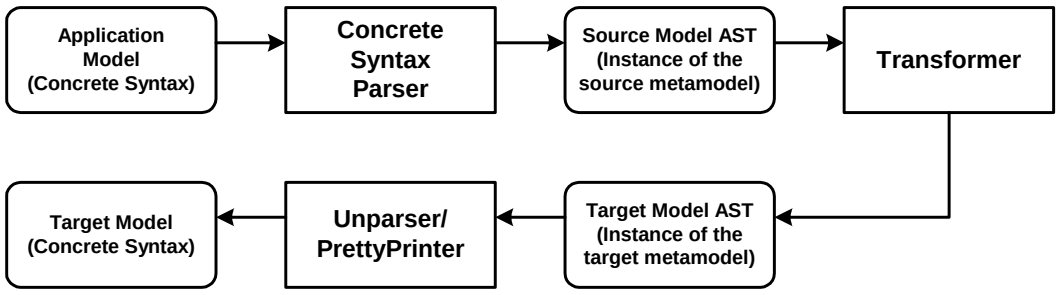
Definition of transformations based on the concrete syntax is typically a very error-prone an inefficient task. Consider using XMI. XMI is a very complicated syntax. Defining transformations based on XMI (and maybe XSLT) is not very efficient.

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**



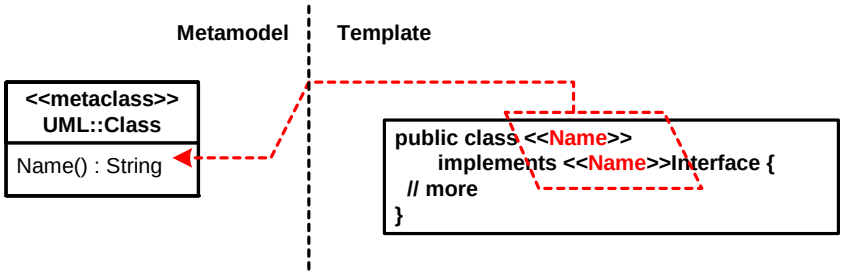
☆ ☆ ☆

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.

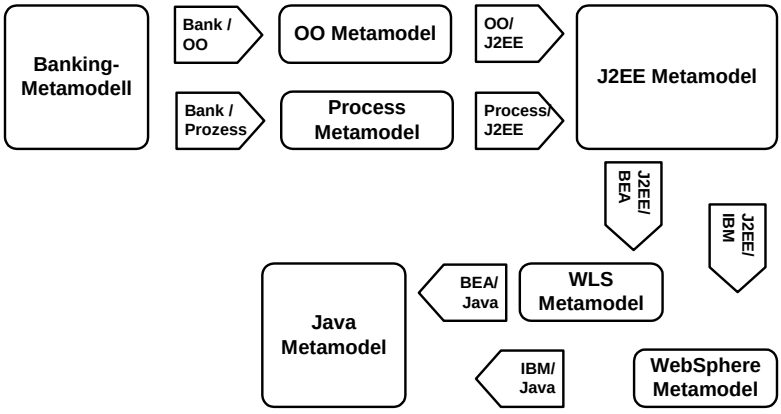
Revisiting the b+m generator framework, we already saw that it represented the applicable meta model as Java classes. The transformations are template-based (since it generates code directly). These templates can contain statements that reference the meta model and its properties. You do not see anything 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.

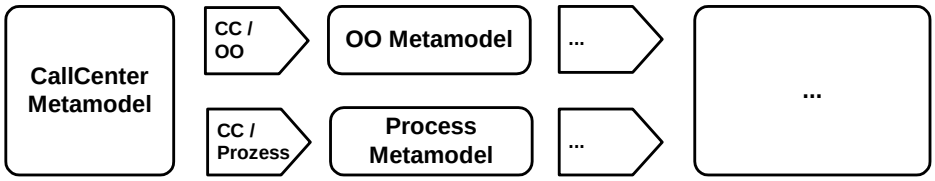
As far as we know, most of today's MDA tools rely on non-standard template languages for the mapping of models to textual artifacts such as code. Limitations of these template languages and issues with proposed alternative approaches are sketched in [Bettin 2003b].

MODULAR, AUTOMATED TRANSFORMS (P)

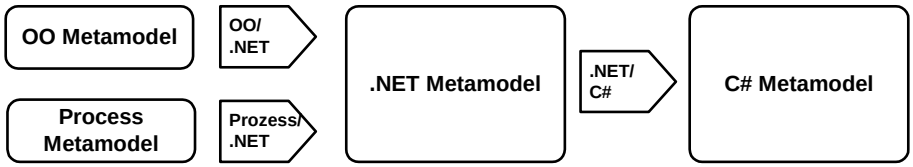
In order to more easily reuse parts of a transformation, it is a good idea to modularize a transform. Consider the following example. There we generated J2EE code from a banking-specific model.



Imagine now we want to generate J2EE code for call center applications. Since the transformations are modular, we just have to exchange the first part. All the work that went into the subsequent transformations can remain unchanged.



Finally, if we want to port both software system families to .NET, we just have to exchange the backend.



If you had a only one single, direct transformation, this kind of reuse would not be possible. Note that in contrast to the OMG, we do not recommend looking at, changing or marking the intermediate models. They are merely a standardized format for exchanging data among the transformations. If we need model markups to control or configure some or all of the transformations, these should be supplied as EXTERNAL MODEL MARKINGS.

TRANSFORMATIONS AS FIRST-CLASS CITIZENS (P)

Transformations are an important asset in MDSD. It is not appropriate to consider them "second thought". You have to refactor them, document them, version them, think about how you can merge different branches, etc. In short, you should consider transformations to be first class citizens and treat them accordingly.

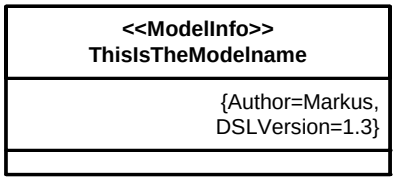
ASPECT-ORIENTED METAMODELS (P)

If you implement the metamodel, make sure you don't mix problem-domain metamodel, stuff with solution-space utility functions. Separate them out of the metamodel using AO techniques.

DESCRIPTIVE INFORMATION IN MODELS (P)

In order to facilitate versioning and multi-version generators, you have to make sure, the generator knows for which version of the DSL a particular model has been built. To make this possible, embed this descriptive information in the model; this might also include author, release status, etc.

In DSLs using custom syntax, adding this information is trivial. In UML it is not as trivial, at least if you want to keep the approach portable. The following approach works portably in all tools: add a class with a reserved stereotype that has tagged values for the you want to represent. The following is an example.



Application Platform Development

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?

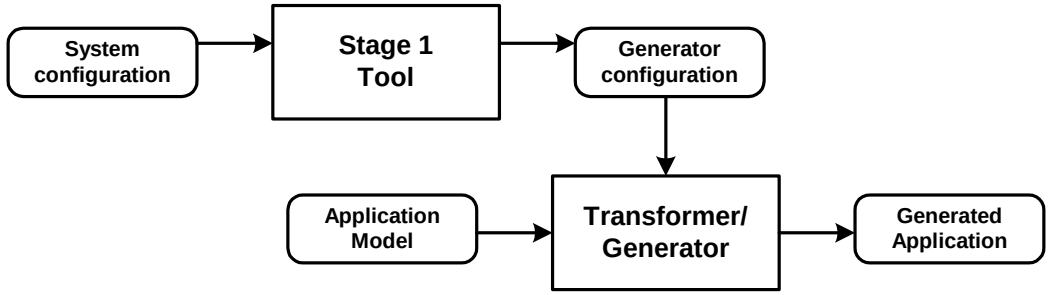
Consider the selection of a target platform. Depending on the platform, different transformations must be executed. The selection of the platform thus determines which transformations to execute. It is very hard to

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.

☆ ☆ ☆

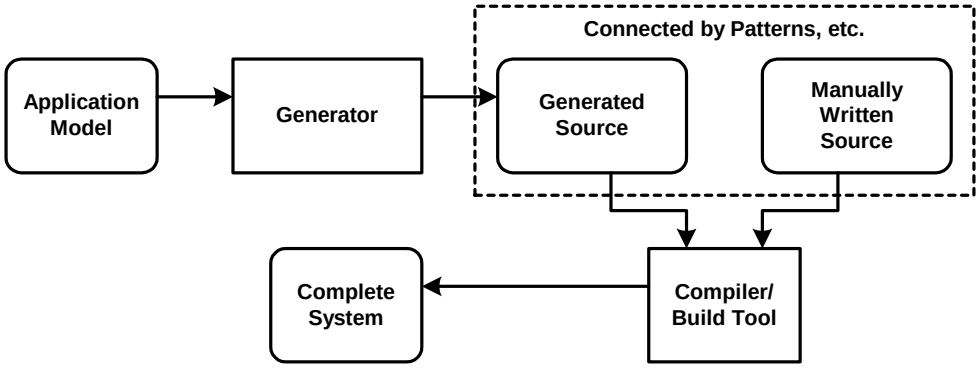
If only parts of the application is generated, "blanks" must be filled-in by manual programming. However, modifying generated files by adding non-generated code creates problems in the areas of consistency, build management, versioning and overwriting of manually written code when regenerating.

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.

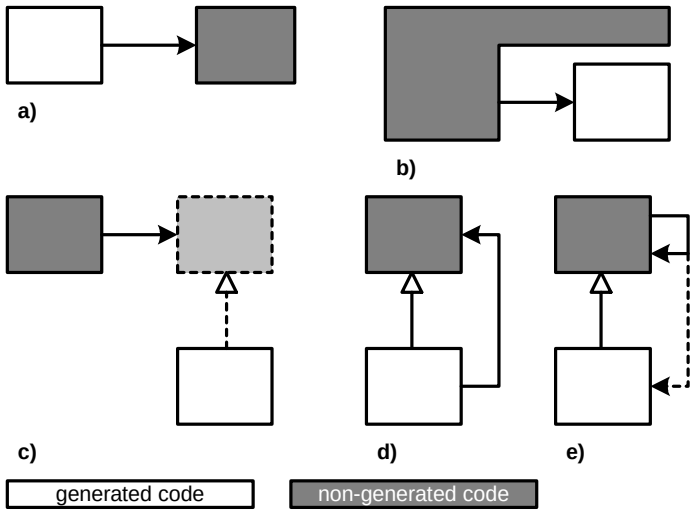
On the downside, this approach sometimes requires a bit more elaborate design or some more (manual) programming.

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.

☆ ☆ ☆

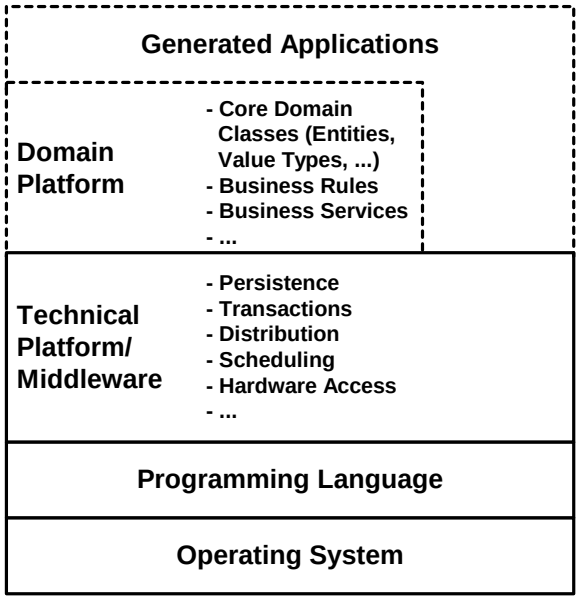
In the end, application models must be transformed to a certain target platform to be executed. The bigger the difference between the domain concepts and the target platform, the more complex the transformations have to be. With today's tools this can become a problem.

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

☆ ☆ ☆

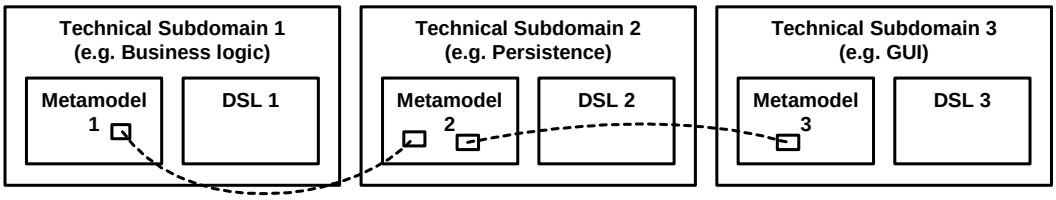
Large systems typically consist of a variety of aspects they cover. Describing all of these in a comprehensive model is a very complex and daunting task. The model will become complicated and full of detail for a variety of aspects. Also, the DSL used for one aspect might not be suitable to describe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

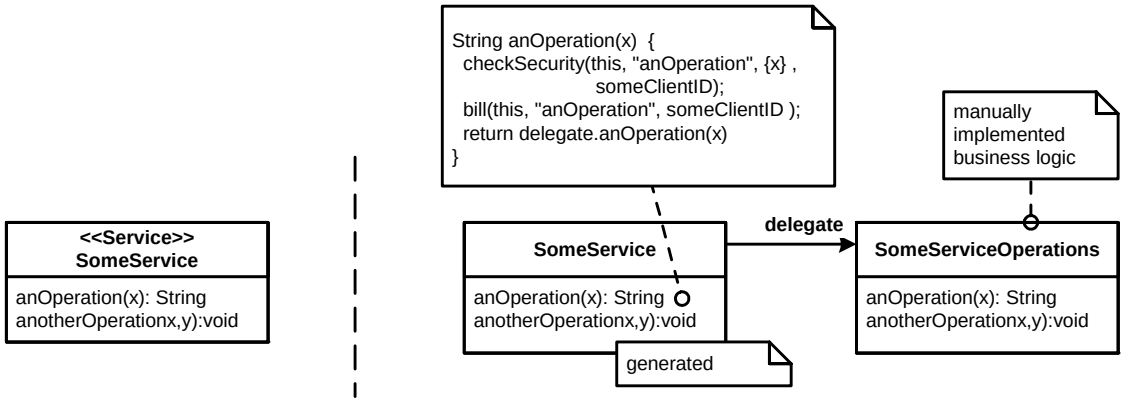
In the case of a business application that should be made available to several clients, it is often required to bill each client's use to the respective client. It is thus necessary to log the execution of each operation and determine the cost associated with the invocation.

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.

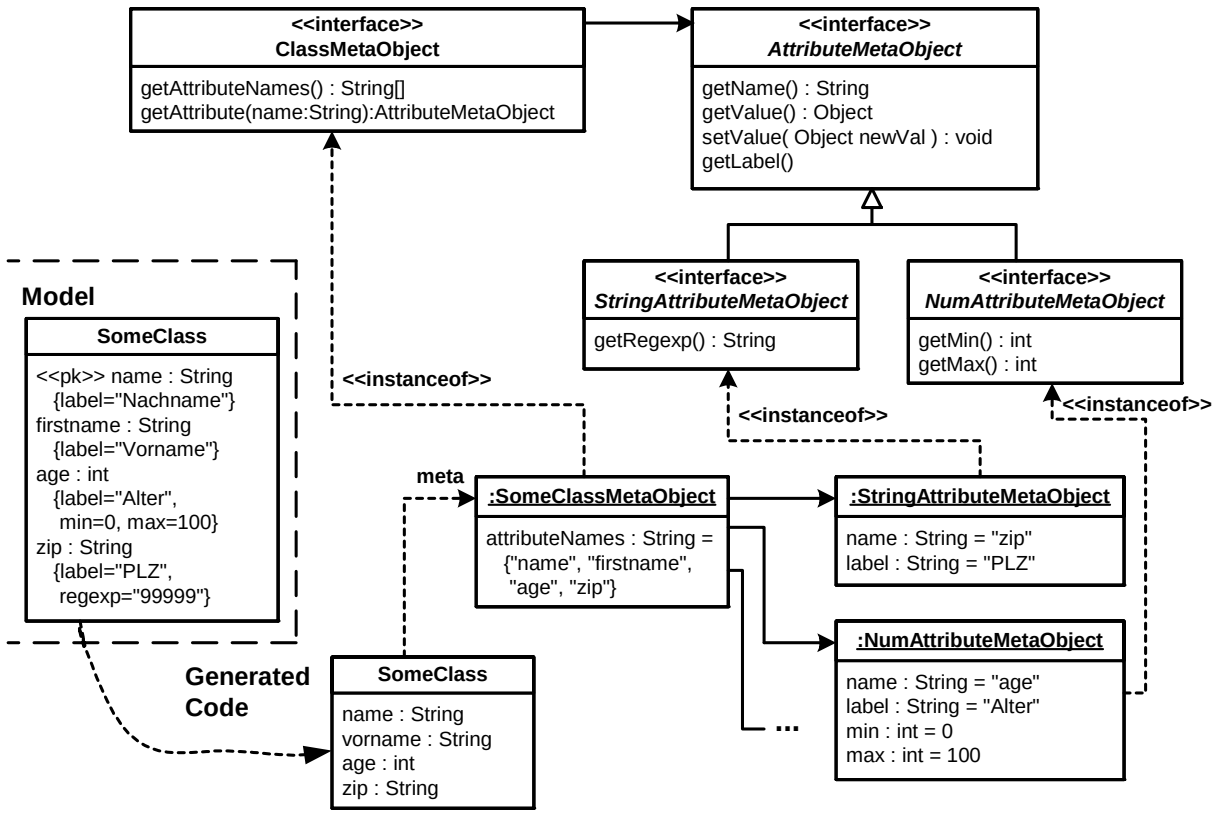
When using a RICH DOMAIN-SPECIFIC PLATFORM for your model-driven development, the application often needs information about some model elements at run time to control different aspects of the application platform. How can you make model information available at run time and associate it with generated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

FRAMEWORK/DSL COMBINATION (P)

A framework provides a set of services. These are often hard to use, since the programming language does not "know" the framework, the compiler cannot provide any support. A DSL that allows to "program against" the framework can solve this problem.

EXTERNAL MODEL MARKINGS (P)

In order to allow the transformation of a source model into a target model (or to generate code) it is sometimes necessary to provide "support" information that is specific to the target meta model. Adding these to the source model "pollutes" the source model with concepts specific to the target model. MDA proposes to add "model markings", but this currently supported well by only very few tools. Instead, we recommend keeping this information outside of the model (e.g. in an XML file); the transformation engine would use this auxiliary information when executing the transformations.

GENTIME/RUN TIME BRIDGE (P)

In many cases, a particular model feature or system configuration item affects not only what is generated, but also the behavior at run time. For example during the model-driven development of the ALMA data model, we had several data encodings in the VOTable XML standard. Dependent on the encoder definition in the EXTERNAL MODEL MARKING, we had to specify the encoding format in the generated XML. Also, we had to do the encoding of the actual values at run time, depending on the very same setting in the markings. The solution to this is to have different encoder classes, one for

each encoding strategy, and have the code generator generate the instantiation of the respective object into the source code. Using the strategy pattern can help to integrate with non-generated code (which needs to have some kind of interface to use, independent of the specific encoding strategy used).

GENERATED REFLECTION LAYER (P)

Metaobject protocols as described for example in [Kiczales et. al, 1991] are a means to introspect, modify and reify metaobjects of a language. This is typically done dynamically (e.g. in languages such as CLOS [Koschmann, 1990]). In the context of MDSD, you can provide at least a read-only-MOP that allows you to introspect classes, as well as dynamically invoke operations. A generic interface allows clients access to any kind of class:

```
public interface RClass {
    // initializer – associates with base-level object
  public setObject( Object o );
    // retrieve information about the object
  public ROperation[] getOperations();
  public RAttribute[] getAttributes();
    // create new instance
  public Object newInstance();
}
public interface ROperation {
    // retrieve information about op
  public RParameter[] getParams();
  public String getReturnType();
    // invoke
  public Object invoke(Object params)
}
public interface RAttribute {
    // retrieve information about op
  public String getName();
  public String getType();
    // set / get
  public Object get();
  public void set( Object data );
}
```

Since these operations are generic, workbenches or other dynamic tools can use this kind if reflective interface to work with the data.

GATEWAY META CLASSES (P)

Using TECHNICAL SUBDOMAINS typically results in having different meta models as well as different concrete syntax for the different subdomains. For example, you can describe the workflow of an application using activity diagrams and the layout of the presentation layer using a textual, XML-like language. If you want to generate a useful system from these different specifications, your generator needs a mechanism to get from one model to

the other (such as moving from an *activity* to its associated UI form). To make this possible, you need two things:

First, you need model elements that are present in the meta models of both TECHNICAL SUBDOMAINS. If you IGNORE CONCRETE SYNTAX in your generator, these GATEWAY META CLASSES can be represented with different concrete syntax in the different TECHNICAL SUBDOMAINS.

The second thing you need is a common meta meta model. Since a generator tool will always use some kind of (maybe implicit) meta meta model. In order to represent both meta model elements in the same generator-internal model representation, they need to use the same meta meta model. The b+m generator, for example, requires Java classes to be used as the meta meta model for all meta models.

THREE LAYER IMPLEMENTATION (P)

When generating code for object-oriented targets, you often have to mix platform code, generated code and manually implemented code. A good solution to this is: Have an abstract base class in the platform, generate a – still abstract – subclass, and implement the app logic in yet another, now non-abstract, subclass class. For example, this allows FORCED PRE/POST CODE.

FORCED PRE/POST CODE (P)

In many situations, you need to make sure that some generated code is always executed before/after some manually implemented code. You have to make sure, it is not possible for developers to circumvent this. Consider an a vehicle class, which has a drive(driver:Person) operation, whereas a driver's age needs to be over 18 years old. The following implementation idiom can be used:

```
public interface IVehicle { // public interface for vehicles
  public void drive(driver: Person);
}

public abstract class VehicleBase implements IVehivle { // generated
  public final void drive(driver: Person) { // final; cannot redefine
    if ( !(driver.age() >= 18) ) throw new ConstraintViolated();
    driveImpl();
  }
  protected abstract void driveImpl(driver: Person);
      // protected; cannot be called by clients
}

public class VehicleImpl extends VehicleBase { // manually implemented
  protected void driveImpl(driver: Person) {
    // do whatever it takes…
```

```
  }
}
```

BELIEVE IN RE-INCARNATION (P)

The final, implemented application should be built by a build process that includes re-generation of all generated/transformed parts. As soon as there is one manual step, or one line of code that needs to be changed after generation, then sooner or later (sooner is the rule) the generator will be abandoned, and the code will become business-as-usual.

INTER-MODEL INTEGRATION WITH REFERENCES (P)

Often you have several sub-models that together constitute the complete application model, for example because you use several DSL for different application aspects or because you partition a big application into several parts. To do this,

- Create proxy metaclasses for each metamodel element that should be referenced
- Upon instantiation, the proxy automatically finds its referenced object – or throws an exception, if it cannot find it.
- As usual, proxies are subclasses of their respective referenced type. Operations forward to the operations of the referenced element.
- These proxies can be generated automatically.
- The generator does not see the difference between the proxies and the actual element.

LEVERAGE THE MODEL (P)

The information captured in a model should be leveraged to avoid duplication and to minimize manual tasks. Hence you may generate much more than code: user guides, help text, test data, build script content, etc. Find the right balance between the effort required for automating manual tasks and the effort of repetitively performing manual tasks by considering "sustainability" in the Extreme Programming sense. As a rule-of thumb: compare the number of keyboard strokes and user gestures required to get to a given result, and select the method that is the least painful and labor-intensive. Make use of SELECT FROM BUY, BUILD, OR OPEN SOURCE in your assessment. Often others have already done the hard work, and you can save

a lot of typing—not to mention the thinking effort going into reinventing the old wheel!

BUILD AN IDE (P)

Model-Driven approaches can result in accidental complexity for application developers due to the fact that many artifacts are generated, and depend on each other. This contradicts, to some extent, the goal to make domain experts use this infrastructure. You have to make things easier for users of the MDSD infrastructure. A proven approach in doing to is to implement a project- or domain-specific IDE. Frameworks such as Eclipse provide an excellent basis.

USE THE COMPILER (P)

When combining generated and non-generated code you can often run into consistency problems between the (new version of the) generated code and the (not-yet-adapted) non-generated code. In order to make sure inconsistencies are detected, use the compiler to help you spot it. For example,

- If you generate abstract classes with abstract methods, and overwrite them in subclasses then the compiler will report an error in case the signature is changed in the generated code (since the subclass does not overwrite the method anymore)
- If you expect manually implemented classes to have certain operations, and for some reason, you cannot enforce these operations by providing a super-interface, etc. you can generate helper class calls all these operations in some dummy method. This class will never be used at runtime; it is just compiled with the rest of the system to "raise" compiler errors.

SELECT FROM BUY, BUILD, OR OPEN SOURCE (P)

Don't be blinded and ignore the potential of well-proven off-the-shelf products and robust Open Source infrastructure that is used by thousands of organizations! Software development organizations are famous for not-invented-here syndrome. The selection between buying, building, and using Open Source needs to be driven by sustainable economics in the same way as LEVERAGE THE MODEL. I.e. don't compare costs over the short term, take into account the longer-term picture, and factor the cost of capital into your calculations. Considering the maintenance burden, is it really worth

developing infrastructure and applications that don't constitute the unique core of your business? However, do LEVERAGE THE MODEL and don't compromise hard-earned domain knowledge that has gone into your domain-specific frameworks and generators by replacing them with unrefined and blunt off-the shelf tools.

Consciously differentiate between

- **strategic software assets**—the heart of your business, assets that grow into an active human- and machine-usable knowledge base about your business and processes,
- **non-strategic software assets**—necessary infrastructure that is prone to technology churn and should be depreciated over two to three years, and
- **software liabilities**—legacy that is a cost burden.

Strive to evolve your core business systems into strategic assets that increase in value over time through (re)use and refinement. Widely used and respectable Open Source software should also be treated as a strategic asset—in this case a public asset. Commercial 3rd party software largely falls into the category of non-strategic assets: usually the software is not built entirely on open standards, and you also cannot assume that the product will be supported indefinitely. Hence investments in commercial software should be treated as capital investments affected by depreciation.

- Be honest and don't portray all your legacy systems as strategic assets in the sense described above, identify the liabilities and strive to replace them by a combination of strategic assets and non-strategic assets.
- You need to differentiate between the 3rd party products and your organizations' information that builds up in the databases of such products. A subset of your information is a strategic software asset, and needs to be treated as such. The same can be said about legacy software that has become a liability: the applications have become a liability but some of the information managed by these applications constitutes a strategic asset.

Unless continuous attention is being paid to the quality of strategic software assets, these can quickly degenerate into liabilities, hence *refactor mercilessly.*

As appropriate, use Open Source infrastructure to reduce risk exposure in terms of vendor dependence. When evaluating Open Source software, carefully consider the Open Source licensing model attached to the software: some licenses allow Open Source software to be built into non-Open Source commercial products, whereas others only allow usage in products bound to the same Open Source license.

Note: a healthy mix of strategic and non-strategic assets may sometimes only include a small core of strategic software assets. Non-strategic assets have a

permanent role in supporting the business, in some respects they can be considered as necessary consumables.

Acknowledgements

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. We also would like to thank Ghica van Emde Boas for supplying several proto-patterns.

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