A Lightweight Model Driven Development Process based on XML Technology (Draft)

Gábor Guta[*]
Gabor.Guta@risc.uni-linz.ac.at

Barnabás Szász
bszasz@gmail.com[†]

Wolfgang Schreiner[*]
Wolfgang.Schreiner@risc.uni-linz.ac.at

March 18, 2008

[*]Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, http://www.risc.uni-linz.ac.at

[†]Faculty of Informatics, University of Debrecen, Hungary, http://www.inf.unideb.hu

Abstract

Model Driven Development and domain specific languages attract the attention of the industrial practitioners. Recentely also more and more tools have become available to support these. Unfortunately these paradigms are typically discussed in a the frame of the water-flow development process, which does not fit for small and mid sized agile teams.

To fill this gap, this article presents a lightweight, iterative, model driven software development process which was implemented and tested in industrial projects. After a short summary of the state-of-art of that field, we present the process in an abstract form. Then we give a detailed description of the actual realization based on XML technology. Finally we describe how the explained process and technology was applied in a real-world project.

Contents

1 Introduction

In this paper we describe a new lightweight model driven process and a supporting XML-based MDD tool that we have developed and used in a real-life project. We explain them in the context of the latest development of that field.

Model driven development (MDD) [35], software product lines, domain specific languages (DSL), and other generative approaches of software development receive special attention both from the research community and from the industrial practitioners [8, 36, 38]. There are plenty of tools for MDD, DSL and code generation available for diverse platforms. These technologies are different in approach and have emphasis on different issues, but all of them are trying to solve almost the same problem: to generate executable code from an abstract domain model. The widely accepted enterprise software architectures require huge amount of code and configuration, which is in most of the cases error prone and boring to write by hand [18]. The interest in the community to use code generators indicates that it is useful for the current architectural practice. There are lot of terms to denote these approaches. The Object Management Group (OMG) calls its standard *Model Driven Architecture* (MDA) [31, 30], while Microsoft uses the terms *Software Factory* and *DSL* [17] for its solution. In this article we refer to these technologies in the general sense as *Model Driven Development* (MDD).

In OMG's MDA, the domain model is represented in the *Unified Modeling Language* (UML). Typically this is not pure UML, but extended with some extra notations called *UML Profiles*. MDA calls the initial model *Computation-Independent Model* (CIM) and defines two intermediate models: the *Platform Independent Model* (PIM) and the *Platform Specific Model* (PSM). In MDA the PSM is at the same abstraction level as the artifacts.OMG also defines its own transformation languages and interchange format. Tools supporting the standards are already available.

Microsoft's *Software Factory* [17] initiative has different emphasis: it provides a toolkit that makes it efficient to create one's own domain notation and it provides some tooling to make code generation easier.

1.1 Model Driven Development

MDD can generally be represented as shown in Figure 1. It starts with an abstract *domain model* from which different *generated artifacts* like source code are created by *transformation*. Transformations are also called *templates*, if the results are generated artifacts and not models. The MDD process can contain intermediate models with corresponding transformations. Models are typically represented

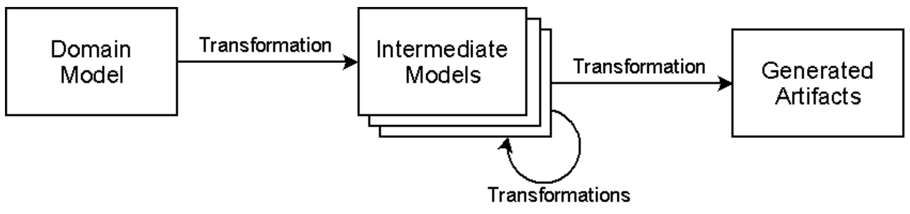


Figure 1: Model Driven Development

in different abstraction levels. *Meta-models* can be assigned for the domain model and the intermediate models, which describe their meaning and the allowed constructs. The *domain model* can model a real life domain like a business process or can be an implementation model like a website navigation flow.

Generally speaking, the different MDD approaches have to deal with the following questions:

- What is the domain model (real life/implementation)?
- How can the domain meta-model be represented?
- What kind of notation do we use to communicate the domain model?
- How are the transformations defined?
- Do we allow to edit the intermediate models or the generated artifacts?

We can ask further important questions which are typically overlooked by the modeling community:

- How does MDD affect the development process?
- Are we able to reverse the transformation (propagate changes of the generated artifacts back to the original model)?
- How can intermediate models be merged if the original generated model was modified and a different one was generated after the domain model was changed?
- What are the quality implications of the MDD? (How does it modify the test and review procedures? What kind of additional problem can be expected?)

1.2 Problem Description

In practice, an important issue is how one can integrate a new technology to existing software processes. For small or mid size project teams with limited resources this issue is even more critical.

Most of the resources available for practitioners focus on the technological aspects [3]. Books about this topic [14, 19, 17, 31, 30] also give no or little hint about how MDD can be integrated the software development process. The process, if mentioned, is explained in the context of technological steps [23]. On the other side, proposed methodologies target enterprise environments with sophisticated, customizable methodologies [11, 15, 29, 16]. A case studies like [26] even explicitly states that the MDA technologies are appropriate only for large organizations. That opinion is also agreed by certain practitioners [39].

1.3 Approach

To successfully apply a new technology in one's project there are several other factors that are independent from theoretical soundness of the technology. The learning curve of the technology, the maturity of the tools, or the adaptability of the transformations to special requirements are probably the most crucial properties. We give here a short overview about these contributing factors in small and mid sized development projects.

A lightweight approach for model driven development have to deal with the following issues:

- Most of the risks should be mitigated: The technology should not be based on immature tools and the introduction of the technology should always leave open the possibility to fall back to the traditional methodology without MDA tools.
- The approach should not delay the schedule in general. It should have immediate business advantage.
- The approach should be cost sensitive. Purchases of a high cost MDA tool set and extensive training of experts is typically not possible.
- The domain-model should be kept as simple as possible and the generated code should be human-readable and modifiable.
- The results should be reusable in other projects.

In the following sections we describe one possible solution to deal with the mentioned issues. First, we describe our lightweight MDD process and show how this process can be supported by an XML-based tool. Both process and tool were developed in the end of 2006 and have been in use since that time. We present them here in a form that is the results of several improvements according to the project experiences. Then we describe how the process and the tool were applied in one of the projects. Finally, we share our best practices which we consider necessary to carry through a successful MDD project.

2 The Process Aspect

In this section we describe the process that we developed to help fulfill the requirements described in Section 1.2. Our process description focuses on the model driven aspects and presents the process in its abstract form.

One of the key ideas behind our approach is that we explicitly support the partial usage of MDD. Our process lets one consider in which aspects of the project the use of MDD will pay off and employ it just for these. The advantage of that approach is that the domain model can be easily kept focused and abstract. On the other side, keeping track of which artifacts are generated and which are not requires only minimal administrative overhead.

The process is defined as an extension of iterative methodologies in general [27]. It fits well in existing methodologies like eXtreme Programming [9] or the Rational Unified Process [25].

2.1 Process Definition

In our development process we define two phases: the *initial phase* and the *development phase*. The *development phase* can be repeated arbitrarily many times. We introduce our own terms for the phases, because different methodologies define varying numbers of phases and different terms and we do not want to stick a particular methodology [28, 24]. For the same reason we name our set of roles, artifacts, and activities distinct from those of other methodologies. We mark those definitions with a star which are supposed to defined by an other particular methodology. On first reading, one may directly proceed to Section 2.1.4 and return to Sections 2.1.1 - 2.1.3 for definitions of the terms.

2.1.1 Roles

We define a role called *model driven development expert* (MDD expert). The responsibility of this role is to deal with the activities related to model driven technologies and support the managers, architects and developers. Naturally, in case of bigger teams this role can be decomposed into more specific roles.

2.1.2 Artifacts

We use the following artifacts in the process description as it was defined in Section 1.1: *domain meta-model*, *domain model*, *generated artifacts*, and *templates*. We define some further artifacts as the following:

- *A MDD vision* is a document containing the strategic design decisions related to the MDD. It specifies which aspects of the requirement are implemented with MDD technology. It also prescribes the abstraction level of the domain model and its scope.
- *A code generation tool tool* is a collection of programs and scripts which runs the templates on the models.
- *A domain test model* is a special model which serves as a test for the generation process. It is an artificially created small model with good meta-model coverage. It is used by the developers and it does not represent any real life model.
- *A MDD environment* is a collection of a domain meta-model, a domain test model, and a Code Generation tool.
- *Hand crafted software* (*) is software or a collection of software fragments created with some traditional software development methodology. It becomes available short before the end of a phase. It can be a design prototype, a part of an internal release, or a part of a beta release.
- *Released software* is the software resulted by the integration of *hand crafted software* and *generated artifacts*.
- *A software requirement specification* (or *requirements in short*) (*) is an evolving document containing the requirements. In practice, other methodologies define other documents as well.

2.1.3 Activities

The following terms for the activities are used in the process:

- *Domain model extraction* is the activity which extracts the domain model from the requirement specification and form other documents. During this process the *MDD vision* and the initial *domain model* are created.
- *The MDD environment setup* is the activity when the *code generation tool* is set up according to the MDD vision and the initial domain model. As part of this activity, the initial version of the *domain meta-model* and the *domain test model* are created.
- *Domain Modeling* is an activity by which the *domain model* is extended and refined.
- *The MDD environment development* is the activity by which the *MDD environment* is further developed. This development activity is typically extensive and not corrective.
- *Template development* is the activity by which the templates are extended and refined. This activity usually happens in three steps: the required functionality is implemented in a software prototype; the implementation is extracted and to edited into the *templates*; finally, the they are tested with the help of the *domain test model.*
- *Code generation* is the activity by which the *generated artifacts* are created with the *templates* from the *domain model.* The consistency of the domain model is always verified prior to the code generation.
- *Integration* is the activity by which the generated code and the hand-crafted software are merged. After the merge, the integration test is carried through.
- *Development* (*) is the activity when new functions are designed, implemented and tested. This activity is defined in more detail by particular methodologies.
- *Requirements Refinement* (*) is the activity when requirements are refined and extended.
- *Architectural prototyping* (*) is the activity by which the architecture of the software elaborated. In the end of this activity the architecture of the software is designed and a prototype which contains the initial implementation of that architecture created.

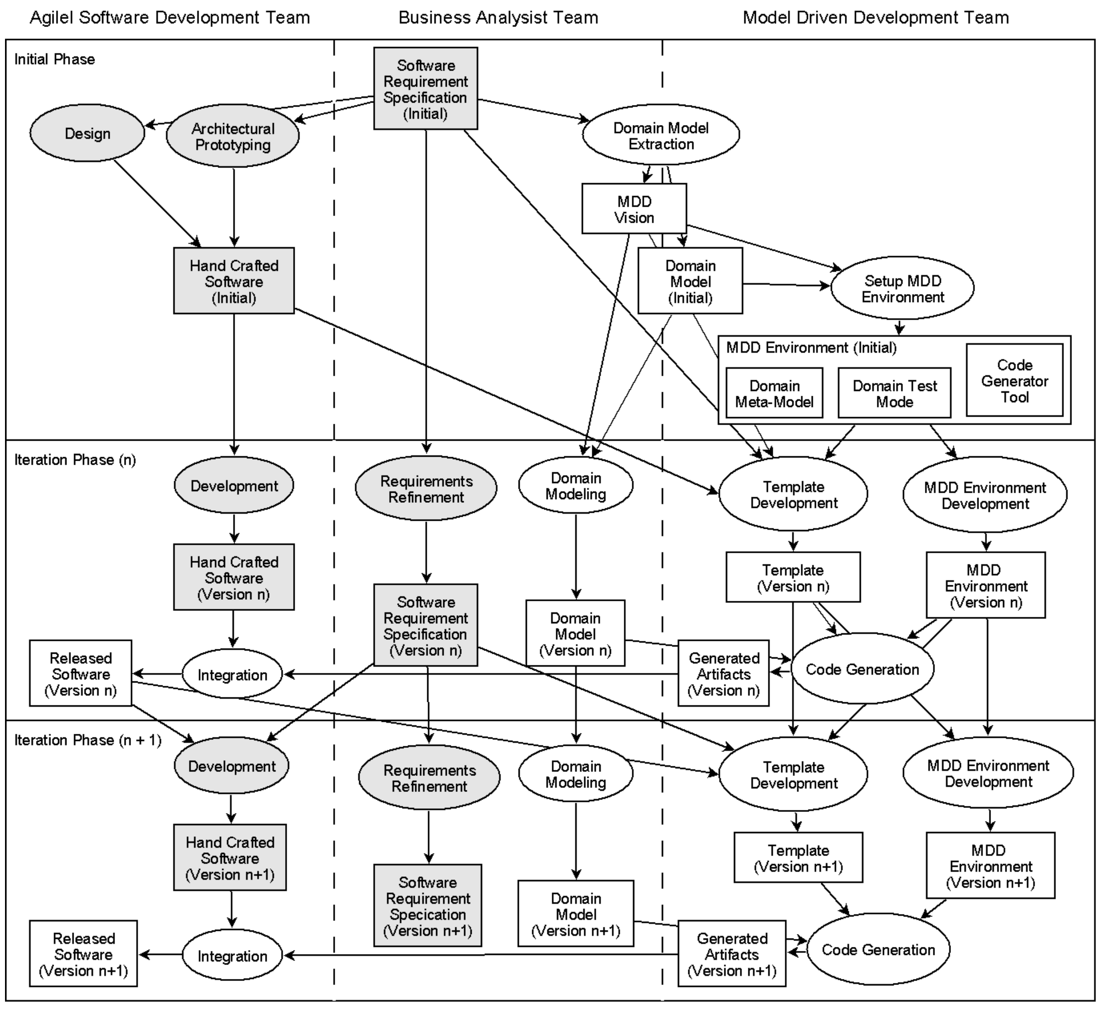
- *Design* (*) is the activity when all kind of design activates are carried through.
- *Development* (*) is the activity when all development related activates (class design, coding, and testing) are carried through.

2.1.4 The Process

Figure 2 gives an overview of the process. The artifacts are represented by boxes and the activities are represented by ellipses. Shading of the shapes means that they can be replaced by other activates or artifacts that are specific to a particular methodology and have the same purpose in the process as we prescribed (they are just included to help the understanding of the process). The flow chart is divided by dashed lines into three ”swim-lanes”. The ”swim-lanes” represent three different internal team, namely: *agile development team*, *business analyst team*, and *model driven development team*. The horizontal position of the activities express which internal teams they belong to. If an activity or an artifact is positioned between two swim-lanes, then it is a joint effort of the two teams.

- The *agile development team* can include roles like *developer*, *designer*, or *tester*. This team is responsible for the development of all non-generated software (functionality which is not represented in the domain model). The non-generated parts of the software can be the following ones: larger special components with custom functionality that are added only in a single situation or components that are used globally in the application like utility classes or authentication mechanisms.
- The *business analyst team* can include role like *business analyst* or *developer* who are responsible for business analysis activities. This team is responsible to keep in touch with the customer and is also responsible for the requirements and the domain models.
- The *model driven development team* includes *MDD experts* and developers. This team is responsible for the development of the MDD environment and the templates.

The vertical axis in Figure 2 denotes the time and the horizontal lines represents the borders of the phases. The figure shows the initial phase and two iterations such that the connections between phases are depicted. The vertical positions of the objects express how they are related in time. The arrows also provide information about the dependency of two actions. If two objects are connected and they have the same vertical position, they follow each other immediately. In detail, the process proceeds as follows:



Agilel Software Development Team
Business Analysist Team
Model Driven Development Team
Initial Phase
Software Requirement Specification (Initial)
Design
Architectural Prototyping
Domain Model Extraction
MDD Vision
Hand Crafted Software (Initial)
Domain Model (Initial)
Setup MDD Environment
MDD Environment (Initial)
Domain Meta-Model
Domain Test Mode
Code Generator Tool
Iteration Phase (n)
Development
Requirements Refinement
Domain Modeling
Template Development
MDD Environment Development
Hand Crafted Software (Version n)
Software Requirement Specification (Version n)
Domain Model (Version n)
Template (Version n)
MDD Environment (Version n)
Released Software (Version n)
Integration
Generated Artifacts (Version n)
Code Generation
Iteration Phase (n + 1)
Development
Requirements Refinement
Domain Modeling
Template Development
MDD Environment Development
Hand Crafted Software (Version n+1)
Software Requirement Specication (Version n+1)
Domain Model (Version n+1)
Template (Version n+1)
MDD Environment (Version n+1)
Released Software (Version n+1)
Integration
Generated Artifacts (Version n+1)
Code Generation

Figure 2: The Process

- The process starts with an initial phase (which can be preceded by phases defined by the methodology that our process extends). There are three groups of activities relevant in this phase:
  - The *business analysts team* and the *MDD team* extract the domain model by a joint effort. Domain experts from the customers can also contribute.
  - The *agile development team* elaborates the architecture of the software. By the end of the initial phase they ship a sufficiently stable prototype to the *MDD team* which starts to develop the transformation.
  - The *MDD team* sets up the initial version of the *code generator core*. This team also has to be familiar with the technology environment used by the *agile software team* in order to be capable to take over the architecture from the prototypes.
- During the iteration, the following activities are carried through in parallel:
  - The *business analyst team* refines and extends the domain model. The changes in the domain model do not affect the developers, so the model can be changed until the *code generation* activity starts. This team keeps in touch with the customer; its responsibility is to handle the change-requests of the requirements.
  - The *agile development team* develops the non-generated part of the software. The customization of the generated parts and the integration of the custom components are also the responsibility of this team. In the end of the iteration this team receives the generated artifacts and carries through the integration:
    * the generated artifacts are collected together with the hand crafted software;
    * in case an artifact generated in the previous iteration is modified by the agile development team, then this modification is applied to the newly generated artifact too;
    * the whole software is built and finally tested.

    There are always cases when the newly generated artifacts need manual modifications or interface mismatches have to be corrected. These cases can be resolved in three different ways:
    * The hand crafted software is modified by *agile software team*.

    * The *MDD team* solves the problem in the next iteration; a temporary workaround is provided by the *agile software team*. If it is not possible to solve the problem, then the templates are fixed by the *MDD team* and the process is repeated starting with the code generation activity.
    * The problem can not be solved; consequently the automatic merge does not become possible again. In that case, the affected artifacts have to be merged in every iteration manually, if the corresponding part of the domain model is changed. The business analyst team keeps track of these artifacts and tries to avoid any changes to them. It is crucial to keep these special cases minimal and freeze the corresponding parts of the domain model as early as possible.

  To ensure the success of the integration also integration tests and regression tests are carried through. Critical bugs are corrected during the integration and the whole test activity is repeated.

- The *MDD development team* is busy with the *template development* and the *MDD environment development*. In the end of the iteration this team produces the generated artifacts from the domain model with the help of the latest MDD environment and templates. The enhancement of the MDD environment is usually orthogonal to the template development.

  During the iteration two different versions of the MDD environment are used. The latest version from the stable branch is available at the beginning of the iteration used for the template development and a separate development branch is used for the MDD environment development. As the iterations make progress, the MDD environment needs less and less modification; then the *template development*, in which new features are added to the templates, becomes the main activity. This will be explained in detail in Section 2.1.5.

2.1.5 The Template Development Activity

The *template development* is the most complex and critical activity of the process. Typically for an iteration of the process several features are planned to be implemented iteratively in the frame of this activity. The implementation of a feature starts with its design, then the feature is coded and tested. During the implementation the generated development artifacts are extended.

Generated development artifacts are parts of the runnable application which is generated by the latest version of the templates, of the code generation tool, and of the domain test model. The implemented feature is extracted from the source code and inserted into the templates with the necessary modifications. The generated development artifacts are re-created by the modified templates. The result typically contains small errors, which can be detected when the result is built and tested. The test is carried through with the test cases of the original feature implementation. The templates are modified as long as no more errors occur during the test of the generated development artifacts. Then the feature is considered complete and the team moves on to implement the next one.

If a problem is recognized with the code generation tool, then it can be corrected in the frame of the MDD environment development. The tool used in the template development can be updated if the fixes are ready or if a new version is released during the MDD environment development.

The first iteration differs sightly from the other iterations, because at that point of time there are no prior generated development artifacts available. Before the MDD team starts to implement the first feature, it has to extract the utility artifacts, which is necessary to build and run the generated development artifacts. These artifacts can be kept separate or they can shared with the agile development team. In both cases these artifacts have to be stabilized and frozen as early as possible; if change is necessary, the update must carried through in a planed and controlled manner.

3 The Technology Aspect

In this section we show a particular XML-based code generation tool that we designed and implemented to support the development process explained in Section 2. First we describe how the components of the XML technology fit to general MDD. Then we present the architecture of our tool. We will not disclose any details about the domain model to protect the business interest of our customer.

3.1 XML and MDD

XML technologies can be used efficiently to implement one's own code generation tool. In this subsection we go through the elements of MDD as mentioned in Section 1.1 and explain which XML technology is appropriate to implement each element.

The domain model is usually described by graphs. The graphs are represented as trees with pointers [EXPLAIN], which allows to store the graph in XML format.

OMG's MDA uses this approach for its XML based interchange format named XMI [7]. Moreover, a tree based representation of the domain model and its transformation fits better to problems with small and medium complexity [21].

The syntax of the domain meta-model can be expressed in the XML Schema language [10]. XML Schema defines a grammar that expresses which XML structures are valid. However XML Schema is not expressive enough to formulate all properties, e.g. it can not say "If an *N* type node is referenced by an attribute *A* of an *M* type node, then the *M* type node must have an attribute *B* whose value is *V*". This type of properties can be checked with the help of Schematron [6] or of XSLT (see below). More details on the expressive power of XML Schema and related notations can be found in [32].

XSLT is the most well known standardized transformation language for XML. It is capable to express all kinds of transformations [20] and is designed for the efficient description of templates [22]. The main features that make it efficient for this kind of task are: basically it is a declarative (functional) language, but it can also be used in *rule-matching* style (constructs `template` and `apply-templates`) and *imperative* style (constructs `call-template`, `for-each`, and `if`); it is capable to output the results into different files; it supports output in both plain text and XML format.

XML technologies are a good choice, because they are standardized and well supported by the industry. Consequently, it is easy to find trained developers and one can expect long-term support. Furthermore, there are various kinds of tools available, e.g.: editors for XML, XML Schema, and XSLT and engines and debuggers for XSLT. Most of these tools have been available for almost a decade, so they can be considered mature.

XSLT was already used successfully by others for model transformation or for code generation [37]. A detailed comparison of XSLT with 3GL programming languages can be found in [34]. Borland's *Together 2007 DSL Toolkit* uses a special extension of XSLT to generate source code from models [2].

We have evaluated several alternative tools, before decided to build our XML-based solution. The two most promising ones were AndroMDA 3.2 [1] and Iron Speed Designer [4]. AndroMDA is an open source MDA tool supports the Java platform. This tool uses the XMI output of an UML editor and its code generator is written in Java. Although the tool initially seemed promising, we estimated that it would need too much effort re-target it to our domain model and .NET platform. On the other hand, Iron Speed Designer is a commercial tool which first generates an ASP.NET web application from an existing database schema and then allows a GUI based customization of the generated application. Also this tool did not fit to our process, because we required an editable domain model from which the

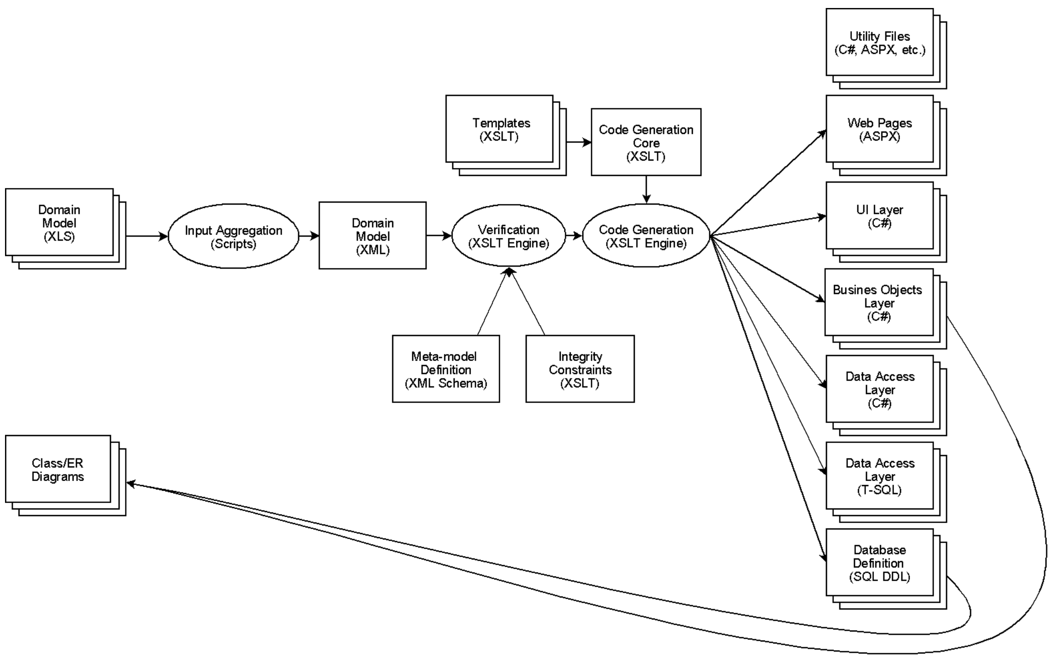


Figure 3: The Architecture of the Code Generation Tool

application could be generated. The architecture of the generated application did also not meet our requirements.

3.2 The Code Generation Tool

Figure 3 shows the architecture of our code generation tool. The ellipses indicate three processing phases and the rectangles represent the artifacts which are processed or produced. The purpose of the code generation tool is to generate a runnable ASP.NET 2.0 web application from a given domain model. The artifacts that are used to specify the domain model can be seen on the left side (the diagrams used to support the communication of the domain model are also indicated there). These artifacts, which serve as an input for the tool, were created and refined by the *business analyst team*. The generated artifacts are represented on the right side. These artifacts are the result of the *code generation* activity and are handed to the *agile development team*. The templates can be developed separately from the other parts of the tool.

The whole system is implemented with the help of approximately 5000 lines of XSLT templates and some hundred lines of shell scripts. The components of the tool must be as simple as possible and easy to learn, because the development process assumes short iterations (between two weeks and two months).

3.2.1 The Input

The domain model is represented in the form of Excel spreadsheets. These are used for communication between the customer and the *business analyst team*. Excel spreadsheets are a straight forward way to represent the domain model, because all stake-holders fill comfortable to work with them (according to our experience managers find editing spreadsheets more simpler than editing diagrams).

The Class/ER diagrams are only used to provide visualization. They augment the understanding of the big picture, but even customers trained to work with diagrams show resistance to contribute to them (this behavior may have at least two explanations: they do not feel comfortable with the diagramming tool or they do not fully understand the details). The initial format of the Excel spreadsheets is created during the *domain model extraction*; it can be refined during the *domain modeling*.

3.2.2 Input Aggregation

Input aggregation is the first processing phase of the the code generation tool. The Excel spreadsheets are converted to XML files, then are aggregated into a single XML file with the help of scripts. The two representations of the domain model are semantically equivalent. The conversion step also adds additional flexibility: the representation format of the Excel spreadsheet can be changed without affecting the other phases of the code generation tool. The part of the tool responsible for the input aggregation is fine-tuned during the *MDD environment development* activity.

3.2.3 Verification

Verification is the second processing phase. It consists two steps: an XML Schema validation and a constraint check:

- In the first step, the XML Schema validation, it is ensured that the XML representation of the domain model conforms to the XML Schema of its domain meta model. This serves as an extra safety check of the aggregated XML file.

- The second step, the constraint check, is the more important part of the verification. During this step several global constraints are checked (currently 24). The checks are implemented by some hundred lines of XSLT templates (currently 156). While is it possible to check the properties ensured during the first step by Excel macros, these global constraints can not, because this check needs information stored in separated Excel spreadsheets.

The required infrastructure for both steps can be quickly elaborated by the *MDD team* during the *MDD environment setup* and refined in the iterations during the *MDD environment development*. The code generation tool can be used by the *business analyst team* to verify the *domain model* by running only the first two processing phases. This usage is typical during the domain modeling.

3.2.4 Code Generation

The final processing phase is the code generation. The code generation tool and the templates are implemented in pure XSLT. The *code generator core* is a single XSLT file which is implemented in *rule-matching style* and calls the *templates* successively. The first template denormalizes the domain model before the templates are executed. The denormalization makes the templates simpler, because it eliminates a lot of the complicated referencing and repetitive computation of the values. The size of the XSLT templates varies between approximately 100 and 1500 lines. The bigger templates are already at the edge of maintainability.

The advantage of using XSLT to represent verification constraints, code generator core, and templates is that the developers can be quickly reallocated and they can also understand each others work with small extra effort. There were only two minor annoyances with the XSLT templates: the generation of ASPX and the formating of output.

- Microsoft's ASP.NET technology, called Active Server Pages, defines an extension for HTML to create web forms (ASPX). These files are neither proper XML nor HTML formats; thus the output mode for they have to be plain text, which implies that all special characters have to be escaped.

- The problem of output formating is more typical: to have the correct indentation and human readable formating, extra effort is needed. This problem is common with other template notations as well.

3.2.5 The Output

The result of the code generation is the source code of a web application. It is runnable with the necessary utility files, and can be tested immediately. The architecture of the generated application follows the current architecture guidelines [5, 12, 13, 33].

The *MDD team* has the task to keep the balance between the simplicity and size of the generated code. *Simplicity* means that the generated code must remain easy to understand and to extend. *Size* means that unnecessary duplication does not occur. Certain architectural guidelines imply a lot of generated code: stored procedures, data access objects, etc. On one hand, replications are not harmful if they are applied in a controlled fashion, because they can make the code faster, more readable, and easier to customize. On the other hand they make a certain type of changes much harder. Certainly, in a lot of cases it is a question of taste where and to which extend to apply replication.

The code generation approach helps also to eliminate certain types of errors, e.g. broken string references. The references are represented as strings that denote the stored procedures on the SQL-Server or information in the configuration files. The integrity of these string references can not be checked by the compilers.

4 The Practical Aspect

In this section we describe how the process and the technology explained in the previous sections have been applied in practice. We also describe minor details necessary to understand for carrying through a successful project. Both the process and the technology were successfully used in several projects. Here we describe that project in which they were applied for the first time. The project was carried through in the end of 2006 at the site of a customer who ships applications to its end customers.

4.1 Description of the Project

The aim of the project was to deliver an enterprise web application for a specific domain. The deliverable application had to build on top of Microsoft's ASP.NET 2.0 technology and an MS-SQL 2000 back-end. It also had to conform to architectural guide lines for the multi-layer enterprise ASP.NET applications at that time. Readable source code and complete developer documentation was also a must, because the end customer intended to extend and maintain the delivered application in house. The secondary goal was to introduce and evaluate the the process and technology concepts which we have developed. The evaluation was done according to the criteria explained in Section 1.2.

At the beginning of the described project a detailed preliminary database plan and a detailed software requirement specification were available. These described approximately 100 database tables and over 300 screen masks. In these early documents it was already easy to recognize repetitive functions and recurring simple Create, Read, Update, Delete (CRUD) dialogs. Both of the mentioned documents were accepted by the end customer, but they were too detailed and did not reflected its real expectation. Additionally, the end customer also had a detailed intranet application style guide. ASP.NET 1.1 software prototypes were available from our customer to demonstrate certain features of the software it planed to build. Also a technology demonstration prototype of the code generation tool was available which generated HTML files out of a small domain model which was similar to the target domain.

As a part of our work to develop the “Process and Technology Concepts” guide we reviewed the available literatures and the tools. Our customer already knew both the domain and the end customer well. This was an important factor, because, if one is not confident with the the domain and the requirements of the end customer, a longer initial phase is needed.

4.1.1 Initial Phase

The initial phase was started with the domain model extraction and took approximately one and a half months. In the first two weeks we fixed the initial version of the the domain model. After that, we started to create the domain meta-model and its XML Schema syntax. In parallel with that we started to set up the code generation tool. Our aim was that the initial version of this tool would include some simple templates which generate from the XML files an ASP.NET 2.0 application containing empty screens masks with navigation elements.

The first problem we faced was that the editing of the XML files was more challenging for the end customer than we thought. Initially, we planed to use XML files as a temporary solution, until we would build our own domain language and corresponding diagramming tool in a later iteration, but the time was not sufficient for this. Thus, we adapted Excel spreadsheets as a solution.

By the end of the initial phase we had the first version of the domain model represented in Excel spreadsheets, an informal description of the domain meta-model, a working version of the code generation tool and a part of the domain model as a domain test model. Our customer successfully used the first generated code to demonstrate to the end customer the navigation structure of the application.

4.1.2 First Iteration

The main goal of the first iteration was to generate in two weeks simple artifacts for all layers of the application. During that time also the initial architecture was fixed according to the architecture prototype developed by the agile development team. By the end of the iteration, the templates were capable to generate the following artifacts:

- the DDL scripts to create the database;
- simple stored procedures in T-SQL with SELECT statements;
- the methods in C# of the data access layer to call the stored procedures;
- the skeleton of the business objects in C#;
- and the ASPX forms to display data from the database.

The different generating phases in the prototype of the code generation tool had to be run manually. That issue was solved with the help of simple shell scripts.

At that point the internal team structure allocation stabilized. Our method completely fitted completely to the situation: the concept of business analyst team was not only a theoretical concept, but there was a real team who regularly met with the end customer. That team collected the requirements and communicated to the developers. The division of the development work between the agile development team and the MDD team was also proved efficient. Summarizing, we did not have any serious issue at the first iteration.

4.1.3 Second Iteration

This iteration was planed to take one month; its aim was to complete the functionality of the data access layer together with the necessary stored procedures and the business objects. The other enhancement was that a special domain test model was developed. This was necessary, because the original, which was a part of the real domain model, did not cover a lot of possible cases in the domain meta-model and was already too huge to get generated and compiled quickly.

4.1.4 Third Iteration

During this iteration, user interface features were implemented and enhanced. Extra functionality was also added to the data access layer according to the request of the agile development team. The iteration was completed in two weeks. Here we had the first serious integration problem: the iteration resulted in a lot of files that needed to be merged and fixed manually. This issue emerged from the fact that the teams did not take serious enough to keep track of the artifacts modified after the generation. This issue was solved by moving features to separate methods in C# partial classes, rather than code the features directly in the generated artifacts. The files in the version tracking system were also cleaned up.

4.1.5 Further Iterations

The fourth iteration was a one week long bug fix iteration. An error was found in the framework; this error was triggered by certain combination of the properties in the domain model. The error was not reproducible with the domain test model. [EXPLAIN] We closed this issue by developing a workaround and extending the integration test. The fifth iteration contained the last set of the features and the integration was gone smoothly.

4.1.6 Maintenance

During the maintenance most of the modification was carried out by the agile development team. If the domain was modified, new artifacts were generated. The integration of these new artifacts was carried through in a piece by piece manner: instead of collecting together the generated artifacts and the hand crafted artifacts and integrating them, the MDD team delivered just the requested new or modified files to the agile development team, which merged these artifacts manually. This ensured that the modification had as little effect on the other parts of the software as possible.

4.1.7 Metrics

In the final release the total amount of source code developed was approximately 300 thousand lines of code (KLOC). This contained 200 KLOC of generated code, which was produced from 900 lines of XML specification. [OTHER METRICS]

4.2 Best practices

In this section we describe the best practices were collected during our projects. Although they are simple, they can save a lot of unnecessary effort.

4.2.1 GUI Prototyping

During the initial setup of the code generation tool, it is an important question what kind of artifacts should be generated first. The generation of a rapid GUI prototype is a reasonable choice. This task is a realistic aim and during its implementation the team can face with the typical implementation challenges. It also results in valuable artifacts for the project stake-holders: a large scale GUI prototype is capable to demonstrate that the planed navigational or menu structure is not only effective with a small demonstration example.

4.2.2 Features Priority

[EXPLAIN]

4.2.3 Skip Integration

Integration is an important activity of the described MDD process. During this activity the generated artifacts and the hand crafted software are merged. Even if it proceeds smoothly, it takes time. If the changes do not affect the interfaces and the customer can be satisfied by demonstrating the improvement of the deliverable system separately, the integration can be safely skipped. If the interfaces are changed, then it is strongly recommended to do the integration. The more integration problems are postponed, the more problems have to be fixed later, which can be much more painful.

4.2.4 Minimize Manual Merge

By the integration, two different versions of the artifacts may emerge: one version of the artifact is generated during the previous integration and modified by the agile development team, the other version of the artifact is newly generated by the MDD team from the new templates and domain model. These versions have to be merged manually by human interaction. If too many artifacts need manual merge during the integration, this must be considered as a serious warning. The more manual effort the integration needs, the more inefficient the MDD approach is. If one does not start to solve this issue early, it can quickly become the bottleneck of the process. This issue can be resolved by the following two options:

- split either the generated artifacts and handcrafted software into separate files (e.g.: use partial classes in C#);
- or generate artifacts at the beginning and then, if the artifacts are customized, freeze the specification and those templates which can change the result of the generation.

It is important to document these decisions in the MDD vision document and evaluate which parts of the technological environment and the domain are more likely to change. This documentation also helps the business analysts to understand how much effort a different type of modification costs.

4.2.5 Variable Naming

The importance of selecting good variable names is well known by the software development community. There are a lot of guidance and coding standards to help developers in this task. While in the case of handcrafted software the variable names chosen by the developers, in the case of generated artifacts it is derived from the specification. Hence, business analysts have to be educated about the importance of name selection. The developers of the templates also have to pay extra attention to the variable names, even if it not affects their work directly. They should avoid adding long or senseless pre- or postfixes or loop counters to the identifiers.

4.2.6 Testing Issues

Testing is an important part of the current software quality assurance practice. In our process we prescribe several point where testing should be done. If the domain test model is carefully chosen, a high quality of the generated artifacts can be ensured by testing a relatively small amount of code. The quality of the domain test model can be expressed by the coverage of the templates and the coverage of the domain meta-model. This kind of test does not rule out the traditional integration, regression and stress tests.

5 Conclusion and Outlook

Model driven development and domain specific languages are active research areas both for the academic and the industrial communities. In this paper we described a lightweight MDD process and a supporting XML-based technology which were successfully applied in industrial projects. On one hand, we have provided a concise and abstract description of the core ideas which should serve as a referenceable source for the academic community. On the other hand, the process and the tool have been explained in simple and practical way, so they can be easily understand and applied by industrial practitioners. The process fits well into the existing methodologies and we provide a comprehensive description of the minor details which need to be understood for the real world projects. Our approach not only usable for small and mid size projects, but can be also used as preparatory step before the introduction of a heavyweight approach.

[OUTLOOK]

References

[1] AndroMDA 3.2. http://www.andromda.org.

[2] Borland Together 2007. http://www.borland.com /us/products/together/index.html.

[3] Code Generation Network. http://www.codegeneration.net/.

[4] Iron Speed Designer. http://www.ironspeed.com.

[5] *Enterprise Solution Patterns Using Microsoft .Net: Version 2.0 : Patterns & Practices*. Microsoft Press, 2003.

[6] *ISO/IEC 19757-3 First edition, Information technology - Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation - Schematron*. ISO, 2006-06-01.

[7] MOF 2.0 / XMI Mapping Specification, v2.1.1. http://www.omg.org /technology/documents/formal/xmi.htm, 2007-12-01.

[8] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing Applications Using Model-Driven Design Environments. *Computer*, 39(2):33–40, Feb. 2006.

[9] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[10] Cliff Binstock, Dave Peterson, Mitchell Smith, Mike Wooding, Chris Dix, and Chris Galtenberg. *The XML Schema Complete Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] Fatemeh Chitforoush, Maryam Yazdandoost, and Raman Ramsin. Methodology Support for the Model Driven Architecture. In *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC'07)*, pages 454–461, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Bill Evjen, Scott Hanselman, Devin Rader, Farhan Muhammad, and Srinivasa Sivakumar. *Professional ASP.NET 2.0 Special Edition (Wrox Professional Guides*. Wrox Press Ltd., Birmingham, UK, UK, 2006.

[13] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[14] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.

[15] Anastasius Gavras, Mariano Belaunde, Luís Ferreira Pires, and João Paulo A. Almeida. Towards an MDA-Based Development Methodology. In Flávio Oquendo, Brian Warboys, and Ronald Morrison, editors, *EWSA*, volume 3047 of *Lecture Notes in Computer Science*, pages 230–240. Springer, 2004.

[16] Marie-Pierre Gervais. Towards an MDA-Oriented Methodology. In *COMPSAC*, pages 265–270. IEEE Computer Society, 2002.

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

[18] Jack Herrington. *Code Generation in Action*. Manning Publications, 2003.

[19] Richard Hubert. *Convergent Architecture: Building Model Driven J2EE Systems with UML*. John Wiley & Sons, 2002.

[20] Wim Janssen, Alexandr Korlyukov, and Jan Van den Bussche. On the Tree-Transformation Power of XSLT. *Acta Inf.*, 43(6):371–393, 2006.

[21] G. Karsai. Why XML is Not Suitable for Semantic Translation. Research note, Nashville, TN, 2000.

[22] Michael Kay. *XSLT 2.0 Programmer's Reference, 3rd Edition*. Wrox Press Ltd., 2004.

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

[24] Alan S. Koch. *Agile Software Development: Evaluating The Methods For Your Organization*. Artech House Publishers, 2004.

[25] Per Kroll and Philippe Kruchten. *The Rational Unified Process Made Easy: a Practitioner's Guide to the RUP*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[26] Vinay Kulkarni and Sreedhar Reddy. Introducing MDA in a Large IT Consultancy Organization. In *APSEC*, pages 419–426. IEEE Computer Society, 2006.

[27] C. Larman and V.R. Basili. Iterative and Incremental Developments. A Brief History. *Computer*, 36(6):47–56, June 2003.

[28] Craig Larman. *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional, 2003.

[29] Jason Xabier Mansell, Aitor Bediaga, Régis Vogel, and Keith Mantell. A Process Framework for the Successful Adoption of Model Driven Development. In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 90–100. Springer, 2006.

[30] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley, 2002.

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

[32] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML Schema Languages Using Formal Language Theory. *ACM Trans. Inter. Tech.*, 5(4):660–704, 2005.

[33] Joachim Rossberg Rickard Redler. *Pro Scalable .NET 2.0 Application Designs*. Apress, 2005.

[34] S. Sarkar. Model Driven Programming Using XSLT: an Approach to Rapid Development of Domain-Specific Program Generators. *www.XML-JOURNAL.com*, pages 42–51, August 2002.

[35] B. Selic. The Pragmatics of Model-Driven Development. *Software, IEEE*, 20(5):19–25, Sept.-Oct. 2003.

[36] Bran Selic. Model-Driven Development: Its Essence and Opportunities. *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, pages 7 pp.–, 2006.

[37] Galen S. Swint, Calton Pu, Gueyoung Jung, Wenchang Yan, Younggyun Koh, Qinyi Wu, Charles Consel, Akhil Sahai, and Koichi Moriyama. Clearwater: Extensible, Flexible, Modular Code Generation. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 144–153, New York, NY, USA, 2005. ACM.

[38] Axel Uhl. Model-Driven Development in the Enterprise. *Software, IEEE*, 25(1):46–49, Jan.-Feb. 2008.

[39] Axel Uhl and Scott W. Ambler. Point/Counterpoint: Model Driven Architecture Is Ready for Prime Time / Agile Model Driven Development Is Good Enough. *IEEE Software*, 20(5):70–73, 2003.