# Abstract


We would like to understand the interests of our target audience. Please register at **www.softmetaware.com/whitepapers.html** to provide us with some information about yourself, and to obtain access to the full content of all SoftMetaWare white papers.

# Model-Driven Software Development

An emerging paradigm for

Industrialized Software Asset Development

Author: Jorn Bettin

Version 0.8

June 2004



SoftMetaWare

## Revision History

| Version | Author | Description |
|---|---|---|
| **0.1 - 0.6** | Jorn Bettin | Initial versions, January & February 2004 |
| **0.7** | Jorn Bettin | Update May 2004.<br><br>Included pointers to papers on *MDSD Teams* and *MDSD Activities* (process description).<br><br>Added diagram on MDSD adoption strategy in section *Economics of getting from X to MDSD*.<br><br>Added a glossary. |
| **0.8** | Jorn Bettin | Update June 2004.<br><br>Updated the *Techniques* section with further best practices.<br><br>Included pointers to the paper on *Complexity & Dependency Management*, which provides the link between MDSD and Design By Contract as well as the link between MDSD and the Open Source concept.<br><br>Included reference to role descriptions in *MDSD Teams* paper. |

# 1 Introduction

In the 21st century software is pervasive, the software industry has become one of the largest industries on the planet, and many of today's most successful companies are organizations built around the production of software and related services.

This article investigates the root causes of escalating software development costs, and presents an overview of an emerging paradigm for industrialized software development. Software is a critical part in the "engine room" of all technology-based and many service-based industries today. High software development costs have a huge economic impact, and poorly designed software that restrains user productivity possibly has an even larger impact. One result of these costs is significant pressure to shift problems to low-cost locations, usually referred to as off-shoring or outsourcing.

It is easy to overlook the fact that software development productivity varies by as much as an order of magnitude between organizations, and that off-shoring consequently is not always the best option for cost reduction in terms of achievable gains and risk exposure.

Many business software vendors have been side-tracked by keeping pace with the constantly changing set of implementation technologies. Neither off-shoring nor the next product release from tool vendors that provide infrastructure such as integrated development environments, middleware, databases, operating systems, etc. will solve productivity problems that are caused by crumbling architectural integrity of applications, poor dependency management within enterprise systems, and dysfunctional software development processes.

## 1.1 Yesterday's hype

The 90s were dominated by two major paradigms for software development: in the early 90s the concept of Computer Aided Software Engineering (CASE) and 4GLs made their appearance, and in the second half of the decade Object-Orientation made it into the mainstream. CASE methodologies and associated tools largely collapsed under the weight of hefty price tags and proprietary approaches that conflicted with the increasing demand for open standards. Often organizations were burnt by more than one vendor, and out the door went not only the tools, but also the concept of model-driven, generative approaches. Object orientation also did not live up to all expectations, but here the picture is somewhat different: it provided the foundation for component-based development, and object-oriented languages are there to stay, having successfully replaced most general-purpose procedural languages. With the fall of 4GLs and CASE, tool vendors focussed on object modeling tools, which led to the Unified Modeling Language (UML) notation and tools based on a "round-trip" philosophy, where users can switch seamlessly between a UML model and corresponding implementation source code. Superficially these tools impress by their ability to keep diagrams and source code in synch. A closer analysis however reveals that these tools don't in any way increase productivity, at best they provide a mechanism for producing nice looking documentation.

## 1.2 Today's hype

These days the boundaries between UML tools and IDEs are evaporating. Modern software development tools provide sophisticated wizards that assist users in applying design patterns, in building user interfaces, and in generating skeleton code for interaction with frameworks used in popular implementation technology stacks. Although this represents an improvement over earlier UML tools that were only capable of generating skeleton class structures, the approach smacks of the inflexibility of earlier tools. For example, if a design pattern needs to be changed, the current tools are not capable of automatically propagating the implications of the changed pattern through the code base.

Some traditional CASE vendors and several new players are exploiting the weaknesses of mainstream IDEs to offer Model-Driven Architecture (MDA) tools that allow users to specify precisely how high-level UML models should be mapped onto their specific implementation technology stack. MDA is a term coined and owned by the Object Management Group, a consortium that includes most mainstream vendors of software development tools. The MDA approach relies on UML and customizable code generation, and explicitly distinguishes between the concept of Platform Independent Model (PIM) and Platform Specific Model (PSM). In practice, commercial MDA tools are expensive, and rely on proprietary languages to specify the transformations between PIM and implementation code. This means that similar to CASE tools, there is an element of vendor dependence when going down the MDA track. An important aspect that is not addressed by MDA is the development of a rich domain layer that encapsulates core domain business logic; the UML notation is largely inadequate for the specification of domain business logic. Generative techniques as used in MDA are ideal for the generation of framework completion code, but they do not in any way eliminate the need for well-designed domain-specific frameworks. On the positive side, MDA is an approach that allows designers to raise the level of abstraction of specifications and to capture implementation technology knowledge in machine-readable format. Currently the OMG is working on a specification for a standardized language for model transformations. The limited degree of practical UML tool interoperability enabled through the OMG's XMI standard can be used as a reference point for realistic expectations for future MDA tool interoperability.

In parallel with advances in software development tools, there has been a major shift in software development methodologies. The emergence and rapid rise in popularity of agile methodologies is a good indication that methodologies perceived as being heavy are going out of fashion. If heavier methodologies were the flavor of the day in the late nineties, it does not mean that the majority of organizations actually ever fully embraced these methodologies wholeheartedly and followed them to the letter. On the contrary, it is much more an admission that high-ceremony approaches are only practical for the development of life-critical applications, where the associated costs of a heavy process can be justified and absorbed under the heading of quality assurance. For development of more mundane business application software, any process that involves a high degree of ceremony is incompatible with the push of the markets for lower software development costs. However, the focus of agile methodologies is largely limited to the management and process aspects of small to medium size software projects. An agile method such as Extreme Programming (XP) alone does not provide sufficient guidance for building high quality software, and it is no substitute for necessary analytical capabilities and software design skills within a team.

## 1.3 Model-Driven Software Development

The market for software development methodologies and tools is largely defined by the vast majority of software development projects, which is undertaken by teams of up to ten people, and typically delivers highly specific applications. This means that many software development methodologies - in particular of the agile variety, and many tools don't explicitly cater for software development in-the-large, i.e. distributed, multi-team software product development and projects involving 20+ people.

In the approach we call Model-Driven Software Development (MDSD) we combine aspects from popular mainstream approaches that can scale to large-scale industrialized software development with less well-known techniques that are needed to prevent architectural degradation in large systems, and with techniques to automate the repetitive aspects of software development.

MDSD can be defined as a multi-paradigm approach that embraces

- domain analysis,
- meta modeling,
- model driven generation,
- template languages,
- domain-driven framework design,
- the principles for agile software development,
- the development and use of Open Source infrastructure.

The remainder of this white paper explains how MDSD weaves together these ingredients into a consistent paradigm for software development. The foundation is provided by a small set of core values.

In the list of values below, the term "software factory" warrants an explanation. We use the term software factory to refer to domain-specific assets, in particular domain-specific frameworks, which are used as a platform to build a family or a series of applications using a highly automated production process.

1. We prefer to validate software-under-construction over validating software requirements
2. We work with domain-specific assets, which can be anything from models, components, frameworks, generators, to languages and techniques.
3. We strive to automate software construction from domain models; therefore we consciously distinguish between building software factories and building software applications
4. We support the emergence of supply chains for software development, which implies domain-specific specialization and enables mass customization

Considering the need to communicate the intention of these values to a wider audience beyond the software engineering community, we also use the term Industrialized Software Asset Development (ISAD) to refer to MDSD. This term accurately describes the state-of-the-practice (2004): domain-specific assets usually

need to be built, and only in some cases can they be bought off-the-shelf or are available as a public asset. To fully understand how this picture will evolve in the future requires a discussion of the economics of software development in a separate section in this paper.

# 2 Relating MDSD to Relevant Software Development Methods

## 2.1 MDSD and Software Product Line Engineering

Key features of MDSD come from the field of domain engineering, in particular the differentiation between building a product platform including relevant application development tools, and building software applications. In MDSD the product platform for a product family or product line is developed using domain-driven design principles, and the application engineering process is automated as far as possible using model-driven generative techniques.

One way of looking at MDSD is as a set of techniques that complements domain engineering (DE) methodologies such as FAST [WL 1999] or KobrA [ABKLMPWZ 2002], providing concrete guidance for:

- managing iterations,
- co-ordinating parallel domain engineering and application engineering,
- designing model-driven generators,
- and designing domain-specific application engineering processes

MDSD is intended to be compatible with DE methodologies such as FAST or KobrA, therefore the main focus of MDSD is on the description of techniques, and not on the specification of work products which can be adopted as required from proven domain engineering methodologies.

The concept of core "assets" from DE carries through into MDSD and is directly reflected in "Industrialized Software Asset Development" (ISAD), the subtitle of MDSD. The relationship between MDSD and DE can be compared to the relationship between Component Based Development and Object Technology: one is building on the other, and the terminology of MDSD can be seen as an extension of the terminology for DE. This means it makes little sense to draw up a point by point comparison between the two approaches.

Just as DE, MDSD leverages the concept of component specifications that are separate from component implementations. MDSD goes beyond the design by contract principles [Bettin 2004c], and mandates *Fully Externalized Interface Definitions* (see Techniques section in this document), which provide not only a very important tool for dependency management, but also provide an explanation for the importance of the Open Source concept for innovation in general, and for MDSD in particular.