UDC 004.413
Preliminary Communication Article

# Suitability of Modern Software Development Methodologies for Model Driven Development

**Ruben Picek** *ruben.picek@foi.hr*
*University of Zagreb*
*Faculty of Organization and Informatics Varaždin*

**Abstract**

As an answer to today's growing challenges in software industry, wide spectrum of new approaches of software development has occurred. One prominent direction is currently most promising software development paradigm called Model Driven Development (MDD). Despite a lot of skepticism and problems [8], MDD paradigm is being used and improved to accomplish many inherent potential benefits. In the methodological approach of software development it is necessary to use some kind of development process. Modern methodologies can be classified into two main categories: plan-driven/traditional or heavyweight and agile or lightweight. But when it is a question about MDD and development process for MDD, currently known methodologies are very poor or better said they don't have any explanation of MDD process[5], [7]. As the result of research, in this paper, author examines the possibilities of using existing modern software methodologies in context of MDD paradigm.
**Keywords:** Model Driven Development, Software Development, Modern Methodologies, Methodologies for Model Driven Development

## 1. Introduction

Detailed consideration of today's methodologies inevitably leads to the questions: How MDD paradigm fits with existing methodologies? Is it at all possible to use these methodologies for MDD or do we need to develop new methodology for it?

All the methodologies that are applied in today's, traditional development are based on the generic development phases: planning, analysis, design, coding, testing and delivery. In practice, during the development of a new system, most of the time and risks is spent on manually coding.

However, MDD paradigm changes the view of software development. It raises the level of abstraction, putting emphasis on the initial stages of development, especially during the analysis in which models has to be developed. Based on those models, programming code would be fully or partly generated using integrated development environment (IDE). Models which are created have to be accurate, consistent with sufficient level of details in order to achieve automatization using model transformations. MDD paradigm is still in development and some of the problems are not resolved: there are limited possibilities of development tools (problems with model transformation implementation), problems in defying standard modeling notation and languages for describing the model transformation.

Comparing these two "worlds" and taking into consideration their basic ideas, the differences are clear. Some of the classical phases are being automated. Because focus is shifted from the lower (implementation) to the upper (modeling) levels of abstraction, some activities lose their significance and new activities like creating transformation definition, implementing model transformations, modeling in some DSL language appears. Also, new roles in a team, which requires new forms of knowledge, are needed.

Do these differences exclude use of today's modern methodologies for MDD or they point the elements that these methodologies have to be improved and adjust? Author in article analyzes which of those two views are more suitable.

## 2. Modern Software Methodologies

In today's environment, in which software development is faced with many challenges because requirements of new and/or existing systems are growing, systems are complex and it is hard to build them on time and within budget limitations, awareness of the importance of using right methodology in software development is rising with each project. It is unthinkable to develop modern applications without proven methodology. In the last 30 years many different approaches for developing software were tried. If we want to classify today's methodologies one of the classification could be:

- plan-driven/traditional or heavyweight and
- agile or lightweight

While plan-driven/traditional methodologies emphasis detail planning, modeling and system documenting, agile methodologies emphasize that, due to today's environment in which software has to be created quickly and without redundant documentation, rapid developing and delivering a software will satisfy client requirements which changes frequently anyway.
Although there are no clearly defined borders between these two categories, plan-driven/traditional methodologies emphasize consistent commitment to the development process, while agile methodologies emphasize values and principles on which they are based.

In practice many organization use hybrid methodologies which are mix of above defined types and their own best practices.

### 2.1. Plan-driven/traditional methodologies

A main feature of methodologies in this category is good governance with system complexity - one of the two main challenges in software development. How they achieve this? The main characteristics of plan-driven/traditional methodologies are:

- Extensive planning.
- Large number of artifacts and formally described activities that are required to obey during the software development.
- Demand time, discipline and a large quantity of documentation that must track entire development cycle.

These methodologies are applied in the large, complex software systems development in which teams consisting of large number of people participate. The two most frequently used methodologies are: RUP (eng. Rational Unified Process) and MSF (eng. Microsoft Solution Framework).

### 2.2. Agile methodologies

Plan-driven/traditional methodologies dedicate considerable amount of time in defining *how* to develop software, and after that, focus is shift to programming and testing. On the other hand, in agile methodologies focus is on software, and they are trying to offer a way of developing SW with less extensive and not so detailed methodology, which brings quick and active processes. The idea is managing changes during the software development which is second of the two main challenges in software development.

With this idea, in the middle 1990's, developing less extensive software development methodologies, which typically contained only a few rules and activities that are light for tracking, began. Formally the term agile development was adopted after manifest "Agile Software Development Manifesto" was published in 2001 [4].

Of the 12 agile development principles published in the manifest, the 4 principles are basic [4]:

- Individuality and interaction are more important than processes and tools.
- Software that works is more important than comprehensive documentation.
- Cooperation with the client is more important than formal contract.
- Response to changes is more important than following formal plans.

As a result of agile approach following agile methodologies are developed [1], [17]:

- XP - Extreme Programming,
- Scrum,
- Crystal group of methodologies,
- Feature Driven Development,
- Dynamic System Development Method,
- Adaptive Software Development,
- Open Source Software Development,
- Agile Modeling
- Lean Software Development

## 3. Model Driven Development Paradigm

It can be said that, in last few years, software development evolve in significant manner. MDD represents a set of approaches, theories and methodological frameworks for industrialized software development, based on the systematic use of models as primary artifacts throughout the software development cycle [9].

### 3.1. Core Issues of Model Driven Development

The basic idea of this paradigm is to move the development efforts from programming to the higher level of abstraction, by using models as primary artifacts and by transforming models into source code or other artifacts. The ultimate objective is the automated development (fully or partly). Models are the key artifacts and the focus shifts from the programming to the modeling [18].

Traditionally, models are mostly used as sketches that informally convey some aspects of a system or they can be used as blueprints to describe a detailed design that is then manually implemented [20]. In MDD, models are used not just as sketches or blueprints, but as primary artifacts from which efficient implementations are generated, transforming models into programming code or other executable artifacts. According to Selic [15], the essence of model driven development is about two things. One is abstraction, in terms of how we think about the problem and then how we specify our solutions. Second thing that often gets forgotten is the introduction of more and more automation into the software development by using computer based tools and integrated environments.

The heart of MDD paradigm is: models, modeling and model transformation. In order to be suitable for the MDD, models must satisfy additional criteria – they must be machine readable. Machine-readability of models is a prerequisite for being able to generate artifacts. Automated model transformations are the key for realization of the MDD idea [3].

MDD paradigm addresses a core set of problems which are present in software development. Main identified problems are:

- *Overwhelming complexity:* MDD manages complexity by managing level of abstraction.
- *Not considering appropriate viewpoints:* MDD provides multiple views to address multiple concerns.
- *System does not meet functional, performance and other system concerns:* MDD integrates forms and functions.

- *Lack of scalability:* MDD consists of isomorphic composite recursive structures and method to address scalability.

Many industrial software practitioners express concern about the technical difficulties involved in translating models into code. From author's viewpoint this is the most important issue. Some of these important issues are discussed in section 3.3.

### 3.2. Benefits of Model Driven Development

According to [20], [19] MDD has the potential to greatly improve current practices in software development. This potential manifests in overcoming the current challenges – reducing the cost of development and increasing the consistency and quality of software.

Some of the more significant benefits include:

- *Reducing risk:* Many activities are strictly designed to reduce risk. Models increase understanding, reducing what is unknown, both technically and operationally, so that technical knowledge increases as iterations are completed. By increasing knowledge and reducing variance, MDD reduces risk.
- *Enhancing team and stakeholder communication:* Because words can be imprecise, teams use models to improve communication by making specific a particular aspect of a system. Models make system issues visible through the use of diagrams with which ambiguous is eliminated.
- *Explicit processes for reasoning about system issues and performing trade studies:* Many design decision are implicit –resulting from architect's experience. But the knowledge has to be explicate which indicate that the process also has to be explicate.
- *Early detection of errors:* Well designed process enables early error detection and resolution. The cost of errors rises significantly when is discovered in late phases of life cycle.
- *Traceability:* often is common requirement for the systems begin built. It is also needed to do effective fault or impact analysis to determine causes for faults and to determine which parts of the system will be affected by a requirements change.

Beside this advantages, authors in [20], [19] include following: increased developer productivity, maintainability, reuse of legacy, adaptability, consistency, repeatability, capture of domain knowledge, models as long-term assets and ability to delay technology decisions.

The potential benefits of using models are significantly greater in software than in other engineering disciplines because of the potential for a seamless link between models and the systems they represent. Unfortunately, models have rarely produced anticipated benefits. The key lies in resolving pragmatic issues related to the artifacts and culture of the previous generation of software technologies.

### 3.3. Review of Model Driven Development

This part of the article, provide a systematic look at MDD from the developers perspective and it is presented as a brief discussion of problems

The primary goal in MDD paradigm is to raise the level of abstraction at which developers operate. It should reduce both the amount of developer's efforts and the complexity of the software artifacts that the developers use [10], [12]. Of course, there is always a trade-off between simplification by raising the level of abstraction and oversimplification, where details for any useful transformation are missing.

As you can assume, problems are bound to model abstractions at different stages of the software life cycle. The open issue is how to transform a model at one level of abstraction, into a model or code at a lower level? In trying to answer this question, new ones arise. How to use models? Some developers use models only for sketching, others for blueprinting while MDD community presumed models as programming language.

Which notation and modelling language should be used in order to provide automation? The standardization of modelling notations is unquestionably an important step for achieving

MDD. Standardization provides developers with uniform modelling notations for a wide range of modelling activities. In SW industry today, the Unified Modelling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artefacts of software systems. The UML represents a collection of best engineering practices which have been proven in the modelling of large and complex systems. Although UML is widely recognized and used as modelling standard, it provoked a lot of criticism.

Is UML suitable as model programming language? The notion of UML 2.0 as a model programming language is predicated on the belief that the use of higher levels of abstraction will make developers more productive than current programming languages. Is this belief true? Furthermore Greenfield et al. [9] argue that although UML 2.0 is a useful modelling language, it is not an appropriate language for MDD, because UML is designed for documenting and not for programming. They promote use of special-purpose, domain-specific languages (DSL's). According to [12], MDD creates other problems, like: redundancy, rampant round-trip problems, moving complexity rather than reducing it and more expertise that is required.

Selic in [16] point out that having right answers on questions like *Will the code be fast and compact enough? Will it be a correct rendering of design intent?* is one of the key elements of MDD success. The same thoughts were on minds when compilers were introduced. Like all compilers, automatic code generators are idiosyncratic and often generate program code that, as a result of various internal optimizations, is not easily traceable to the original model. Thus, if an error is detected in the generated program, finding the place in the model that must be fixed either at compile time or runtime might be difficult. In traditional programming languages, we expect compilers to report errors in terms of the original source code and, for runtime errors, we now expect a similar capability from our debuggers. The need for such facilities for models is even greater because the semantic gap between the modeling language's high-level abstractions and the implementation code is wider. This means that model-level error reporting and debugging facilities (in essence, "decompilers") must accompany practical automatic code generators. Otherwise, the practical difficulties encountered in diagnosing problems could be significant enough to nullify much of MDD's advantage. Programmers faced with fixing code that they don't understand will easily break it and will likely be discouraged from relying on models in the future. This is a particularly important factor to consider for model-driven development that is based on the notion of customizable transformation "templates". Other important questions are: *how the generated code is equivalent to hand-written code? How to merge two or more possible overlapping models drawn in different IDE versions into one and generate code?* and finally, *how to integrate this systems with existing legacy systems?* We must wait to find right answers to these questions. Currently we must argue that full realizations of the MDE vision may not be possible in the near to medium-term primarily because of the wicked problems involved. This discussion can be concluded with fact that MDD's success is not predicted only on resolving obvious technical issues like defining suitable modeling language and automatic code generation.

This is the state of the art. The MDD paradigm brings a lot of open issues on ice and solutions are being searched in two directions: methodology and technology. The Object Managements Group (OMG) proposed the approach called Model Driven Architecture (MDA). Industrial leaders are also developing their own solutions, such as the Microsoft's Software Factories (MSF) [10].

## 4. Analysis – How Today's Methodologies Fits for Model Driven Development

With emergence of new types of software and development paradigms, need for discovering suitable ways (new or improving existing) of methodological development, is growing. In section 2, today's methodologies were classified into 2 groups, this analysis analyzed concepts that are characterized in theory for each category and then compared with MDD concepts.

### 4.1. Model Driven Development and Plan-Driven/Traditional Methodologies

MDD paradigm can be observed as a next evolutionary step in the software development, which is mostly based on plan-driven/traditional methodologies (agile occurred in middle 90's). Analysis, in which MDD paradigm range currently meets main features of plan-driven/traditional methodologies, can be seen in Table 1.

Table 1 presents comparison between characteristic of plan-driven/traditional methodologies and MDD concepts.

| Characteristics of plan-driven/traditional methodologies | MDD paradigm concepts |
|---|---|
| Based on extensive planning and detailed descriptions of problem domain. | In planning, emphasis is put on achieve problem domain understanding in order to define usable models. |
| A large number of artifacts and strictly formally described activities that are required to abide during the software development. | The main types of artifacts are models and model transformations. Ordering enforcement activities, it is necessary to respect to successfully generate programming code. |
| Demand time, discipline and a large quantity of documentation that must track entire development cycle. | Time of development and documentation writing is reduced by applying generators.<br>During defining models, mode to model (M2M) and mode to code (M2C) transformation, big discipline is required.<br>Writing documentation is not a follow-up activity, but part of the model specification process, from which documentation can be generated any time. |

Table 1. Suitability of MDD paradigm concepts with main characteristics of plan-driven/traditional methodologies

From author opinion, segments that need to be adapted, changed or expanded to those plan-driven/traditional methodologies would be minimally appropriate for MDD development relates to [14]:

- *Team:* because of MDD characteristics, new roles and additional knowledge is required, so that elements have to be formally added in methodologies [11].
- These new and important roles can be divided into two groups: domain and model transformation and application design. In the first group, we can distinguish the following roles: *domain expert* (involved in defining a DSL), *language engineer* (uses a meta language to specify the concrete syntax and abstract syntax of a DSL), *transformation specialist* (defines how models defined in DSL's are executed or transformed into an executable model using model transformation language) and *implementation/platform expert* (has expertise knowing everything about executing / interpreting a model.
- In the second group, we can distinguish the following roles in which scope of activities are changed (name is the same): *business engineer* (translates a business problem into a formal application model specified in a DSL. This role needs both an understanding of the problem domain and skills to express that understanding in a formal model.), *application/solution architect* (decides on the application architecture – platform implementation), *test engineer* (testing is performed at the Meta level).
- *Development process:* changing the importance and scope of activity in the some development phases and introducing new activities (e.g. transformation definition,

implementing this transformation on models and modelling in some DSL language), which should support automation is necessary to transform some aspects of development phases. Model analysis and development with appropriate transformations have greater importance than coding, so in that part especially has to transform some steps and add those activities (e.g. defining model transformations, implementing transformations and so on) which are typical for defining model transformations. It is important to stress that transformations have to merge classical phases: analysis and coding into one which would consist of defining model transformations and their execution in IDE and manually coding (modelling + transforming + coding). This new activities, for each role, have to be strictly defined. Having standard modelling language for defining model transformation is also important. Some guidance in that segment is expected to.

- *Technology:* development environment and its capabilities is a critical component of success because they brings the opportunity for fast reaction to changes in user requirements, reduces the duration of iteration, gets faster feedback from users, saves time in activities that are repeated and error prone and reduces risk. So, development environment and its capabilities play a key role in MDD and act as a critical component of success. Some guidelines have to be written in this segment.
- *Modeling:* the importance of models is recognized if someone wants to use them as building blocks, so it is necessary to define when which elements has to be added in models (marking models). The main types of artefacts are *models* and *model transformations* (model to model -M2M and model to code - M2C). All details in defining these artefacts have to be strictly formally respected (e.g. updating) to successfully generate programming code.
- *Problem domain:* All domains aren't suitable for MDD development. The solution is in developing different domain specific modeling languages which bring as to new question: Is it necessary to have one methodology for each domain with modeling language which is proven as best practice?

### 4.2. Model Driven Development and Agile Methodologies

Can MDD paradigm be seen in the context of agile methodologies, when at first glance most of them (XP, Feature Driven Development, Dynamic System Development Method, Adaptive Software Development, Open Source Software Development) emphasize different aspects of development (programming vs. modeling)? Author in papers [2], [5] find intentions that MDD paradigm can be compatible with the principles of agile development. With this idea, group of researchers decided to start the approach called agile MDD paradigm [2]. In further analysis focus will be directed on displaying differences and similarities of these two concepts. Differences are based on the fact that agile methodologies emphasize people, while the MDD relies on advanced technology that define the technology independent models and generate code. Another fundamental difference is visible in the fact that agile methodologies emphasize software development with using programming as a basic technique while MDD is based on modeling.

Through Table 2 some aspects of those differences are clearly observable.

| Aspect | Agile methodologies | MDD paradigm |
|---|---|---|
| People | Represent the most significant factor with the highest priority during the software development. | People are seen from a technological perspective, through roles in the processes. In paradigm new roles appears and social aspect is ignored. |
| Development process | All activities during the process are not defined in detail. The emphasis is on the testing and coding activities. Applies iterative and incremental process. | In comparison with the plan-driven/traditional process of developing some of the phases (design, coding, testing) are being automated, and some activities are added in order to achieve code generation. Analysis with modeling is a key phase. Defining and implementing the transformation are critical parts of the process. |
| Technology and IDE | Has the lowest priority. Moreover, it is important that the tools are easy to leave the impression that person has complete control over the development. | Greatest importance is given to technology. Depending on the tool assesses the performance of enforcement paradigm. |
| Modeling | It has a marginal importance. Understanding the system is achieved through developing a prototype. | The central activity. Development depends on its quality (success or fail). |
| Coding | Manually | Seeks to implement as much generation (for now partial) of programming code as possible. Striving for the entire generating code from the well-defined models. Re-generation because of changes in requirements is not a problem if the changes are quality implemented over the model. |
| Problem domain | Environment with dynamic change requirements. | Environment with stable requirements. |

Table 2. Differences between agile methodologies and MDD paradigm

Although, at first glance, it may be concluded that the MDD paradigm better fits with the plan-driven/traditional methodologies where some phases seek to be automated, it is possible to find the common points with agile approach. That can be seen through the principles which emphasize agile development.

Table 3 shows the most significant similarities [5].

| Aspect | Agile methodologies | MDD paradigm |
|---|---|---|
| Individuality and interaction are more important than processes and tools. | Emphasize the link and fellowship programmers and human roles versus the institutional processes and development tools. | The structure and activities of the team are not strictly defined. Although generators applications play a key role, the tools are based on the specific domains. |
| Software that works is more important than comprehensive documentation. | Updating key artifacts at all levels of abstraction trying to avoid inconsistencies. | Models at the highest level of abstraction have to be always updated. Consistency is achieved by generating from those models. |
| Cooperation with the client is more important than formal contract. | Client is permanently engagement to project. | There is no special attention, but it opens the possibility of including client intensively for testing applications. How MDD enables quickly model transformation, development does not lose time if the customer is not satisfied with the realization of a request. But, for now, this is grounded in the theoretical level of MDD paradigm. |
| Response to changes is more important than following formal plans. | Change managing is more important than following plan that does not match the new (changed) requirements. | The generation of code, allows quick reaction to the new request or change in the already existing one. It is enough to change the model and restart the code generation, what is considerably simpler than to modify code manually. |

Table 3. Similarities between agile methodologies and MDD paradigm

Based on these agile methodologies principles, it looks that, for the MDD development, connection points can be found.

IDE (MDD generators) naturally leads to the realization of some agile practices: brings the opportunity for fast reaction to changes in user requirements, reduces the duration of iteration, gets faster feedback from users, saves time in activities that are repeated and error prone and reduces risk. Despite the differences that exist, the MDD paradigm can be seen in the context of agile methodologies; they do not exclude each other. But, how agility emphasizes general values and principles, and do not determine formal steps that needs to be applied, remains an open question of suitability and compatibility degree of using agile methodologies in the context of MDD development.

## 5. Conclusion

This paper presents a short review of current state in modern software methodologies and model driven development. Article presents a core issues, problems, benefits and discussion about MDD paradigm. Central activity in this article was analysis in which it is explained how today's plan-driven/traditional and agile methodologies fits into the context of MDD.

From this analysis a few things can be emphasizes:

- Although the MDD paradigm is investing considerable effort in order to solve problems that encountered, methodology support has been largely overlooked. It remains insufficient because it does not provide a concrete and comprehensive process for governing software development activities. There are very few MDD based software development methodologies available, and those with precise processes are even fewer.
- Today's methodologies with an appropriate adjustments and changes can be used for MDD projects. But, when you use it you are condemned to combine the parts you think that are suitable. Using try and error method in developing hybrid MDD methodology isn't adequate approach of developing SW.
- Currently examples of using MDD within today's methodologies in the literature can't be found. It is partial because the MDD paradigm is still developing, and that there are many problems beyond which skepticism and modesty exists in using MDD paradigm.
- MDD paradigm has more sense in the context of traditional methodologies, and that in the context of agile methodologies her contribution will be modest.

All of this leads to the conclusion that it is necessary for the MDD paradigm to ensure and define its own process with phases, activities and roles with detail description.

Attempting to realize the MDD vision it is necessary to ensure and define its own process with phases, activities and roles with detail description [8]. This will provide insights that can be used to significantly reduce the gap between evolving software complexity and the technologies used to manage complexity.

## References

[1] Abrahamsson, P. et al.: Agile Software Development Methods - Review and Analysis, *VTT Publications*, 2002

[2] Ambler, W. S: Agile Model Driven Development (AMDD): The Key to Scaling Agile Software Development, 2007, http://www.agilemodeling.com/essays/amdd.htm, downloaded: December 14th 2008

[3] Balmelli, L., Brown, D., Cantor, M., Mott, M.: Model - Driven Systems Development*, IBM Systems Journal*, Vol 45, No 3, 2006., p. 569-585.

[4] Beck, K., et al.: Agile Software Development Manifesto, 2001, http://agilemanifesto.org/, downloaded: September 12th 2008

[5] Bettin, J: Model-Driven Software Development, *SoftMetaWare*, 2004 http://www.softmetaware.com/whitepapers.html, downloaded: January 22nd 2009

[6] Chitforoush, F., et al.: Methodology Support for the Model Driven Architecture, Software Engineering Conference, *APSEC 2007*. 14th Asia-Pacific, Volume , Issue , pages:454 – 461, 2007

[7] Chitforoush, F., Yazdandoost, M., Ramsin,R.: Methodologiy Support for the Model Driven Architecture, 14 Asia-Pacific Software Engineering Conference, 2007., IEEE DOI 10.1109/ASPEC.2007.58

[8] Franc, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap**,** I*nternational Conference on Software Engineering*, Pages 37-54, 2007, ISBN:0-7695-2829-5, IEEE Computer Society Washington, DC, USA

[9] Greenfield, J., Short, K., Cook, S. and Kent, S.: *Software Factories - Assembling Application with Patterns, Models, Frameworks and Tools*, Wiley Publishing, Indianapolis., 2006

[10] Greenfield, J., Short, K.: Moving to Software Factories, 2004, http://blogs.msdn.com/askburton/archive/2004/09/20/232065.aspx, downloaded: December 13th 2008

[11] Haan, J.: Roles in Model Driven Engineering, *The Enterprise Architect*, February 2009., http://www.theenterprisearchitect.eu/archive/2009/02/04/roles-in-model-driven-engineering, downloaded: April 01th 2009

[12] Hailpern, B., Tarr, P.: Model-Driven Development: The Good, the Bad, and the Ugly, *IBM System Yournal*, Vol 45, No 3. 2006, str. 451-461., http://www.research.ibm.com/journal/sj/453/hailpern.html, downloaded: September 02th 2008

[13] Mellor, S., Clark, A., Futagami, T.: Model Driven Development, *IEEE Software*p. 14-18., 2003

[14] Picek, R., Stubljar, S: (2009) Methodological Aspects of the Model Driven Development, IADIS Multi Conference on Computer Science and Information Systems, *Proceedings of INFORMATICS 2009*, ISBN 978-972-8924-86-7, p. 155-159., Algarve, Portugal 17-19.06.2009.

[15] Pierson, H.: ARCast #5, http://channel9.msdn.com/ Showpost.aspx?postid=132943, downloaded: August 02th 2008

[16] Selic, B.: The Pragmatics of Model-Driven Development, *IEEE Software*, p. 19-25., 2003

[17] Sommerwille, I.: *Software Engineering 8*, Addison-Wesley, 2007

[18] Stein,D., Hanenberg, S.: Why Aspect-Oriented Software Development And Model Driven Development Are Not The Same, Electronic Notes in Theoretical Computer Science 163, p 71-82, 2006

[19] Swithinbank, P., Chessell, M., Gardner, T., Griffin, C., Man, J., Wylie, H., Yusuf, L.: Patterns: *Model-Driven Development Using IBM Rational Software Architect*, IBM Redbooks, 2005

[20] Yusuf, L., Chessel, M. and Gardner, T.: Implement Model-Driven Development to Increase the Business Value of Your IT System, 2006, http://www-128.ibm.com/developerworks/library/ar-mdd1/, downloaded: December 13th 2008

[21] Yusuf, L., Gardner,T.: Explore model-driven development (MDD) and related approaches: A closer look at model-driven development and other industry initiatives, 2006, http://www-128.ibm.com/developerworks/library/ar-mdd3/, downloaded: December 13th 2008