The Impact of Model Driven Development on the Software Architecture Process

Conference Paper *in* Conference Proceedings of the EUROMICRO · September 2010

DOI: 10.1109/SEAA.2010.63

CITATIONS
15

READS
474

2 authors, including:

Michel R. V. Chaudron
Eindhoven University of Technology
257 PUBLICATIONS **3,992** CITATIONS

SEE PROFILE

The Impact of Model Driven Development on the Software Architecture Process

Werner Heijstek Michel R. V. Chaudron

Leiden Institute of Advanced Computer Science
Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
http://www.liacs.nl/~{heijstek,chaudron}
{heijstek,chaudron}@liacs.nl

Abstract

While Model-Driven Development (MDD) is an increasingly popular software development approach, its impact on the development process in large-scale, industrial practice is not yet clear. For this study the application of MDD in a large-scale industrial software development project is analyzed over a period of two years. Applying a grounded theory approach we identified 14 factors which impact the architectural process. We found that scope creep is more likely to occur, late changes can imply more extensive rework and that business engineers need to be more aware of the technical impact of their decisions. In addition, the introduced Domain-Specific Language (DSL) provides a new common idiom that can be used by more team members and will ease communication among team members and with clients. Also, modelers need to be much more explicit and complete in their descriptions. Parallel development of a code generator and defining a proper meta-model require additional time investments. Lastly, the more central role of software architecture design documentation requires more structured, detailed and complete architectural information and consequently, more frequent reviews.

1 Introduction

In the past decade, there has been increasing interest in Model Driven Development (MDD, [10]) in industry and academia. MDD is not a new idea. Model-centric development and code generation from higher abstraction languages have been discussed, studied and applied for decades. Claimed productivity benefits generated many studies to advance MDD practices. Nonetheless, few empirical studies are available that study the impact of adopting MDD on industrial software development processes. As a result, the general impact of adopting MDD tools and techniques on software development processes is unclear. Adoption of MDD tools and techniques demands a shift in boundaries between traditional roles in the software development process as the model replaces the code as the central artifact. This has a profound impact on the architectural process and the work of the software architect who has to work with a different set of tools, a different vocabulary, a different type of development team and who has an increased responsibility to maintain consistency throughout the development process. In this study, we aim to understand how the adoption of MDD tools and techniques affects the architectural process and the role and responsibilities of the software architect.

2 Related Work

One of the suggested benefits of Model Driven Architecture (MDA, [4, 8]) according to a study by the Middleware Company [13] mentions 'architectural advantages'. The study explains that by adoption of MDA, an architect is forced to spend more time designing an architecture due to the necessity of modeling high-level domain entities. The Middleware Company argues that increased upfront design effort reduces 'the possibility of introducing architectural flaws into [your] system later in the development life cycle'. The study further reports on an experiment in which the same application is developed by two teams of developers. One team applies MDA and one team does not. The MDA team finished their development ahead of schedule and significantly faster.

A case study by MacDonald et al. [7] of modification of a legacy system using MDD found that development lead time increased due to 'workarounds required to integrate with legacy systems'. This directly impacts the work of a software architect. Furthermore, they found as many defects as the authors would have expected with 'traditional development'. Moreover, these defects were more difficult to find and repair in the models due to difficulties in tracing

errors from the compiler directly back to the model without using the generated code as a reference. Suffice to say that technical support of this type of development process is demanding. The study did not use a fully functional executable model. Also, it was hard to maintain platform independence due to work methods and a lack of generic libraries.

Advantages of application of MDD reported, include increased ease of communication of the design (including to the client) and consistency between design and code. Both are closely related to the core activities of a software architect.

According to a survey by Staron [11], among the main aims for adopters of MDD were improving quality by increasing understanding, improving communication within development team and traceability throughout software development artifacts (models). These three expected benefits are directly related to the responsibilities of a software architect. Adoption of MDD is therefore expected to alter the role of the software architect.

Farenhorst et al [1] list five categories of architecting activities. At least three categories are expected to be impacted by adopting MDD. First, communication of architecture design is expected to be easier because models are the dominant artifact throughout the project. Second, quality assessment is likely to be more important because of the more formal nature of MDD. Lastly, a stronger focus is expected on documentation due to the use of a Domain-Specific Language (DSL).

Hailpern and Tarr [2] assert that 'MDD has a chance to succeed in the realm of large, distributed, industrial software development'. This study gathers evidence to validate this assertion.

3 Case Study Design

In this section we describe the case study design. For our case study design, we used the guidelines by Runeson and Höst [9].

3.1 Context and Subjects

We examine a project in which a system was defined, designed and built for supporting the mid-office processes of the mortgage business. The client was a large financial institution that operates globally and the contractor is the Dutch subsidiary of an international IT service provider. The responsible department for development of the system has extensive, multi-year experience with building tens of software systems for the financial sector. Important project characteristics include:

- A function point analysis that was based on the requirements was executed in the early stages of the project reported a total of 1,973 function points. During the execution of the project, various change requests have been made.
- A total of 32 team members worked on the project of which only a few did not work on this particular project full-time. This corresponds to 28 full time equivalents (FTE).
- Total project duration was 24 months.
- Only four team members had experience with a previous project in which MDD techniques were applied.
- The Rational Unified Process (RUP) was used for process management. The RUP is an adaptable process framework that is architecture-centric and risk-driven and can be used for iterative software development [5].
- Compared to similar projects in the company, this project had relatively long inception and elaboration phases (one year combined).

The project was carried out distributedly. A team of six developers and six testers worked in India. Modeling was done in the Netherlands by a team of four modelers, development was done at both locations and testing is done in India. The Dutch project leader was also the main communication line to the customer in the Netherlands. The contractor has successfully offshored several large-scale projects.

3.2 Data collection and Analysis Methods

We used a grounded theory approach. Grounded theory [12] is rooted in the data that has been collected in observation of a studied phenomenon. As opposed to forming theories top-down, based on a priori assumptions, theory is formed bottom-up from collected data, in a systematic manner. Our method of data collection and analysis is schematically outlined in Fig. 1.

We used semi-structured interviews to survey a subset of the project team members. We interviewed both the project manager and lead architect before, during and after the project. The lead architect was interviewed extensively six times over the course of two years. In the first interviews, the structure of the project and the approach of the application of MDD techniques were discussed. The high amount of interviews was needed because the contracting organization was not used to manage MDD projects and therefore had limited insight in the approach and progress of the project. We extensively studied the use of models in this case and reported on this in earlier work [3]. We also extensively interviewed other team members including modelers, developers, lead developers, project leaders, a test manager, a system analyst, the architect and an estimation & measurement officer involved in sizing and tracking the project.

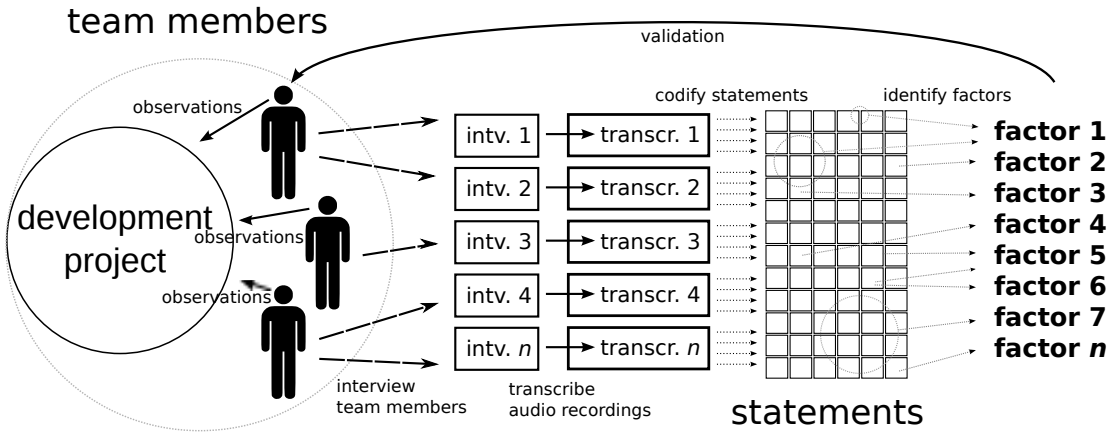

Figure 1. Data collection and analysis method

During the interviews we asked questions regarding the impact of the application of MDD on the activities of the participant and on the process of software development in general. All questions were directed at (1) identifying every possible architectural process-related differences with a non-MDD project and (2) finding all possible confounding factors. An audio-recording was made of all interviews. The next step involved transcribing and coding the audio recordings. All separate statements made by the subject were collected in a list. We then marked each statement that related to MDD. After this initial coding process, we grouped the statements and identified (formulated) a common impact factor that best described all statements in one group. We then removed and merged duplicate and overlapping impact factors and we established whether the impact was either (1) caused by the application of MDD, (2) the cause of MDD and other, non-MDD, factors or (3) most likely not the cause of MDD. We then confronted the interview participants with this distilled list to validate our interpretations. We repeated the coding process and updated factors in the same way.

4 Results

Through the iterative data collection process we identified 14 impacts of application of MDD on the architectural process. The impacts are listed in Tab. 1 and discussed in the following paragraphs.

4.1 Increased likelihood of scope creep due to ease of change

Scope creep occurs when system functionality expands beyond the initial project objectives. Any software development project will meet changing requirements. Generally, project management evaluates whether a change is in scope before accepting it as part of the original system or whether it should be treated as an additional functionality. The adoption of MDD can make it more difficult to guard against scope creep for two reasons. First, extending functionality can be easier than in non-MDD development. Second, an existing meta-model may contain more functionality than specified in the requirements, making it even easier to generate new functionality. This impacts the discussion between software supplier and client whether added or changed functionality is part of the original system requirements or if it should be treated as a change request. It might be easy from a technical perspective to generate functionality that is beyond project scope. However, the impact of added functionality extends beyond the technical implementation. Extra functionality requires increased test and documentation effort. Working beyond project scope might require additional iterations of the analysis of the business modeling because added functionality might impact existing business processes of systems in the environment which might imply the inclusion of additional interfaces. In addition, not all code in an MDD project is generated and a part of the newly generated code might need to be amended by hand. This is costly, time consuming and it might well add to the complexity of the system.

The organizational impact of features beyond initial project scope could also include additional training of future users or an extension of the pool of future users which in turn might impact other requirements. In addition, income from change requests might be an important part of the business model of a software development organization.

Because MDD still has to prove its benefits in large-scale industrial development, in some organizations its methods and tools are applied in a test-case or project. The (technical) team leader in such projects is generally very enthusiastic about the possibilities offered by MDD. Project management regularly budgeted client-requests for specific meta-

model functionality at zero hours of person effort. Management reasoned that adding functionality already available in the meta model did not cost any effort. In this case, severe project time and budget overruns could in great part be ascribed to this practice.

4.2 Late changes beyond project scope have a more fundamental impact

In MDD projects, fluctuation of requirements has an additional penalty because it impacts the meta-model which is used as the basis for code-generation. Late changes to the meta-model may require a disproportional amount of effort in modifying existing models or generated code. To prevent major rework, any requirement that impacts the meta-model must be clear upfront. The architect should make sure that before commencing modeling, the meta-model is as mature as possible. In this case, at a late stage in the project, a specific requirement regarding navigation through the graphical user interface was discovered. In an earlier version of the architecture it was prescribed that to navigate from one screen to another, that those two screens had to be explicitly connected to each other in the model. Marking every navigation step with an arrow implies that an increase in the amount of screens that can reach each other exponentially increases the amount of arrows that needed to be drawn in the models. Since it was not made clear that most of the screens would require the possibility to navigate to one another, that particular aspect of the architecture would probably have been redesigned at an early stage in the project. In a traditional project, this problem would probably be solved at the code level. In the case of this project, the screen navigation had to be altered at the model level and even required changes to the architecture, reference model and generator. The considerable amount of effort that had to be spent to rework all models after the late revision of fundamental aspects of the meta-model, indicates that it is imperative to find all requirements that significantly impact the meta-model *before* taking up modeling.

4.3 Models facilitate easier communication for a broader group of disciplines

An important task of a software architect is communication of design decisions with a broad group of team members. A substantial amount of the architectural process is spend on communication of design and architectural decisions. By introducing models as a common language that is used throughout the development process, this time consuming undertaking is less laborious. Depending on the composition of the applied DSL or the complexity of any existing or envisioned meta-model, diagrams are more appropriate means for communication than the source code of even a third generation programming language. The DSL offers a common language for team members across disciplines. The project team members that were interviewed acknowledge that technical discussions related to aspects of the system were easier to conduct than they were used to in non-MDD projects. Reasons consistently mentioned for the discussion benefits were that the models could be used as a basis for discussion and because more different team members of various disciplines were familiar with the models. However, the transition to models as a common language might not be equally undemanding for team members of all disciplines.

4.4 Models become a language for business engineers

The technical possibilities and potential advantages in terms of synergy offered by consistent use of diagrams from early requirement workshops to the generation of a working software system are evident. Much is lost and misunderstood in translation from requirements to architecture to design to source code. And while MDD does not completely remove the need for translation, it at least limits it. In practice, this implies that more team members need to be able to communicate in a common language. Nevertheless, a clear distinction between business related activities and IT-related activities is still often made in software engineering practice. Requirements engineers do not write source code, developers do not bother with domain models and a project manager might not be up to speed with specific testing techniques employed. The extent to which this division into 'technical' and 'non-technical' lies at the heart of some of the problems which seem inherent to the complex process dynamics of software engineering in general, is not a topic of debate in this study. Suffice to say that software engineering processes could benefit from increased convergence in the knowledge and responsibilities that the team members in the various types of available roles have, within a project. This is even more the case in MDD processes where the business domain is more closely related to the implementation of a solution. In the project under study, a group of requirement engineers were only willing to participate in modeling their use-cases more formally on the condition that they would not be concerned with 'programming'. However, the models they made were directly generated to code, code that would be directly used in the system and which would become the vast majority (over 87%) of the final code of which the system would be comprised. Another example are a group of business analysts who refused to work with a UML case tool as they regarded it 'technical work'. They eventually left the project.

Table 1. Impact of MDD adoption on the Architectural Software Process

I.	**Changes**	
	A.	Increased likelihood of scope creep due to ease of change
	B.	Late changes beyond project scope have a more fundamental impact
II.	**DSL as a Common Language**	
	C.	Models become a language for business engineers
	D.	Models facilitate easier communication for a broader group of disciplines
	E.	Maintainers need to be educated during development to be able to understand the modeling and generation processes
III.	**Modelers are more central**	
	F.	Modelers need to truly understand the DSL
	G.	Modelers need to build more unequivocal models
IV.	**Programmers are less central**	
	H.	Less skill is required of most developers, more skill is required of some
	I.	Collective code ownership is more important
V.	**Added Effort**	
	J.	Code generator is an additional application that needs to be developed and maintained in parallel
	R.	Mismatches between meta-model domain reality and client reality need to be acknowledged
VI.	**More Formal Development**	
	L.	More tooling is needed to support the MDD process
	M.	Adherence to modeling guidelines by modelers can be enforced
	N.	Architectural descriptions need to be more extensive, formal and structured

4.5 Maintainers need to be educated during development to be able to understand the modeling and generation processes

To ensure that models and system stay in-sync, it is imperative that maintainers are trained to understand the models and the generation process. It is essential that post-release changes are applied consistent with the MDD process used during development. Therefore, intimate knowledge of the meta-model buildup and the code generator as well as the frameworks involved is required. This knowledge is best obtained by close involvement of the development process. By waiting until system deployment for handing over the development process to a maintenance team, it becomes challenging to train maintainers to a level where they would be capable to repeat the applied MDD process when applying changes. For smaller changes it is especially tempting to apply manual changes in the generated code. This would dismiss the principle prescribes that generated source code should be treated as a build-time artifact, creating an inconsistency with the source models and instantly removing all generative possibilities. Maintainers therefore need to be trained up to a level that they are knowledgeable and comfortable enough with the specific MDD approach to apply it even for the smallest of changes. This implies that also more capable maintainers in general are needed. An additional disadvantage for a client is that although fewer people are needed to maintain the application, these people do need more training to be able to understand the models and the principles of the type of MDD applied.

4.6 Modelers need to build more unequivocal models

In traditional software development, modelers build a set of diagrams to convey certain key aspects of a system. Requirements are translated to a technical solution according to architectural rules. It is often up to developers how to precisely implement an aspect described by a design. The UML offers a great degree of freedom. In practice, this freedom leads to inconsistent, incomplete and otherwise ambivalent diagrams [6]. The work of a modeler in MDD is different in the sense that it is not only to communicate functionality but also to directly implement that functionality by modeling. This implies that traditional trade-offs regarding design effort and detail and completeness of diagrams are no longer made. There are far fewer solutions that are correct.

One of the consequences of this shift in focus is the introduction of model validators. At the very least, the syntax of the model built by the developer has to be valid. A validator does not verify – it does not prevent the modeler

from developing a flawed model but it at least actively controls the correctness aspect of the output of a modeler. For a software architect, this has consequences in the sense that he is responsible for facilitating this focus shift. Modelers need to continuously be supported and corrected to learn to work with the DSL, the modeling tool and the validator. The model validator used in the project was not at all seen as a means to ensure or improve model quality. In the project, modelers struggled to understand the implications of their design choices and found it difficult to create models fit for code generation.

The models built by the modelers are both a direct translation of requirements and architectural rules, and also the technical implementation of these. Therefore, the architect needs to more closely monitor the modeling process and the models - in addition to the code - as they are developed, to ensure compliance to the software architecture. In a non-MDD project, the architect is less concerned with the design documents used by the developers. If the designers and developers find a method of properly communicating their ideas through a method of design, the architect is concerned with little more than the compliance to the architectural rules. In MDD, the design is strongly intertwined with the development process and closely related to the code generator for which the architect is responsible. Therefore, the architect is far more concerned with the design documents.

4.7 Modelers need to truly understand the DSL

One of the purposes of using a DSL is that domain experts at the client side can be more intimately involved in the development process because of the use of familiar terminology. Code generation from models enables a client domain expert to have direct impact on the system implementation. One of the implications of using the same models for client interaction as for system implementation is that modelers play a more central role in the requirements engineering process. In the project, modelers were present during requirement workshops drawing diagrams on a whiteboard to make direct translations of requirements to the DSL.

In order to ascertain that modelers shift their orientation from communication of technical implementation to modeling for direct code generation, they need to be fluent in the DSL of the project. Not only do they need to understand the implications of the use of specific elements for the working of the resulting code, they should be able to apply design patterns and should be able to set up comprehensible models that make sense from the domain perspective. These requirements naturally also hold for modelers involved in non-MDD development but are more stringent in MDD.

The necessity of modelers to truly understand and apply the DSL in accordance with the architectural rules and other modeling guidelines, broadens the architectural process in the sense that it requires additional effort from the architectural team. Apart from training the modelers, communicating and checking adherence to the DSL is primarily the responsibility of a software architect. Modelers need to be well trained in the DSL, UML in general and the use of patterns.

4.8 Less skill is required of most developers, more skill is required of some

The use of MDD profoundly impacts the work of developers. Compared to a non-MDD project, a developer has less freedom to interpret designs and architectural constraints due to the central role of the models and the strict guidelines that need to be adhered to in order to guarantee the system can be generated correctly.

In addition, the increased level of abstraction attained on the generated section of the system lessens requirements regarding developer skill. Aspects of a system that lend themselves particularly well for code generation are data related constructs such as CRUD[1] functionality. Much of the more straightforward code has therefore already been generated. However, to enable code generation, and to attain this level of abstraction, a substantial set of frameworks is used. Learning how to work with these frameworks can be difficult. And while implementation activities are mostly comprised of not too complex tasks, not all developers that would normally work on implementation of a system of similar complexity are able to cope with the more complex use cases or exceptions which require them to understand the interactions between all these different frameworks. In short, highly skilled developers are needed to implement the more complex parts. During the project, several junior developers were not able to cope with the complexity of the code that had to be developed. These developers had to be replaced by more capable or experienced developers.

An architect is responsible for communicating the more complex buildup of frameworks that is chosen for MDD development and therefore has spent more time to train new developers and to evaluate whether they are up to the task.

4.9 Collective code ownership is more important

The notion of 'collective ownership' stems from the rules of Extreme Programming (XP). It encompasses the notion that team members are collectively responsible for various aspects of the system under development, specifically system design. The key benefit that this practice aspires to obtain is the elimination of bottle necks for changes to certain

[1] Create, Read, Update and Delete

aspects of the system. In addition, people that are bottle necks may leave a project at any time, taking with them valuable information regarding an aspect of the system only they had deep understanding of. Developers tend to prefer to be responsible for their part of the system. Because less code is written and this code is more complex, developers must more often work with code that they did not author themselves. It is not possible to couple a developer to a particular use case. In fact, no developer should have objections to working at another use case or to have somebody change code related to a use case they initially authored. An additional benefit of collective code ownership is that it facilitates increased contact between the programmer and the modeler. In addition, the idea of collective ownership is as important for the models as they are essentially maintained by the entire project team.

4.10 Code generator is an additional application that needs to be developed and maintained in parallel

The applicability of code generation as a development method is limited to how specific the system requirements are. An 'off-the-shelf' generator is rarely capable of generating exactly what a specific customer wants. A specific methodology bundled with a code generator only allows for very specific applications to be built, in which case the client has little to say about the software architecture. Therefore, to facilitate the specific requirements of a large corporate client for a sizable system, a generator must evolve with both models and the code. Consequently, in addition to the development of the software system that is central in the project, a software architect is responsible for development and maintenance of a code generator. As the main system, the code generator has its own requirements, architecture, design and code. In the project, a separate team of developers was responsible for development and maintenance of the code generator. The main influence of this practice was found to be that changes have a larger impact. The impact of changes has to be checked in great detail so the impact analysis of a change requests is more detailed. However, a side effect of needing to more carefully examine changes is that the impact of changes is very clear and potential problems and defects are spotted much earlier. This prevents rework and thereby saves time.

4.11 Mismatches between meta-model domain reality and client reality need to be acknowledged

A benefit of MDD is that an existing meta-model can be used to quickly deploy applications within a certain domain. In the case of the project, a pre-existing meta-model of a specific aspect of the Dutch mortgage domain was the main motivation of applying MDD. However, an existing domain model might not correctly represent a domain in the way the client perceives it. Many assumptions made by experts in this domain regarding business processes and product-composition were not completely consistent with the business approach of the client. Redevelopment of the meta-model lengthened development time and hampered potential productivity improvements from the use of an existing meta model. In deciding between a detailed and a more generic meta-model describing a certain domain, the latter approach could be more feasible. The biggest gains of MDD can therefore be expected in stable domains or in domains in which much commonality exists.

4.12 More tooling is needed to support the MDD process

Considerations for deciding on tooling for software development include a trade-off between the standard tooling used by the development organization, specific project requirements and the wishes of the client and possibly the maintenance organization. Primarily responsible for this process is the software architect. The use of MDA requires more tooling than a non-MDD development process. As in most development processes, an MDD project requires a configuration and change management system, requirement-, defect-, time- and change tracking systems and modeling-, development- and testing- environments. However, a set of requirements are added to the tool selection process for supporting the DSL or reference model and extra environment for supporting the generator.

In addition to selecting candidate case tools and evaluation, team members must be trained to work with new tools. Traditionally, an architect will prescribe the use of only a subset of the functionality offered by the tooling, limit the use of the tool. Team member's use of the tooling must therefore be monitored. As described earlier, the project must deal with team members resisting to use particular tooling and perhaps needs to convince the client that a lesser known tool is indeed a proper solution. Finally, one of the lessons learned from the case is that adopting the use of an existing code generator for large-scale application of MDD is not feasible. As meta-model functionality changes, the generator and validator need to be altered. Using an existing generator would make that process more complex and time consuming.

The extra tooling employed in an MDD process make that an architect spends more time investigating, testing and explaining development tools. A rapid pace of development and the fragmented offering of state of the art MDD case tools requires an extensive evaluation process as a part of the inception of any MDD project.

4.13 Adherence to modeling guidelines by modelers can be enforced.

Tools that enforce adherence to architectural rules are not commonly used in industrial practice. For an architect it is therefore important to check architecture adherence throughout the development process. In MDD, adherence to architectural rules takes less effort because (1) modeling is more formal, (2) architecture is more formally defined and thus easier adhered to and (3) less steps of translation take place as models are directly translated to code by a code generator. In addition, a code generator is often equipped with a model validator, the model equivalent of a code parser. This validator checks syntactical adherence and provides some level of quality check. However, in the project, model verification was done by the architect.

4.14 Architectural descriptions need to be more extensive, formal and structured

The set of architectural artifacts used in the project is larger and more detailed than found in similar (non-MDD) projects of equal size. Architectural documentation includes:

- *The Software Architecture Document (SAD)* contains the most important architectural knowledge such as descriptions of actors and development tools, overviews of architecturally significant use cases and their realizations and descriptions of the distribution of the system over various nodes and its interaction with a selection of surrounding systems.
- *Supplementary Specifications* which contain additional requirements such as quality requirements of interfaces and general additional system requirements.
- *Interface documentation*
- *Modeling Guidelines* which contain a tool-independent description of how to describe functional requirements of an IT system using the DSL. Topics include naming and ordering of model elements and data modeling guidelines.
- *A team wiki* which among others contains an overview of how to work with the release process.
- *Separate Model Documentation* specifying meta-model functionality only used in specific models.
- *Additional Design Decisions* which contain an elaboration of certain design decisions such as how deep packages are nested or how certain functionality is split up.

Since more detailed descriptions of use cases are required in early stages of the project, documentation is reviewed more often. The central role of a document such as the modeling guidelines implies that more team members use and comment on contents. This requires more formal and complete descriptions which is better structured. Architectural documentation in the project was updated more frequently and up until later stages in the process.

5 Validity

The main threat to external validity is the use of a single case. This case study is the first iteration of a grounded theory study into industrial adoption of MDD. Future work includes analysis of other cases of industrial adoption of MDD. A threat to internal validity is the difficulty of distinguishing between the impact of adopting MDD specifically versus effects of adoption of new technology in general. During the data collection process, we repeatedly asked team members whether any MDD impact we identified was MDD related or specifically related to the adoption of a new technology.

6 Conclusion

For this study the application of MDD in a large-scale, industrial software development project was analyzed over a period of two years. Through a grounded theory approach, 14 factors which impact the architectural process have been identified. The lessons learned from this study provide an overview of how to prepare for large-scale adoption of MDD tools and techniques.

To prevent major rework, more effort should be planned up-front so that all requirements that impact the meta-model are known upfront and so that architectural design documentation is of sufficient quality, detail and completeness. Also, project scope should be more strictly defined to prevent the ease of generation of functionality to push the project beyond its original scope. Team members, including business engineers and maintainers should be properly trained, early in the development process, to be able to work with new CASE tools and should be prepared to understand and work with the DSL. Modelers need to be made aware of their more central role and their responsibilities as maintainers of the most central artifacts in the project. In addition, team members should be able to rework each others use cases, models and code. Software architect and project management should prepare and plan for two simultaneous projects as a code generator will be developed in parallel.

In the studied project, applying MDD with an existing meta-model did not provide the productivity gains expected by the software supplier. This can partly be ascribed to

the novelty of the approach and other adoption related influences such as training of team members. Also, using an existing meta-model might have been more effective in a domain in which more commonality exists. Other new, time consuming activities include development and maintenance of a code generator. We found that when applying MDD, the architectural process is more structured and rigorous and the role of the architect during MDD is broader and more demanding. When applying MDD in the large in industry, the software architect must be both an MDD expert and a realistic enthusiast. As one of the most important facilitators of the MDD process the software architect must evangelize MDD and continuously communicate the added value of MDD for each of the roles of the team members involved. This is not a trivial task. Developers will generally have few problems understanding the principles behind MDD and how it might benefit their work. Modelers play a far more central role and need to be supported with the responsibilities that come with that role.

7 Acknowledgment

The authors would like to express their gratitude towards the involved organizations and the interviewed project members for their time and effort.

References

[1] R. Farenhorst, J. F. Hoorn, P. Lago, and H. van Vliet. What architects do and what they need to share knowledge. Technical Report IR-IMSE-003, VU University Amsterdam, April 2009.

[2] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

[3] W. Heijstek and M. R. V. Chaudron. Empirical investigations of model size, complexity and effort in a large scale, distributed model driven development process. *Software Engineering and Advanced Applications, Euromicro Conference*, pages 113–120, 2009.

[4] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[5] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[6] C. Lange, M. R. V. Chaudron, J. Muskens, L. J. Somers, and H. M. Dortmans. An empirical investigation in quantifying inconsistency and incompleteness of UML designs. In L. Kuzniarz, Z. Huzar, G. Reggio, J.-L. Sourrouille, and M. Staron, editors, *Proceedings of the IEEE Workshop on Consistency Problems in UML-Based Software Development II*, pages 26–34. IEEE and Department of Software Engineering and Computer Science of Blekinge Institute of Technology, 2003.

[7] A. MacDonald, D. M. Russell, and B. Atchison. Model-driven development within a legacy system: An industry experience report. In *Australian Software Engineering Conference*, pages 14–22. IEEE Computer Society, 2005.

[8] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. Object Management Group, Framingham, Massachusetts, June 2003.

[9] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–161, April 2009.

[10] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[11] M. Staron. Adopting model driven software development in industry - A case study at two companies. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *LNCS*, pages 57–72. Springer, 2006.

[12] A. C. Strauss and J. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Sage Publications, Inc, 2nd edition, September 1990.

[13] The Middleware Company. Model driven development for J2EE utilizing a model driven architecture (MDA) approach: Productivity analysis. Technical report, The Middleware Company, June 2003.