Chapter 8

Analysis of the Consequences of Model-Driven Development for Global Software Development

The promotion of models over code to first-class entities is a central theme of Model-Driven Development (MDD). In theory, this has a profound impact on the architectural process and the work of the software architect. MDD is emergent in GSD projects and the main challenges in GSD include difficulties to share knowledge, to align tasks and to obtain and maintain a shared mental model. In theory, MDD has the potential to mitigate some of these difficulties. In this chapter we aim to understand (1) how the application of MDD tools and techniques affects the architectural process and (2) how this relates to the problems commonly associated with GSD.

This chapter is based on the following publication:

Werner Heijstek and Michel R. V. Chaudron (2010) **The Impact of Model-Driven Development on the Software Architecture Process.** *In Proceedings of the 36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2010) pages 333–341, Lille, France*

8.1 Introduction

Recent studies focus on decision making in the process of software architecting (e.g. Kruchten et al., 2005). As a result, the practice of software architecting and the position of the software architect in the software development process are under investigation (Clements et al., 2007, Farenhorst et al., 2009). Also in Global Software Development (GSD) projects, where software architecture is often transferred from one development location to the other, the role of the software architect is not clearly defined. In distributed settings, software architects have to interact with software developers through (often great) geographical, temporal and socio-cultural distances. So far, it has been unclear how these challenges are coped with in industrial practice.

At the same time, MDD tools and techniques claim improved team communication, better (or even automatic) architecture compliance and resulting productivity gains. Application of these tools demands a shift in boundaries between traditional roles in the software development process as the code is no longer the central artifact – the model is. This would have 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. The architect also has an increased responsibility to maintain consistency throughout the development process.

MDD is emergent in GSD projects (Jiménez et al., 2009). The main challenges in GSD include sharing knowledge, to align tasks and to obtain and maintain a shared mental model (Cannon-Bowers et al., 2001, Espinosa et al., 2001). In theory, MDD has the potential to mitigate some of these difficulties. Using models as the central artifact enables teams to use software architecture and design to direct team composition, development process and even communication structures. In this chapter, we aim to understand (1) how the use of MDD tools and techniques affects the architectural process and (2) how these impact the problems commonly associated with GSD.

The outline of this chapter is as follows: Section 8.2 contains an overview of the study objective and the data collection and analysis methods. Section 8.3 outlines related work. Sections 8.4 and 8.5 discuss the results and the specific impact of MDD on GSD. Finally, Section 8.6 contains conclusions and future work.

8.2 Objectives and Data collection and Analysis Methods

In this chapter, we address **RQ3** (Section 1.3). One of the main implications that Šmite et al. list in their recent structured literature review of empirical evidence in GSD is that there exists a *"gap regarding in-depth empirical investigations addressing particular aspects of software engineering."* They continue to note that *"thus, future research ought to evaluate different practices, methods and techniques rather than mainly focus on managerial problem-oriented lessons learned."* We have motivated that existing studies hint at the positive influence that MDD could have on the GSD process. Therefore, it is not only

necessary to understand what the impact is of using MDD tools and techniques in the context of GSD in general. Specifically, we will address the impact in the context of the problems associated with GSD. The research question addressed in this chapter can therefore be formulated as follows:

> *How does the application of model-driven development tools and techniques impact the problems associated with Global Software Development?*

The subject of our analysis is the case outlined in Chapter 7. To analyze our data, we apply the grounded theory approach. 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 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 also extensively interviewed other team members including designers, developers, lead developers, project leaders, a test manager, a system analyst and an estimation & measurement officer involved in sizing and tracking the project.

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 these distilled lists to validate our interpretations. We repeated the coding process and updated factor descriptions in the same way.

8.3 Related Work

This section addresses related work regarding the generic impact of MDD on the software architecture process and the (potential) benefits of application of MDD in the context of GSD.

8.3.1 General Impact on the Software Architecture Process

A commonly adopted framework for MDD is Model-Driven Architecture (MDA, Kleppe et al., 2003, Object Management Group, 2003b,a). MDA is a specific MDD approach that employs UML, MOF (Object Management Group, 2006) and XMI (Object Management Group, 2007). A study by The Middleware Company (2003) of the application of MDA specifically mentions *"architectural advantages."* The study explains that by application of MDA, an architect is forced to spend more time designing an architecture due to the necessity to also model 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. Advantages of the 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 (2006), 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. Application of MDD is therefore expected to alter the role of the software architect. Farenhorst et al. (2009) 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 will likely 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). A DSL is a modeling language that, by design, is particularly fit to express concepts related to a specific field or e.g. a branch of business.

8.3.2 MDD in Global Software Development

On the surface, MDD appears to have the potential to mitigate some of the difficulties that are associated with GSD. Using models as the central artifact would enable teams to use software architecture and design to direct team composition, development process and even communication structures. The main challenges in GSD include sharing knowledge, to align tasks and to obtain and maintain a shared mental model (Cannon-Bowers et al., 2001, Espinosa et al., 2001). The sources of these challenges are to be found in the three types of distance that are introduced in GSD projects: geographical distance, temporal distance and socio-cultural distance.

The model-centric nature of MDD and some of the requirements that lie at its foundation could have a positive influence on the problems associated with GSD.

Limited related work exists that explicitly addresses this conjecture. Nevertheless, evidence of the potential positive influence of MDD on GSD can be found in various studies. For example, that the higher level of abstraction inherent to MDD facilitates more effective stakeholder communication is explicitly mentioned in the context of GSD in many other studies. For example:

- *"explicit, shareable models and descriptions [...] facilitate collaboration between developers on an abstract level"* (Pahl, 2005),
- *" great advantages of [our MDD language] in the context of global software development [include] a common understanding of the software being developed"* (Heistracher et al., 2006),
- *"advantages of a unified, model-driven approach to requirements elicitation include significantly improved communication"* (Berenbach and Gall, 2006).

In addition, the advantages of the use of common tools on GSD is often referred to. For example:

- *"[MDD] tool vendors develop more and more tools to be applied during architecture development."* (Spanjers et al., 2006);
- *"Enforcing common tools and processes makes collaboration much easier "* (Lings et al., 2007);

Only limited more detailed evidence is available. For example, Lester and Wilkie (2004) present an empirical evaluation of the selection of a commercial CASE tool that supports UML in the context of a large GSD project. Lester and Wilkie specifically aim to address the problem that *"the lack of synchronization between design models and source code, for a development team working in different time zones, can lead to strained relationships between the geographically disparate sites."* MDD brought forth a host of tools that support model-code correspondence in general and code generation in particular.

Another example of more detailed analysis of the role of MDD-related tools and techniques is the work by Clerc et al. (2007). In this study, a case is reported in which distributed team organization was formed strictly along the lines of an architectural design — including dependency and subsystem-related constraints on communication. The authors claim that this helped GSD-related problems they define as *"difficulty to build a team."*

Application of MDD requires the early and complete definition of an architectural framework. This is in line with the requirement that an architecture must be sufficiently mature to be able to distribute team composition, development process and communication structures (Mullick et al., 2006, Conway, 1968). In line with the previous finding, Clerc et al. (2007) also found that *"alignment via architecture"* is beneficial. They found that in the same case, *"alignment of tasks and responsibilities [was] mainly done via the architecture,"* they go on to exemplify this by noting that, *"[r]equirements [were]*

assigned to subsystems, which [had] dedicated resources assigned." Apart from a similar requirement regarding early architecture maturity, this model-centric method of project management is much in line with the premise of MDD in which models are even more central.

Another specific study of benefits of MDD techniques to GSD problems was executed by Andaloussi and Braun (2006) who outline their experiences in developing a model-driven test framework based on the the UML 2 testing profile (Schieferdecker et al., 2003). In their effort, Andaloussi and Braun specifically sought to obtain communication benefits for GSD teams: *"The advantage is to represent the system and its tests through one single notation."* In preliminary findings of a case study in which they implemented their framework, they found that through using the test framework they *"[overcame] the language barriers in releasing the test specification from [a] textual description filled with buzzwords and jargon."* In addition, they note that they made *"nearshore more independent from offshore, in avoiding initial training phases and requiring only standard skills (UML, U2TP and TTCN-3)."* Their use of a DSL made it easier to distribute the architecture in small components, which in turn increased comprehensibility.

8.4 Results

In this section, the results of the analysis of the case (as outlined in Chapter 7) are discussed. Structured around the three core concepts behind MDD (Section 1.1.5), we will discuss the influences of application of MDD on the GSD that were found in the project under study. We will particularly address the ramifications for the software architecture process. An overview of the implications of adoption of MDD in GSD on the software architecture process is presented in Figure 8.1.

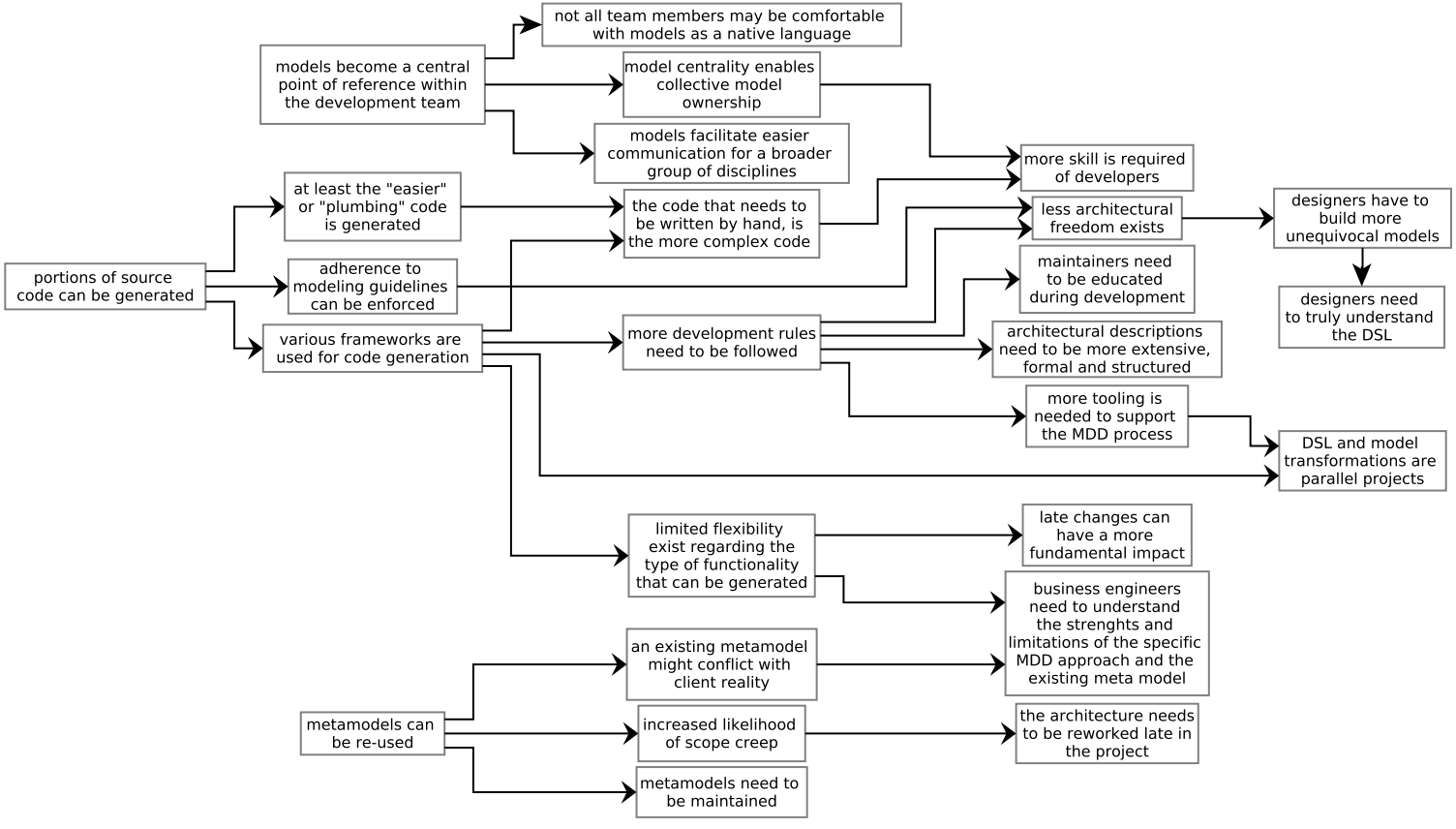

Figure 8.1: *Factor integration graph (⟶ denotes cause and effect)*

8.4.1 Model-Centrism

Source code is not easy to communicate even between those who understand the programming language. In addition, source code represents one of many representations of the system to be built. In traditional software development, each stakeholder has his own preferred representational conventions to describe the system. In MDD, models, rather than code, are treated as first-class entities. As models are abstractions of more technical details, the potential proportion of team members that understand the models is larger.

All team members noted that intra-team communication was much easier because the models were used as a single point of reference was used. Models were numbered and requirements engineers, the architect and even project management would refer to that same model number to discuss an particular issue that was relevant to their role. As a result, use of models as a common language eases communication and enables a larger group of stakeholders to participate in implementation-related discussions. The strongest evidence exist in literature for this particular effect of application of MDD.

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 becomes less laborious. 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 different disciplines were familiar with the models. This translated to fewer traveling back and forth between the offshore and onshore locations than normally would be the case in projects of similar size and complexity.

In the following sections we discuss two other effects that model-centrality had on the case. First, the transition to models as a common language is easier for some disciplines than for others. Particularly, business analysis were reluctant to work with software CASE tools. Second, as more stakeholders are directly involved with the models (now the central development entities), "collective ownership" becomes an important development concept.

Common Language as a Challenge

In the case, 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 what they referred to as "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 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.

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 of requirements to architecture to design to source code. And while MDD does not completely remove the need for translation, the use of models as a central language 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. However, the central use of models requires team members from all disciplines to work with the same language, concepts and tools.

Collective Model Ownership

The notion of "collective ownership" stems from the rules of Extreme Programming ((XP), Beck, 1999). 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 bottlenecks for changes to certain 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.

The architect explained that all designers need to be able to develop and expand most models. While some of the more complex models were still assigned to one or two designers, the majority of models were worked on by a variety of team members. Various team members explained that they enjoyed working on various different models. They were also convinced that model quality improved because of this practice. Having more team members working at the same model increased the chance of spotting defects. Developers explained that they were not used to work with each others code and that they needed to get used to having other people work with "their" models and code.

An additional benefit of collective code ownership is that it facilitates increased contact between the programmer and the designer. However, the diagrams were never branched so enable that designers could work in parallel. In this case, no tool support existed for model version control. Although some tools are available (e.g. EMF

Compare (Brun and Pierantonio, 2008) and DSMDiff (Lin et al., 2007)), these are not easily integrated with existing MDD tools.

8.4.2 Code Generation

In this case, a source code generator was developed to generate code. Using this generator, a significant portions of source code (90 percent) of source code was generated. We found this to have four direct consequences:

First, at least all the "easy" code is generated. All of the "hand-written" code, therefore, is more difficult to write. This requires more skilled developers. Second, standardization on mature software components as well as integrating generated code and hand-written code, requires the use of a variety of frameworks. Increasing the amount of frameworks involved in the software architecture requires an increasing understanding of the complexity of the interaction between these frameworks. This requires even more skilled programmers. Third, modeling can no longer be done haphazardly (Lange et al., 2006). On the contrary, adherence to modeling guidelines must be enforced if models are to serve as the basis for code generation. Fourth, if code is generated then a code generator needs to be developed and maintained in parallel with the original project. The generator is a separate project with its own stakeholders.

These four consequences lead to a variety of implications. The first and second consequence directly require more skilled developers. The use of a variety of frameworks and a code generator require more structured models, a more formal development process, documentation and an increase in tooling. The use of these frameworks also limits the flexibility regarding the type of functionality that can be generated. In addition, developer compliance to architectural rules is no longer optional. Furthermore, the increase in development rules require software maintainers to be involved in the implementation process at an earlier stage than would have been the case in a traditional development process. The fourth implication, the generator being a separate project, greatly increases project complexity. These implications are discussed in the following sections.

More Skilled Developers are Required

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.

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. In addition, to enable code generation and to attain this level of abstraction, a substantial set of frameworks is used. Understanding how these framework interact can be a difficult process. As a result, not

[1] Create, Read, Update and Delete

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. In short, highly skilled developers are needed to implement the more complex parts — which form the majority of the "hand-work."

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

Strict Quality Assurance for Modeling

Developers explained they had less freedom to interpret designs and architectural constraints due to the central role of the models and the strict guidelines that needed to be adhered to in order to guarantee the system could be generated correctly.

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 done more formally, (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, the code generator used in this project was equipped with a model validator, the model equivalent of a code parser. This validator checks syntactical adherence and provides some level of quality check. In the case, model verification was done by the architect.

More Extensive and Structured Architectural Descriptions are Required

The set of architectural artifacts used in the project is larger and more detailed than found in similar (non-MDD) projects of equal size. The sources of architectural knowledge available during the project are detailed in Table 8.1.

Next to the sources in Table 8.1, architectural knowledge exists which is not captured in any artifact but the system itself. In interviews, project members refer to design decisions which are visible in the models but which are not explicitly documented. Project management did not allow for the time to explicitly document all design decisions due to time constraints.

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 in comparison to non-MDD projects.

Table 8.1: *Architectural Artifacts Available in the Case Project*

artifact	description	contents
Software Architecture Document (SAD)	The most important architectural knowledge is described in this document. The SAD is used by all team members except the testers. The author is the software architect.	Description of actors and development tools; List of architecturally significant use cases and their realizations (use case view); Overview of logical layers (logical view); Definition of communication and process principles that are relevant for the software architecture (process view); Description of the distribution of the system over various nodes and its interaction with a selection of surrounding systems (deployment view); Architecture of the source code — layering, frameworks and best practices (implementation view); Description of transformation of the UML design model to various data aspects of the software architecture (data view)
Supplementary Specification (SS)	Requirements outside of the requirements described in the use cases.	Quality requirements of interfaces; Additional system requirements
Interface Documentation	Per interface documentation. The author is the system analyst.	Request message specification; Reply message specification; Web Services Description Language (WSDL) specifications; List of related Use Cases
Modeling Guidelines	A tool-independent description of how to describe functional requirements of an IT system using UML. The author is the software architect.	Naming and ordering of model elements; Data modeling guidelines; User interaction modeling guidelines (flows, sub flows, authorization, use of data, use of services, decisions, constraints, composite operations); UML Profile Reference
Wiki	The Wiki of SourceForge Enterprise Edition 4.4 is used. Authors include most project members.	Tips and tricks to set-up your code environment correctly and how to solve problems; An overview of the release cycles of all parallel working teams is managed; An overview of how to work with the release process
Separate Model Documentation	A set of documents which elaborate on some of the elements from the meta-model that are only used in specific models. The authors are the maintainers of the respective models.	Style guide for the screens associated with this use case; Data sources overview for use case
Design Decisions	Elaboration of certain design decisions. This document is based on the modeling guidelines and was created before the system was built. The author is the System analyst.	How deep packages are nested; How certain functionality is split up

More Tooling Is Needed to Support the MDD Process

From their literature review of MDD, Mohagheghi and Dehlen (2008) conclude that suitable tools are of fundamental importance for MDD to succeed. These tools must be selected carefully for fitness to meet requirements and must fit into an organization's existing chain of tools. Factors in 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 for large scale, specific applications. As meta-model functionality changes, the generator and validator need to be altered.

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.

Designers Have to Build More Unequivocal Models

The architect found that he continuously needed to support and correct the model designers to learn to work with the DSL, the modeling tool and the validator. Designers struggled to understand the implications of their design choices and found it difficult to create models fit for code generation. The architect played a central role in the continuous training that designers required. Being located onshore, providing this training to the offshore development team was quite a challenge. Daily intensive (video) training sessions were held.

In traditional software development, designers 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 (Lange et al., 2003). 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.

Generator is a Parallel Software Development Project

The architect found that parallel development of a code generator quickly grew into a separate project with its own architecture, stakeholders and e.g. defect management system.

The applicability of code generation as a development method is limited to how specific the system requirements are. An "off-the-shelf" set of model transformations is rarely capable of generating exactly what a specific client 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, model transformations 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 requirement changes have a larger impact on the development process. The impact of changes has to be checked in great detail so the impact analysis of a change requests is more detailed. However, according to the architect, a side effect of needing to more carefully examine changes was that the impact of changes was very clear and potential problems and defects are spotted much earlier. This prevented rework and thereby saved time.

Late Changes Can Have a More Fundamental Impact

The use of code generation comes at the cost of greater standardization. If certain functionality is not supported by a meta-model, it can only be generated by extending the meta-model, the model transformations and possibly the DSL. This is potentially more time-intensive than adding the functionality by hand. Late changes to the meta-model may therefore 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 needed.

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. Because 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

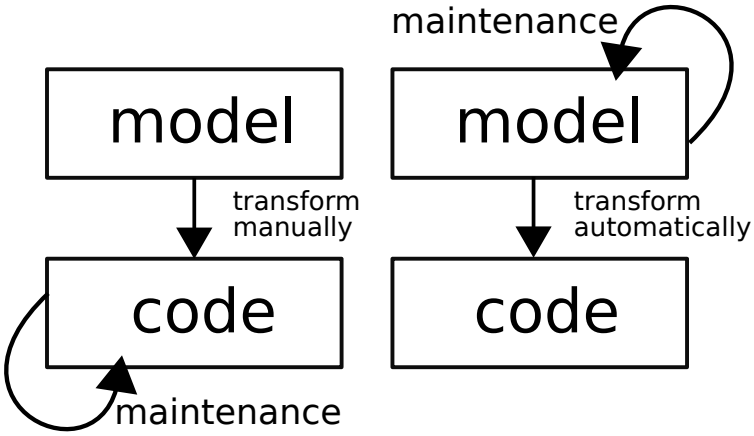

Figure 8.2: *MDD versus non-MDD maintenance (adapted from Van Vliet, 2008)*

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 model transformations. 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. Cabot and Yu (2008) argue for extending of MDD methods with improved requirements techniques.

Maintainers Need to be Involved During Development

Maintenance of software constructed using MDD tools and techniques is different in that not the code, but the models need to be altered 8.2. To ensure that models and system stay in-sync, it is imperative that maintainers are trained to understand the DSL, the models, the model transformations, the generation process employed and the integration between the generated code and the hand-written code.

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 from the maintenance staff. This knowledge is best obtained by close involvement of the development process.

8.4.3 Model Reuse

Domain-specific models can be reused for new software systems within the same domain. For this case, a meta-model was created before the project started. We found

three consequences of the use of that meta-model:

First, the objective was to have that meta-model expanded and kept separate from this project so that it could be used in future projects. This requires external stakeholders that ensure that the meta-model stays generalizable. No resources were allocated for such a role. Second, this existing meta-model, which was created by domain experts, did not represent the client's perception on that same domain. Third, if a client initially only requires a subset of functionality that an existing meta-model offers, it might seem tempting to expand a system's scope to include "things the meta-model already can do." The second and third consequences are discussed in more detail in the next sections.

Increased Likelihood of Scope Creep

In this case, project management regularly budgeted client requests for specific meta-model functionality at zero hours of person effort. In this case, severe project time and budget overruns could in great part be ascribed to this practice.

Scope creep occurs when system functionality expands beyond the initial project objectives. Any software development project will meet changing requirements and 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 a additional functionality. The use of MDD can make scope creep more likely for two reasons: First, extending functionality can be easier than in non-MDD development. This specifically pertain changes already supported by the meta-model. 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 furthermore requires a supplementary iteration of the analysis of the business modeling as added functionality might impact existing business processes of systems in the environment and could 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 introduced 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.

An Existing Meta-Model Might Conflict with Client Reality

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. This model was created in concordance with business analysts with extensive experience in the Dutch mortgage domain. 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. This either stemmed from incorrect assumptions or from domain evolution. 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 (with limited domain evolution) or in domains in which much commonality exists. This same problem can occur at the DSL-level. The strong link between the DSL and the domain benefits development by domain experts, but backfires when that domain evolves. Various studies propose methods for addressing domain model evolution (Deng et al., 2006, Sprinkle and Karsai, 2004).

8.5 Impact of MDD on GSD

In this section, we discuss the advantages and the disadvantages that the identified impacts of use of MDD have on GSD. In their recent structured literature review of empirical studies in GSD, Šmite et al. (2010) give an overview of GSD challenges and the best practices that are so far known. In Table 8.2, these are linked to the MDD impacts that were discussed in the previous sections. The first two columns in this table have been taken from Šmite et al.

Most of the best-practices associated with GSD are directly affected by process changes that are found with use of MDD tools and techniques. As discussed earlier, many studies hypothesized that the communication benefits that a DSL entails would benefit the GSD process. The increased communication efficiency that was found in the case enforces — or at least positively impacts — many of the best practices in the overview of Šmite et al.. A DSL provided for a common language and therefore mitigated some of the problems associated with what is commonly regarded as the toughest of the three distances (Herbsleb et al., 2000): socio-cultural distance. However, models also made for the richer communication that GSD needs. In addition, the close interaction with the client through use of a DSL, enabled the incremental short-cycle development that is beneficial for GSD.

However beneficial all these communication-related benefits are, the evidence that

MDD mitigates some of the problems of GSD reaches further. The requirement of shared model ownership implies that a centralized project repository — essential in GSD — had to be employed. Furthermore, use of MDD required a more extensive and more explicitly defined architecture. This enabled easier task distribution based on architectural decoupling, which is one of the most concrete best practices for GSD (Herbsleb et al., 2000). Also, an increased reliance on tools that accompanies the use of MDD enabled a more reliable infrastructure through a more formal method of working. Still, integrating these tools in the existing chain of tools was a challenge and so was training people to use them. MDD was not found to have any direct impact on the "synchronous interaction" best practice that Šmite et al. listed. While MDD does not require this type of interaction it does not inhibit it either. A potential drawback of MDD when used in GSD is the training. Programmers, designers but also management needs to be educated so to understand the MDD paradigm. When assembling an offshore team it proved be difficult to assess the extent to which candidates had the required skills. Furthermore, the short iteration cycles mentioned in Table 8.2 are needed to address the problem of process unclarity or the lack of awareness of either the process followed or current process status (Espinosa et al., 2001, Levesque et al., 2001, Carmel, 1999, Mockus and Herbsleb, 2001). Some organizations tailor their process prescriptions to cater for GSD (Heijstek et al., 2010) but this does not solve the awareness problem. MDD requires that a strict process is followed. This process was defined early in the elaboration phase in concordance with the entire team. As the model transformations would evolve with the diagrams, offshore team members were aware of the status of their work, the work of the onshore team and the status of the project as a whole.

8.6 Conclusions and Future Work

Using MDD tools and techniques fundamentally impacts the software development process in general and the analysis and design phases in particular.

All team members in this case elaborated on how the use of models as a common language eased communication between team members in general and between on- and offshore teams in particular. In addition, models enabled a larger group of stakeholders to participate in implementation-related discussions. This translated to fewer traveling back and forth between the offshore and onshore locations than is normally the case in projects of similar size and complexity.

We found that the use of a common language mitigated some of the problems associated with what is commonly regarded as the toughest of the three distances (Herbsleb et al., 2000): socio-cultural distance. In addition, MDD techniques in general and shared model ownership in particular forces more frequent interaction between more team members.

While MDD enforces most GSD best practices that are currently known in literature,

some drawbacks exist. Staffing requirements include team members that are willing to work with models and model CASE tools and highly skilled developers. However, in GSD contexts, it is not always possible for an architect to influence development team composition. In addition, the application of MDD requires more formal working procedures in terms of e.g. more extensive and detailed design documentation and models that strictly adhere to modeling guidelines. The architect played a central role in the continuous training that team members required. Being located onshore, providing this training to the offshore development team was quite challenge.

Table 8.2: *Relating GSD best practices (Šmite et al., 2010) to MDD-related practices*

Practices	Advantages	Impacted by MDD	GSD Impact
• F2F meetings • temporal collocation • exchange visits	• Trust • cohesiveness • effective teamwork	☒	• Intra team communication was said to be more efficient with MDD because of DSL • Less travel was required since communication was clearer
• Centralized project repository • common configuration management tool support	• Awareness • process transparency	☒	• Central repository was required for collective model ownership • More tools were used
• Effective and frequent synchronous communication	• Trust • cohesiveness	☒	• Communication was said to be more efficient with MDD because of DSL
• Reliable infrastructure • rich communication media	• Effective communication	☒	• Models made for richer communication as they were included in meetings. • More tools were needed. These allow stricter work procedures. The introduction of new tools also introduces some uncertainty. • The development process was more formal.
• Synchronous interaction	• Effective teamwork	☐	
• Task distribution based on architectural decoupling and low dependencies across remote locations	• Effective teamwork	☒	• A greater proportion of the architecture was explicitly defined • Any architectural decoupling was more straightforward to enforce as the implementation was closer to the architecture • Requirements were clearer
• Incremental short-cycle development	• Early feedback, capability evaluation	☒	• Code generation enables faster development • Earlier feedback is obtained because of closer client interaction through the DSL