Chapter 7

Contrasting Model-Driven Development with Code-Centric Development

MDD is seen as the natural continuation of the trend of raising the level of abstraction at which software is developed. Consequently, in the past decade, there has been increasing interest in MDD in both industry and academia. In addition, MDD is emergent in GSD projects. Nevertheless its impact on the development process in large-scale, industrial practice is not yet clear and empirical validations of adoption of MDD tools and techniques are scarce. This chapter therefore addresses how the characteristics of a large scale, industrial model-driven development project in the context of global software development compare to non-MDD projects. We specifically focus on the quantification of process metrics.

This chapter is based on the following publication:

Werner Heijstek and Michel R. V. Chaudron (2009) **Empirical Investigations of Model Size, Complexity and Effort in Large Scale, Distributed Model-Driven Development Processes — A Case Study.** *In Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009) pages 113–120, Patras, Greece*

7.1 Introduction

In the past decade, there has been increasing interest in MDD (Selic, 2003) in industry and academia. MDD is also emergent in GSD projects (Jiménez et al., 2009). The quality and productivity benefits claimed for the use of MDD triggered many studies to advance MDD practices. Nonetheless, few empirical studies are available that study the impact of applying MDD on industrial software development As a result, the general impact on software development of using MDD is unclear.

The structure of this chapter is as follows: Sections 7.2 and 7.3 elaborate on the study objective and related work. Section 7.4 explains the case study design. Section 7.5 discusses the results and the conclusions and future work are presented in Section 7.6.

7.2 Objectives

In this chapter, we address **RQ3** (Section 1.3). This question aims to explain how increasing model-centrality impacts the problems associated with GSD. To this end, the differences between code-centric and model-driven software development are analyzed.

In this chapter in particular, we aim to add to the limited experience reports in which the specificities of MDD cases in general and cases of MDD adopted in GSD context in particular, are reported. As established, it is important to investigate industrial cases of MDD to add to the scarce literature. Such investigation is also beneficial for benchmarking purposes and to evaluate the impact of the process and techniques used in general. To these ends, we pose the following research question:

> *How do the characteristics of a large scale, industrial model-driven development project in the context of global software development compare to non-MDD projects?*

This question is divided into sub-questions regarding key characteristics regarding the software development process in general and models in particular:

1. What types of diagrams are used?
2. How is effort distributed over the "classic" development phases?
3. How big and how complex are these models?
4. How does model size grow over time?
5. Do model size and complexity impact defect count?

7.3 Related Work

The main hypothesized benefits to be gained from adoption of MDD are

1. increase in productivity,
2. improved code-quality,
3. improved re-usability and
4. improved maintainability.

The main two principles that enable these benefits are (1) provision of better abstraction techniques and (2) facilitation of automation (Mohagheghi and Dehlen, 2008, Staron, 2006, Kleppe et al., 2003). The software architect fulfills a central role in ensuring that these potential benefits are actually obtained. The software architect is instrumental in enabling correct model transformations such as code generation. His objective of complying to non-functional requirements implies a careful consideration of available modeling case tools and a leading role in the design and application of a Domain Specific Language (DSL) (Van Deursen et al., 2000).

We focus on productivity. In this section, an overview is presented of related work. First, we elaborate on the state of empirical research in MDD tools and technique application. Second, the impact on productivity is discussed.

7.3.1 State of Empirical Research

Empirical evidence regarding the benefits of application of MDD tools and techniques is sparse. Literature regarding MDD in large-scale, industrial projects often describes processes in which legacy systems are reverse engineered to MDA (e.g. Anda and Hansen, 2006, Reus et al., 2006, Fleurey et al., 2007). Reports are mostly qualitative (Staron, 2006, Raistrick, 2004, Baker et al., 2005).

An extensive review of literature regarding MDD (published between 2000 and 2007) was executed by Mohagheghi and Dehlen (2008), the results of which have been summarized by Hutchinson et al. (2011):

- Most studies of the 25 selected papers were experience reports from single projects;
- MDD was applied in a wide variety of organizations; methods of code generation varied;
- MDD techniques are very dependent on tooling;
- productivity impact varied *widely* and more empirical studies that evaluate MDD are needed.

Concluding their work, Mohagheghi and Dehlen explicitly recommend that *"future work for evaluation of MDE should focus on performing more empirical studies, improving data collection and analyzing MDE practices."*

7.3.2 Impact on Productivity

Most empirical studies regarding MDD address questions regarding efficiency (White et al., 2005). Still, only few studies offer enough data to quantify and baseline productivity (and quality) in industrial MDD projects (Shirtz et al., 2007, Weigert et al., 2007). Anecdotal evidence in literature claims that adoption of MDD has hampered productivity as much as 27 percent (MODELWARE D5.3-1, 2006) and improved productivity as much as 800 percent (Baker et al., 2005). A case study by MacDonald et al. (2005) 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 because of 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.

In their recent multi-method study of the state of the practice of MDD, Hutchinson et al. (2011) specifically addressed the perceived impact of MDD activities on productivity and maintainability. They found that the largest impact was not code generation or meta-model reuse but *"the use of models for understanding a problem at an abstract level."* The second greatest impact was thought to be *"use of models for team communication."*

7.4 Case Study Design

In this section we describe the case study design.

7.4.1 Context

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 department responsible for development of the system has extensive experience with building tens of software systems for the financial sector as well as experience with global software development.

A function point analysis that was based on the requirements was executed in the early stages of the project. It reported a total of 1,973 function points to be built. 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). Only four team members had experience with a previous project in which MDD techniques were applied. Total project duration was 24 months. RUP was used as development process. The RUP is an adaptable process framework that is architecture-centric and risk-driven and can be used for iterative software development (Kruchten, 2003b). 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 designers, development was done at both locations and testing was done in India. The Dutch project leader was the main point of contact to the client in the Netherlands.

7.4.2 Specificities of the MDD Approach

In the development process, a DSL with strict modeling guidelines is used. These guidelines address the dynamic aspects of the system and are based on UML 2. The guidelines are developed without code generation in mind. The rationale behind using UML for this reference model was that UML is more widely known than other suitable candidates such as the Business Process Modeling Notation (BPMN). Model consistency is enforced in two ways. First, the developers are restricted by the constraints imposed by the UML meta-model. Second, a model validator is used. This validator checks syntax and conformance to the UML meta-model. When code is generated, the models are validated first. However, complete validity of the models is not so much the goal as a working result. Source code is generated by using a code generator. This generator is realized through a combination of open source libraries. During the project, developers work at extending and enhancing the code generator. A general overview of system components is depicted in Figure 7.1. The system consists of two parts. Part one is a complex web-based system for user interaction, this system contains a web service client. Part two is a web service that enables existing systems to request information. The system domain model is formally modeled in UML and completely generated into a Java implementation. The model syntax consists of

- classes and properties,
- property types and their names and documentation,
- associations between classes and
- required fields and constraints on classes.

Inside these entities, no other behavior is modeled. The classes contain no operations. Screens that are deemed suitable to be modeled such as “input screens” and “selection

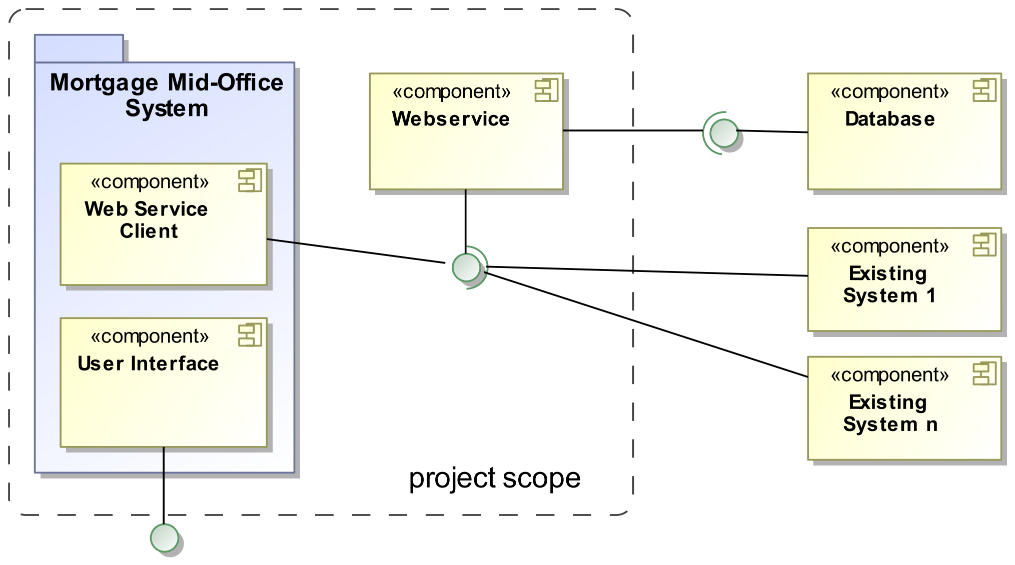

Mortgage Mid-Office System
«component» Web Service Client
«component» User Interface
«component» Webservice
«component» Database
«component» Existing System 1
«component» Existing System n
project scope

Figure 7.1: *Overview of case system components*

screens”, are described in UML and completely generated to source code. For more complex, custom screens, the syntax of the DSL does not suffice. These screen are fully or partly hand-coded. The web service client is completely generated from the UML model. Some parts of the business logic layer can be fully generated from the UML model. Other parts are fully hand-coded.

Initially the target implementation language was a high level, business-oriented programming language that would have been relatively easy to maintain. Due to limitations imposed by using this language, later in the project it was decided that Java 2 Enterprise Edition was to be generated from the models. Both The Spring Framework[1] and Hibernate (Bauer and King, 2004) are used for target development and the tools used are Eclipse[2], JBoss[3] and MagicDraw[4]. The final application is to operate on the IBM WebSphere[5] platform and will use a DB2[6] database. Approximately 90 percent of the code is generated, the remaining 10 percent is written “by hand”.

[1]http://www.springsource.org/
[2]http://www.eclipse.org/
[3]http://www.jboss.org/
[4]https://www.magicdraw.com/
[5]http://www-01.ibm.com/software/websphere/
[6]http://www-01.ibm.com/software/data/db2/

7.4.3 Data collection

We collected quantitative data regarding process and models from various sources. The models were collected from a Subversion repository. Effort and defect data was collected from SourceForge Enterprise Edition. For this study, we use the notion of "diagram groups." Each screen in the application that was created consisted of a set of diagrams. After collecting all diagrams from Subversion, metrics were extracted using SDMetrics (Wüst, 2009). This process was automated using a set of Bash and Perl scripts. Metric data is available on a per diagram basis whereas effort and defect data was only available on a per-diagram group basis. Therefore, the resulting metric files were aggregated per diagram group so that effort and defect data per model could be combined. In this project each diagram group was contained in a separate file.

7.5 Results

UML diagrams were created using MagicDraw 14.5[7]. The DSL specification required diagrams to be grouped together. A total of 119 diagram groups contain a total of 386 diagrams. A bar chart of the UML diagram types (Figure 7.2) shows that activity diagrams are most abundantly used, followed by class and use case diagrams. The reason for the plenitude of activity diagrams is that the development of the models is user interface centric. This means that the process flow of the process that the system will support is captured in the activity diagrams as a set of screens. The process flow of modeling is chosen so that during maintenance changes in the business process can easily be translated into changes to the models. In total, 104 diagram groups contain one or more activity diagrams, 32 diagrams groups together contain 150 class diagrams and all use case diagrams are spread over just two diagram groups. The average model consists of one or two activity diagrams and zero or one class diagrams. Because activity and class diagrams are the most important (and most prevalent) diagram types, we will focus on these during the remainder of the study. In the next sections, we will address model size, model complexity, development effort from various perspectives, defects and changes and defect discovery over time.

7.5.1 Model Size

Model size metrics have been proposed in many studies (e.g. Marchesi, 1998, Genero et al., 2002, Kim and Boldyreff, 2002). Empirical findings regarding model size metrics have been reported fairly scarcely. Most often, class diagram size metrics are reported (e.g. Marchesi, 1998, Lange and Chaudron, 2005, Lange, 2006, Egyed, 2007, Costagliola et al., 2005, Nugroho and Lange, 2007). To establish the size of a model, we summed all the size elements of all diagrams that were used in a model. Definitions of both

[7] http://www.magicdraw.com/

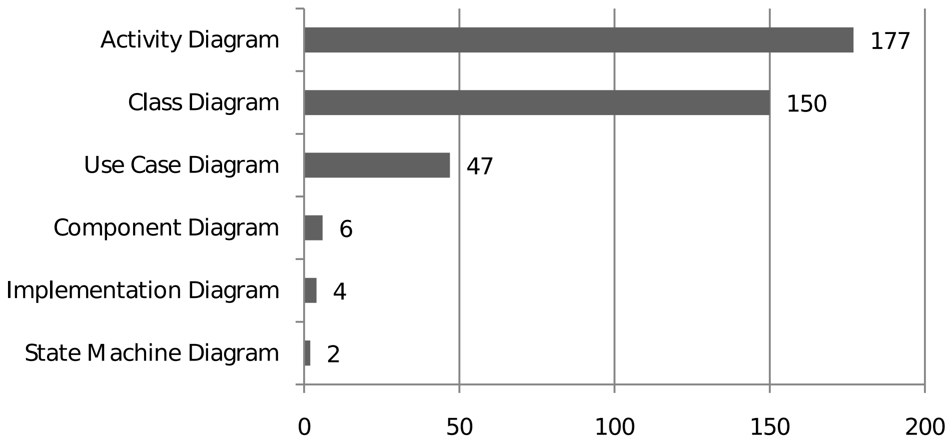


Figure 7.2: *Frequency of UML diagram type use*

activity and class diagram size metrics are presented in Table 7.1. Code size, measured in source lines of code, is an often-used metric to track progress in non-MDD projects. The organization that builds this software, usually estimates and tracks the lines of code as a means of tracking progress. In this case, model size metrics were used. A visualization of size metrics over time is depicted in Figure 7.3(a). As MDD was not formerly employed, the actual progress could not yet be benchmarked to other projects within the same organization for these specific model metrics. The average curve of the growth of metrics over time is plotted in Figure 7.3(b). As is the case with cumulative SLOC visualizations, a fairly gradual sigmoid curve can be discerned. However, the majority of model elements (approximately 72 percent) seems to have been created around development week 20, at approximately a third of the development process. In the same figure, a cumulative plot of revisions over time (as obtained from the repository log) is plotted over the same time period. From this plot, it can be deduced that the amount of revisions per week does not taper off after week 20. In fact, a slight increase can be observed. This implies that most model elements were already in place early in the development process and that model elements are mostly altered. This is consistent with the idea that MDD enables early prototyping and that the majority of development time can then be used for fine-tuning the implementation.

7.5.2 Model Complexity

Model complexity is defined by the sum of the complexity of the activity diagrams and the coupling of the class diagrams as they appear together in a single diagram group. Some complexity diagram metrics for class diagrams, such as *number of methods*,

Table 7.1: *Model Size Metrics*

ACTIVITY DIAGRAMS[1]	
Actions	The number of actions of the activity. Includes actions in all activity groups (partitions, interruptible regions, expansion regions, structured activities including conditional, loop, and sequence nodes), and their subgroups and sub-subgroups.
ObjectNodes	The number of object nodes of the activity. Counts data store, central buffer, and activity parameter nodes in all activity groups and their subgroups.
Pins	The number of pins on nodes of the activity. Counts all input, output, and value pins on all nodes and groups of the activity.
ControlNodes	The number of control nodes of the activity. Control nodes are initial, activity final, flow final, join, fork, decision, and merge nodes. The metric also counts control nodes in all activity groups and their subgroups.
Partitions	The number of activity partitions in the activity.
Groups	The number of activity groups or regions of the activity. Counts interruptible and expansion regions, structured activities, conditional, loop, and sequence nodes, at all levels of nesting.
CLASS DIAGRAMS[1]	
Classes	The number of classes on the diagram.
NumAttr	The number of attributes in the class. Also known as the Number of Variables per class (Lorenz and Kidd, 1994).

[1] source: SDMetrics 2.2 User Manual (Wüst, 2011)

were not applicable to the class diagrams designed by this project due to modeling conventions (no methods were used). Instead, we used coupling measures to denote class diagram complexity. Descriptions of both activity and class diagram complexity and coupling metrics are presented in Table 7.2. Diagram group complexity is defined as the average complexity per diagram type:

$$\text{complexity}_{\text{model}} = \frac{\text{complexity}_{\text{activity diagram}}}{\sum \text{diagrams}_{\text{activty}}} + \frac{\text{coupling}_{\text{class diagram}}}{\sum \text{diagrams}_{\text{class}}} \tag{7.1}$$

Table 7.2: *Model Complexity Metrics*

ACTIVITY DIAGRAMS (COMPLEXITY)[1]	
ControlFlows	The number of control flows of the activity.
ObjectFlows	The number of object flows of the activity.
Guards	The number of guards defined on object and control flows of the activity.
CLASS DIAGRAMS (COUPLING)[1]	
Dep_Out	The number of elements on which this class depends.
Dep_In	The number of elements that depend on this class.
NumAssEl_ssc	The number of associated elements in the same namespace as the class.
NumAssEl_sb	The number of associated elements in the same scope branch as the class.
NumAssEl_nsb	The number of associated elements not in the same scope branch as the class.
EC_Attr	The number of times the class is externally used as attribute type. This is a version of OAEC+AAEC (Briand et al., 1999).
IC_Attr	The number of attributes in the class having another class or interface as their type. This is a version of OAIC+AAIC (Briand et al., 1999) and also known as Data Abstraction Coupling) (Li and Henry, 1993).
EC_Par	The number of times the class is externally used as parameter type. This is a version of OMEC+AMEC (Briand et al., 1999).

[1] source: SDMetrics 2.2 User Manual (Wüst, 2011)

As expected, there exists a positive correlation between diagram group size and average diagram size (Table 7.5.3). This implies that diagram groups that contain more diagrams also contain bigger diagrams. In addition, as diagram group size increases, the average complexity per diagram also increases. This means that certain diagram groups receive more attention than others and might imply that some diagram groups are more important than other models. Not surprisingly, the greater the average diagram size in a diagram group is, the greater the complexity of the diagrams becomes. This underlines our finding that larger diagram groups contain more complex diagrams.

7.5.3 Development Effort

In this section we elaborate on the effort data recorded for the case. The following sections contain an analysis of model development effort, development phase effort and model change effort.

Model Effort

The elaboration phase was executed in three large iterations. The construction phase is executed in many iterations that last one week each. The amount of effort spent per project phase is depicted in Figure 7.4(a). In the construction phase, about 40 percent of effort is spent on development of the models. The remaining effort is spent on the generator and coding. Of the 10,000 hours spent in the construction phase, 500 hours were spent on changes. The amount of effort spent on the models, and the effort types we could distinguish from the data are shown in Figure 7.4(b). As can be seen, a substantial amount of time is spent on adding functionality to the code generator. This effort is disregarded for the analysis of the effort spent on each diagram group. Interesting is that about 9 percent of the time is spent on issue resolution, and 2 percent is spent on changes. This is a relatively low amount of effort. Of all effort, 59 percent could be traced back to a specific diagram. The amount of effort spent on development or modeling does not correlate with model size. Only the effort spent on testing correlates with the amount of defects found. Analyzing the relation between model complexity and effort, we found that, the longer a diagram group is worked on, the more complex the activity diagrams are. Contrastingly, development time does not seem to be related to class diagram complexity.

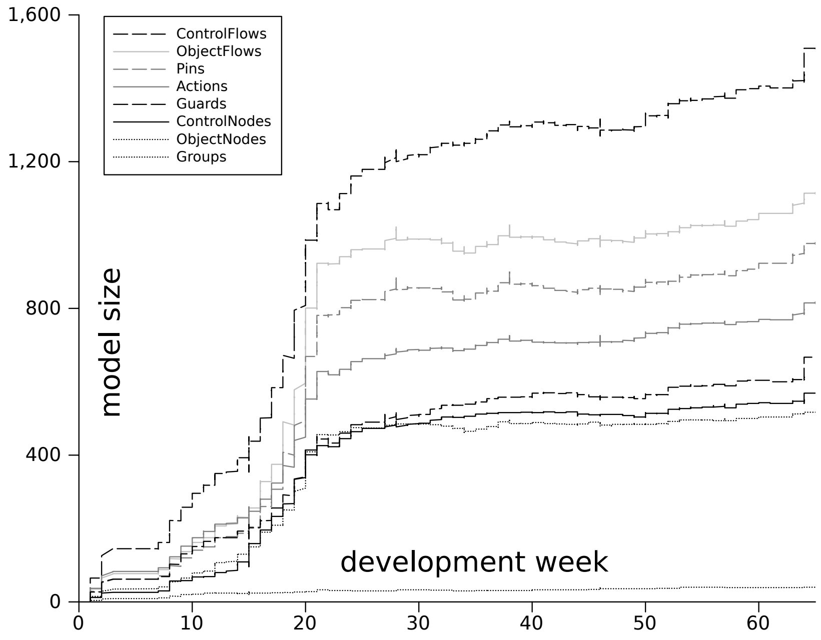
Phase Effort

We compared the effort spent per phase and the length of the phase to the averages of 17 RUP projects that were executed by the same organization (Figure 7.5). Some observations can be made regarding this visualization. First, the inception phase of MDD is quite similar to the other projects. This is most likely because the inception phase of an MDD project is not necessarily different from any other type of project. Second, during the elaboration phase, significantly more effort is spent. This is likely to be caused by a team size increase. Because MDD requires much modeling, more developers are needed at an earlier stage in the project. The team size increase that traditionally takes place at the start of the construction in this MDD project took place in the elaboration phase. Third, the elaboration phase lasted significantly longer. In the interviews, we found three explanations for this phenomenon:

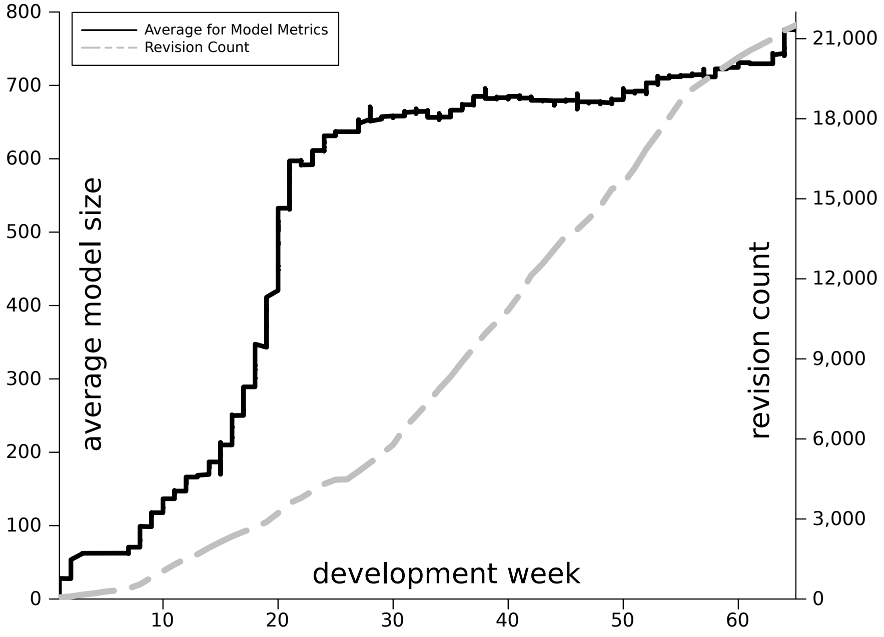
1. the initial design of many of the models is seen as a design activity rather than an implementation activity

2. the switch from the higher level target language to Java, which caused a delay

3. general learning effects of introducing MDD on a large scale

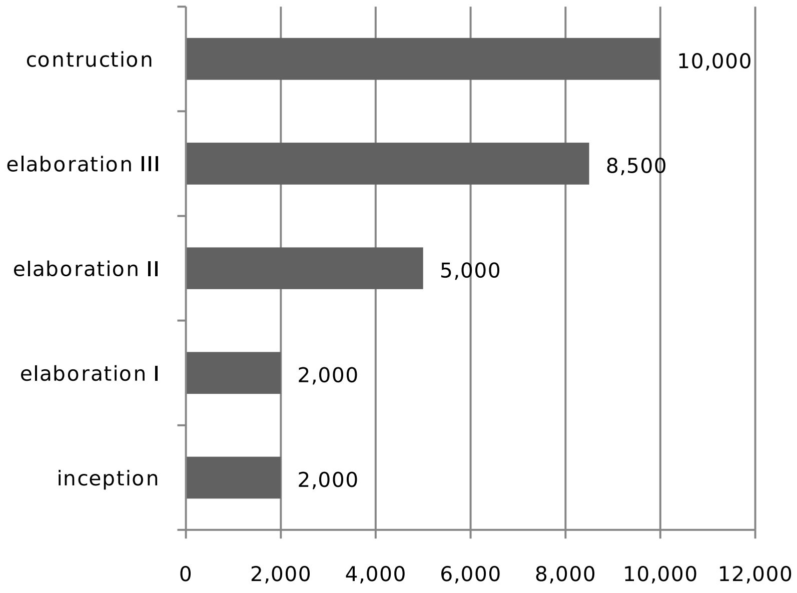


(a) Cumulative model size metrics over time

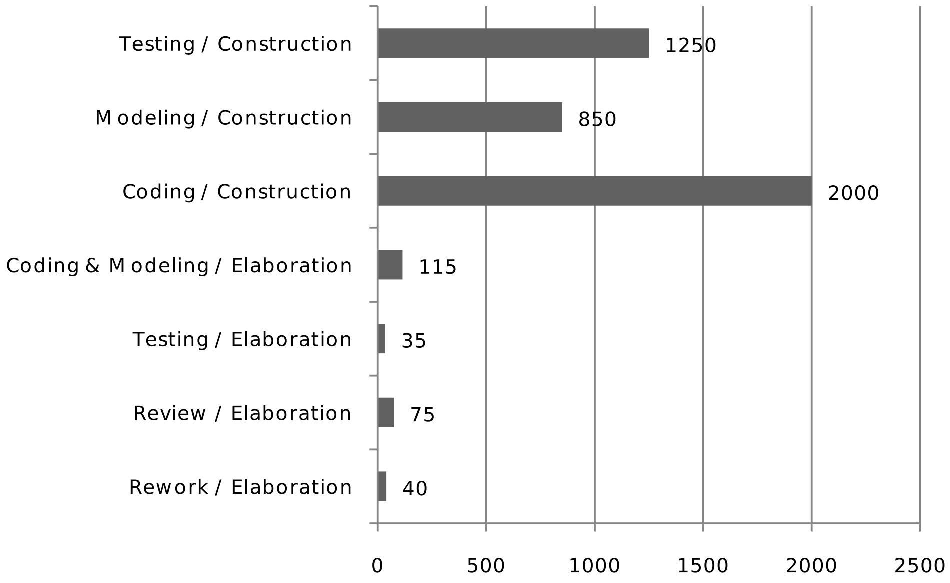


(b) Cumulative average model size metrics over time versus revisions over time

Figure 7.3: *Model metrics and revision count over time*



(a) Hours per development phase



(b) Effort related to model types

Figure 7.4: *Project effort distribution on phase and model level*

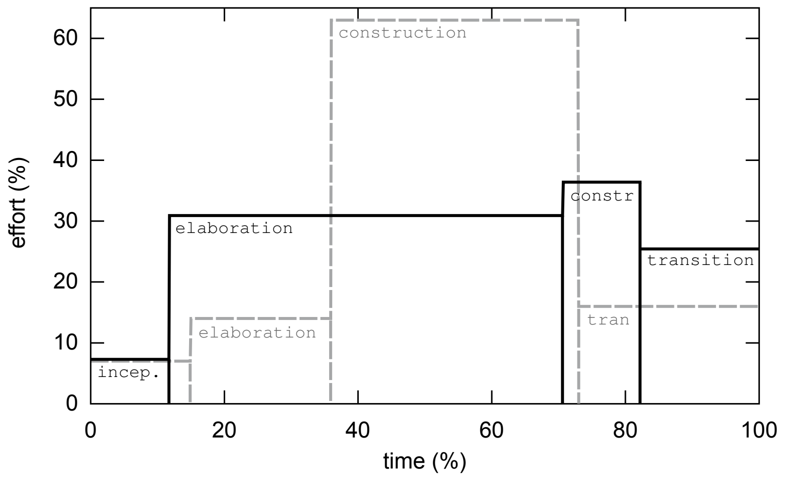


Figure 7.5: *Effort and duration per phase (normalized at 100 percent). The black line represents the case. The dotted gray line in the background represents the average for a set of 17 projects similar in size and complexity*

Table 7.3: *Bi-Variate Correlation Matrix for Common Process Metrics in MDD Context*

		defect count	defect priority	model size	diagram size	model complexity	class diagram size	activity diagram size	defect closing time	activity diagram complexity	number of diagrams	class coupling
defect count	ρ	1	0.176	0.258	0.188	0.103	0.189	0.227	0.294(*)	0.348(**)	0.226	0.093
	p	.	0.191	0.052	0.166	0.452	0.518	0.095	0.026	0.009	0.090	0.753
	N	57	57	57	56	56	14	55	57	55	57	14
defect priority	ρ	0.176	1	0.118	−0.019	0.012	−0.336	−0.020	0.289(*)	−0.032	0.169	−0.440
	p	0.191	.	0.382	0.891	0.928	0.240	0.885	0.029	0.817	0.209	0.115
	N	57	57	57	56	56	14	55	57	55	57	14
model size	ρ	0.258	0.118	1	0.364(**)	0.398(**)	0.428(*)	0.918(**)	−0.075	0.850(**)	0.738(**)	0.613(**)
	p	0.052	0.382	.	0	0	0.023	0	0.577	0	0	0.001
	N	57	57	119	103	103	28	103	57	103	119	28
diagram size	ρ	0.188	−0.019	0.364(**)	1	0.947(**)	−0.175	0.518(**)	−0.021	0.599(**)	−0.451(**)	0.181
	p	0.166	0.891	0	.	0	0.374	0	0.876	0	0	0.356
	N	56	56	103	103	103	28	102	56	102	103	28
model complexity	ρ	0.103	0.012	0.398(**)	0.947(**)	1	−0.190	0.553(**)	−0.064	0.545(**)	−0.456(**)	0.142
	p	0.452	0.928	0	0	.	0.334	0	0.639	0	0	0.470
	N	56	56	103	103	103	28	102	56	102	103	28
class diagram size	ρ	0.189	−0.336	0.428(*)	−0.175	−0.190	1	0.082	0.164	0.158	0.670(**)	0.641(**)
	p	0.518	0.240	0.023	0.374	0.334	.	0.680	0.575	0.421	0	0
	N	14	14	28	28	28	28	28	14	28	28	28

* Correlation is significant at $\alpha = 0.05$ / ** Correlation is significant at $\alpha = 0.01$

(continued on next page…)

SPEARMAN'S RANK CORRELATION COEFFICIENT (ρ) (continued)

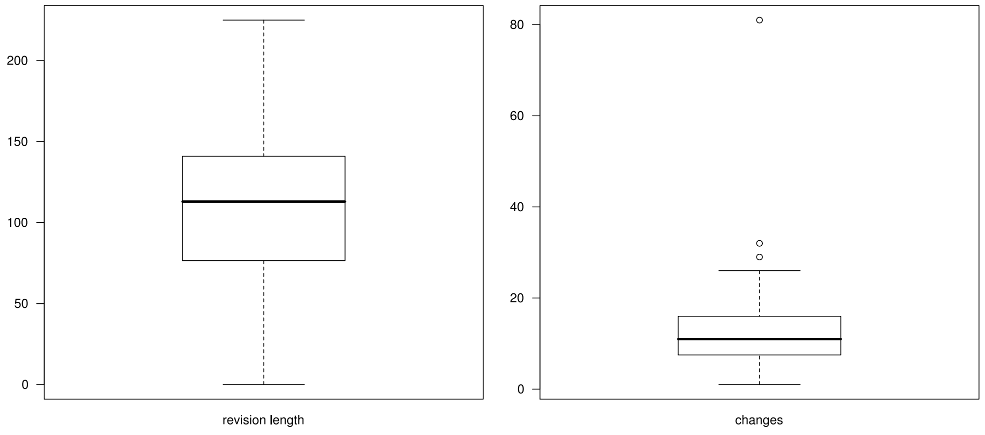
		defect count	defect priority	model size	diagram size	model complexity	class diagram size	activity diagram size	defect closing time	activity diagram complexity	number of diagrams	class coupling
activity diagram size	ρ	0.227	−0.020	0.918(**)	0.518(**)	0.553(**)	0.082	1	−0.117	0.940(**)	0.391(**)	0.332
	p	0.095	0.885	0	0	0	0.680	.	0.395	0	0	0.085
	N	55	55	103	102	102	28	103	55	103	103	28
defect closing time	ρ	0.294(*)	0.289(*)	−0.075	−0.021	−0.064	0.164	−0.117	1	−0.009	0.010	−0.065
	p	0.026	0.029	0.577	0.876	0.639	0.575	0.395	.	0.945	0.939	0.826
	N	57	57	57	56	56	14	55	57	55	57	14
act. diag. complex.	ρ	0.348(**)	−0.032	0.850(**)	0.599(**)	0.545(**)	0.158	0.940(**)	−0.009	1	0.347(**)	0.273
	p	0.009	0.817	0	0	0	0.421	0	0.945	.	0	0.159
	N	55	55	103	102	102	28	103	55	103	103	28
number of diag.s	ρ	0.226	0.169	0.738(**)	-0.451(**)	-0.456(**)	0.670(**)	0.391(**)	0.010	0.347(**)	1	0.405(*)
	p	0.090	0.209	0	0	0	0	0	0.939	0	.	0.032
	N	57	57	119	103	103	28	103	57	103	119	28
class coupling	ρ	0.093	−0.440	0.613(**)	0.181	0.142	0.641(**)	0.332	−0.065	0.273	0.405(*)	1
	p	0.753	0.115	0.001	0.356	0.470	0	0.085	0.826	0.159	0.032	.
	N	14	14	28	28	28	28	28	14	28	28	28

* Correlation is significant at $\alpha = 0.05$ / ** Correlation is significant at $\alpha = 0.01$

Table 7.4: *Bi-Variate Correlation Matrix for Defect Priority and Changes*

Spearman's Rank Correlation Coefficient (ρ)														
		devel. time	changes	number of diagrams	model size	diagram size	model complexity	defect closing time	defect count	defect priority	activity diagram size	activity diagram complexity	class diagram size	class coupling
devel. time	ρ	1	0.769(**)	0.378(**)	0.473(**)	0.215(*)	0.142	0.189	0.653(**)	0.162	0.366(**)	0.440(**)	0.101	0.308
	p	.	0	0	0	0.034	0.164	0.158	0	0.229	0	0	0.648	0.152
	N	107	107	107	107	98	98	57	57	57	97	97	23	23
changes	ρ	0.769(**)	1	0.614(**)	0.634(**)	0.078	0.013	0.138	0.493(**)	0.132	0.451(**)	0.489(**)	0.121	0.544(**)
	p	0	.	0	0	0.445	0.899	0.306	0	0.328	0	0	0.581	0.007
	N	107	107	107	107	98	98	57	57	57	97	97	23	23

* Correlation is significant at $\alpha = 0.05$ / ** Correlation is significant at $\alpha = 0.01$



(a) Revision length (days)   (b) Changes per model

Figure 7.6: *Software configuration and change management system usage*

Change Effort

The amount of changes per diagram were measured by the amount of version updates found in Subversion that were directly related to that diagram. On average, a model had 12.6 versions associated with it. A total of 1,308 change commits to the Subversion repository were associated with a model out of a grand total of 9,035 commits (14.5 percent). The reason for the substantial difference between model-related and non-model-related commits is that the repository contains all documentation regarding the project including status reports and other kinds of management specific files. The files are altered, and subsequently checked-in, frequently. Also, the amount of development time per model was measured as the difference between the dates of the first and the last model related change, measured in days. A summary of the measurements for revision length is depicted in Figure 7.6(a). The average amount of calendar days during which a model was revised was 111. The total amount of days during which all models were altered is 230 days.

7.5.4 Defects and Changes

Defects were stored in a centralized defect tracking system. All team members were able to add defects to the database. Per defect, a unique id, title and description were recorded as well as a priority. Furthermore, defect submission time, closing time and the last modified time were recorded. Lastly, defect status (Assigned, Closed, ...) and defect type were recorded. Six different defect types were used, namely: defects related to deployment, development, generation, modeling, requirements and testing.

A total of 631 defects was registered. Of these defects, 81 percent was directly related to a model. A total of 80 models (or 68.4 percent) had one or more defects associated with them. In this subset, on average, 6.4 defects were found per model. These are pre-release defects. At defect submission time, a defect priority is assigned to the defect report on a scale of 1 (high priority) to 5 (low priority). The mean priority of the defects related to a model is 1.9 whereas the mean priority for a defect that is not directly related to a model is 2.35. This indicates that it is generally seen as more important to solve defects related to a model than to resolve defects that are not related to a model – underlining model centrality. The defects that are not related to a model mainly have to do with the code generator.

We find that the defect count per model positively correlates with defect closing time (Table 7.4). This implies that models with a relatively larger amount of defects, have a higher average defect repair time. This is an intuitive finding as an increase in the number of defects in a single model or diagram can increase the complexity of the repair process and thereby delay a fix.

The finding that models with a relatively larger amount of defects, have a higher average defect repair time is in line with maintenance for source code. A counter-intuitive finding is that while both development time and the amount of changes correlate positively with model size but that model size did not correlate with defect count. This leads us to conclude that larger models are changed more often and worked on longer but do not necessarily contain more defects. However, models that are changed often do contain more defects. The reason for this relation could be that fixing a defect induces extra changes. However, the reverse could also be true, namely that models changed more often contain more defects as a result of an increased amount of changes.

The earlier finding that certain diagram groups receive more attention than others and might imply that some diagram groups are more important than other models is not confirmed by the average defect priority of the diagram because it does not correlate with model size. Bigger models do not contain defects that, on average, are seen as more pressing to resolve. Also, model size does not correlate with defect closing time. We expected a negative correlation between these two variables because larger models are more complex and this could adversely impact the time needed to repair a defect.

Furthermore, bigger models do not contain more defects. While appearing counter-intuitive, this is in line with the Theory of Relative Defect Proneness (Koru et al., 2009) (recently confirmed for closed-source software; Koru et al., 2010).

7.5.5 Defect Discovery

We plotted the cumulative defects found over normalized time for the case (labeled "Project Alpha") and 10 projects that were executed by the same IT organization (Figure 7.7) to enable visual comparison. The lines were smoothed using a Bezier algorithm for readability purposes. In the image, we see a clear difference between the case

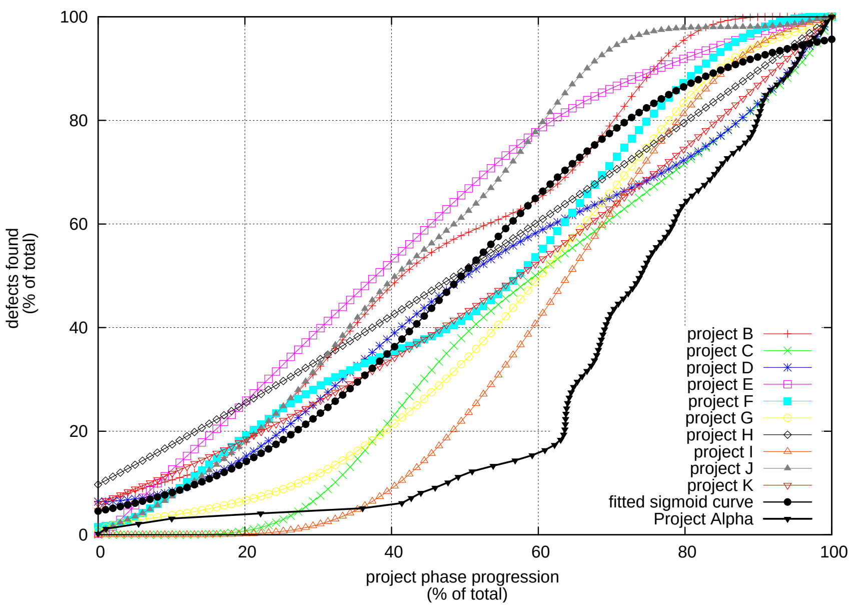


Figure 7.7: *Cumulative defect discovery over time (case is labeled "Project Alpha")*

(Project Alpha in the graph) and the other projects. For example, when 40 percent of the project time has passed only around 10 percent of the defects are reported whereas, on average for conventional projects, 35 percent of defects were found. We found three reasons for this relative slow defect discovery rate. First, ineffectivity in finding defects due to a learning curve that interviewees associate with introducing MDD. Second, because MDD was applied, much effort was spent in creating the models and the generator at first, only later, when the hand coded part of the system was developed, were real defects reported. Initial defects mostly involved the code generator. Third, the standard quality assurance process was not yet tailored for MDD. Finding defects in later stages in a project is commonly regarded as undesirable.

However, the amount of defects found per function point is equal to or less than[8] 0.32. The amount of function points has increased since the initial function point analysis that was executed before the project started development, due to change requests. The average of defects found per function point for 22 similar sized, non-MDD development projects that were executed at the same organization is 0.52. This implies that for projects on average approximately 396 less defects are reported for MDD. This is a drastic decrease in the amount of defects found. A possible explanation

[8]We use the initial functional point count for this calculation. The amount of function points that is implemented is expected to be higher than the initial functional point count.

is the (much) larger proportion of the development effort that is spent on model improvement compared to model build-up.

7.6 Conclusions and Future Work

The main objective for this study was to report on the specific characteristics of a large scale, industrial MDD project and to asses what the impact was of using MDD tools and techniques compared to non-MDD development.

Adopting MDD tools and techniques fundamentally impacts the software development process in general and the analysis and design phases in particular. Because almost all code was directly generated from diagrams, models were first-class citizens (France and Rumpe, 2007, Balasubramanian et al., 2006) in the software development process followed in this case.

Three striking differences in the development process were found. First, 59 percent of all effort was spent on developing the model. That is significantly more than the time spent on code development in classical software development. Second, most model elements were already implemented at one third of the development process. The remaining development time was spent on altering the models. Third, 40 percent fewer defects were found when compared to projects of similar size.

Diagrams and code are fundamentally different and therefore not easily compared as we found absent, for example, a positive relation between model size and complexity on the one hand and defects on the other — relations that have often been observed in source code. Also, larger diagrams were changed more often and worked on longer but did not necessarily contain more defects.