Abstract

Model driven development (MDD) is a software engineering practice that is gaining in popularity. We aim to investigate to what extent it is effective. There is a lack of empirical data to verify the pay-offs of employing MDD tools and techniques. In order to increase the knowledge we have of the impact of MDD in large scale industrial projects, we investigate the project characteristics of a large software development project in which MDD is used in a pure form. This study focuses on analyzing model size and complexity and metrics related to model quality and effort. Furthermore, project team members were asked to elaborate on their views on the impact of using MDD. Our findings include that larger models are more complex, contain more diagrams, 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. Benefits mentioned by team members were an increase in productivity, benefits from a consistent implementation and their perception of improvement of overall quality. Also, a reduction in complexity was attributed to the use of MDD techniques. We could confirm the perceived increase in the quality of the product in that the average amount of defects found is significantly lower than in similar size projects in which MDD was not employed.

1. Introduction

Model-driven development (MDD) encompasses the use of models and model technologies to increase the level of abstraction of the software development process. As a result, the development process would be less complex yet formal enough to be automated, thereby positively influencing various tasks in the software development life-cycle such as software maintenance. MDD furthermore promises, short and long term productivity gains, improved project communication and software quality improvements, implying defect and rework reduction. Critics argue that software complexity can only be moved, not removed or destroyed and that little evidence exists that the MDD approach indeed provides the claimed benefits.

In their survey of the state of the art in applying MDD in industry, Mohagheghi and Dehlen [1] identify threats to validity with regard to their aggregation of MDD literature. Their concerns include the lack of literature that studies MDD applied in larger industrial projects, the lack of baseline data in organizations, the lack of quantitative data, improper definition of metrics and the likelihood of the publication of successful over less successful applications.

In this study, we set out to investigate the industrial use of MDD in a pure form in general and the impact of the size and complexity of the models on defect count and development effort in particular. To this end, we study a large scale, industrial MDD project of which certain tasks are offshored.

The outline of the study is as follows: Section two of this paper describes the objectives of the study. Section three describes related work, section four elaborates on the case and section five outlines the method of the study. In section six, the collected data is described and section seven contains a presentation and discussion of the findings of the study and elaborates on the impact of MDD practices on the case. Lastly, section eight concludes the study.

2. Objective

It is important to investigate industrial cases of MDD for benchmarking purposes and to evaluate the impact of the process and techniques used. To these ends, this study investigates two main questions:

1) What does a large scale, industrial model driven development project look like?
2) What is the impact of using model driven development tools and techniques?

The first question aims to add to the scarce scientific literature regarding large scale industrial applications of
MDD. This question is divided into questions regarding models and diagrams such as: What types of diagrams are used? and: Of what types of diagrams are models made up? Other questions regard model size and quality metrics: How big and how complex are these models? and: Does model size and complexity impact defect count? Yet another group of questions regard the process of MDD: How is effort distributed in a pure MDD project? and What is the amount of changes per model?

The second main question investigates the effects of using MDD tools and techniques. One of the main motives for implementing MDD is productivity improvements through a reduction in the complexity of the development process. Although a certain amount of complexity is inherent to a development process, we believe that some complexity is imposed and can be removed. We therefore ask: Do team members experience the reduction in complexity that MDD promises?

3. Related Work

Literature regarding MDD in large scale, industrial projects often describe processes in which legacy systems are reverse engineered to MDA (eg. [2], [3], [4]). These works do not focus on the engineering of new software but on the challenges of translating existing software to models. Other work that does describe the implementation om MDD techniques in software engineering processes but does not report metrics such as model size, model complexity or ratios such as defects per model size (eg. [5], [6], [7]). For aggregate studies regarding the impact of MDD, such measures are needed. Only few studies offer enough data to quantify and baseline productivity and quality in industrial MDD projects (eg. [8], [9]). Most empirical studies regarding MDD address questions regarding efficiency (eg. [10], [11]). Hailpern and Tarr [12] 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.

4. Case Description

We examine a project in which a system is defined, designed and built for supporting sales of mortgages. The client is a large financial institution that operates globally. The contractor is a Dutch IT service provider. In the development process, formal modeling guidelines are used. These guidelines address the dynamic aspects of the system and are based on the Unified Modeling Language (UML) 2.0. 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 a combination of open source libraries. During the project, developers work at extending and enhancing the code generator.

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 semantics contain 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 screens’, are described in UML and completely generated to source code. For complexer, custom screens, the semantics do not suffice. These 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 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 and Hibernate are used for target development and the tools used are Eclipse, JBoss and MagicDraw. The final application is to operate on the IBM WebSphere platform and will use a DB2 database. Approximately 70% of the code is generated, the remaining 30% is written ‘by hand’.

The Rational Unified Process (RUP) is 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 [14]. Iterations last a single week each. At the time of writing the project is at 90% of total execution time and is planned to take about 17 months. The project is carried out distributedly. A team of developers and testers work in India. Modeling is done in the Netherlands and development is done at both locations and testing is done in India. The Dutch project leader is also the main communication line to the customer in the Netherlands. The contractor has much experience in managing offshore software development processes. In addition, some Indian colleagues are working in the Dutch team.

There are 32 team members working on the project of
which only a few do not work on this particular project full-time. This roughly corresponds to 28 full-time equivalents (FTE). A function point analysis that was based on the requirements and that was executed in the early stages of the project reported a total of 1,981 function points. During the execution of the project, various change requests have been made. The initial function point analysis is not yet corrected for these change requests. The actual amount of function points is therefore expected to be higher.

5. Method

Data was collected from various sources. The models were collected from a Subversion repository. Effort and defect data was collected from SourceForge Enterprise Edition. Data regarding the development process as well as opinions regarding the process were gathered by means of semi-structured interviews with team members of a department involved with project measurement. We interviewed one developer, one lead developer, two project leaders and one estimation and measurement officer.

For this study, we define a model as a set of diagrams and a system as a set of models. After collecting all models from Subversion, metrics were extracted using SDMetrics [?]. 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-model basis. Therefore, the resulting metric files were aggregated per model so that effort and defect data per model could be combined. In this project each model consists of a separate file.

For statistical analysis and data visualization, R 2.8.1, SPSS 15 and Excel 12 were used.

6. Results

The following subsections report on the measurement results for the models, model size, model complexity, effort spent and the defects and change requests.

6.1. Models

UML Models are created using MagicDraw.14.51. We define a model as a set of diagrams. A total of 119 models contain a total of 327 activity and class diagrams. A bar chart of the UML diagram types (Fig. 1) 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. Diagrams such as sequence diagrams are infrequently used. 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 models contain one or more activity diagrams, 32 models contain 150 class diagrams and all use case diagrams are spread over just two models. 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 prevalent diagram types, we will focus on these during the remainder of the study.

6.2. Model Size

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 activity and class diagram size metrics are presented in Tab. 1. Descriptives with regard to model size are presented in Fig. 2(a) and 2(b).

6.3. Model Complexity

Model complexity is defined by the sum of the complexity of the activity diagrams and the coupling of the class diagrams. Some complexity diagram metrics for class diagrams, such as 'number of methods', were not applicable to the class diagrams designed by this project due to modeling conventions used. Instead, we used coupling measures, which we regard as a specific type of complexity metrics, to denote class diagram complexity. Descriptions of both activity and class diagram complexity and coupling metrics are presented in Tab. 2. Boxplots of complexity measures are depicted in Fig. 2(c) and 2(d).

For our analysis, we analyze models and we therefore must define a complexity metric per model. To this end, we need to calculate an aggregate metric. Model complexity (comp) is defined as the average complexity per diagram type. This is presented in Eq. 1.

Figure 2. Diagram Size and Complexity Measures

Table 1. Model Size Metrics

<table>
<thead>
<tr>
<th>ACTIVITY DIAGRAMS</th>
<th>CLASS DIAGRAMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actions</td>
<td>The number of classes on the diagram.</td>
</tr>
<tr>
<td>ObjectNodes</td>
<td>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.</td>
</tr>
<tr>
<td>Pins</td>
<td>The number of object nodes of the activity. Counts data store, central buffer, and activity parameter nodes in all activity groups and their subgroups.</td>
</tr>
<tr>
<td>ControlNodes</td>
<td>The number of object flows of the activity. Counts all input, output, and value pins on all nodes and groups of the activity.</td>
</tr>
<tr>
<td>Partitions</td>
<td>The number of guards defined on object and control flows of the activity.</td>
</tr>
<tr>
<td>Groups</td>
<td>The number of control flows of the activity.</td>
</tr>
<tr>
<td></td>
<td>The number of associated elements in the same namespace as the class.</td>
</tr>
<tr>
<td></td>
<td>The number of associated elements that depend on this class.</td>
</tr>
<tr>
<td></td>
<td>The number of associated elements in the same scope branch as the class.</td>
</tr>
<tr>
<td></td>
<td>The number of associated elements not in the same scope branch as the class.</td>
</tr>
<tr>
<td></td>
<td>The number of times the class is externally used as attribute type. This is a version of OAEC+AAEC [16].</td>
</tr>
<tr>
<td></td>
<td>The number of times the class is externally used as parameter type. This is a version of OMIE+AMIE [16].</td>
</tr>
</tbody>
</table>

1. source: SDMetrics 2.1 User Manual

\[
comp_{\text{model}} = \frac{comp_{\text{actdiagram}}}{\sum \text{diagrams}_{\text{act}}} + \frac{coupl_{\text{cladiagram}}}{\sum \text{diagrams}_{\text{cla}}} \quad (1)
\]

6.4. Effort

In this section, we elaborate on the effort data, recorded for our case. Project effort is divided into activities and categories. Fig. 4(a) and 4(b) display how effort was distributed. The total size of both pie charts is equal. 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 model. Interesting is that about 9% of the time is spent on issue resolution, and 2% is spent on changes. A seemingly low amount of effort.

We could not benchmark this metric with data from other projects executed by the same organization as this data is not available. We could trace back 59% of total effort to a specific model.

The amount of changes per model were measured by the amount of version updates found in Subversion that were directly related to a model. On average, a model has 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. The reason for this large difference is that the repository contains all documentation regarding the project including status reports and other kinds of management specific files which are altered 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 Fig. 3(a). The average amount of calender days during which a model was revised is 111. The total amount
of days during which all models were altered is 230 days.

6.5. Defects and Changes

In this section, we examine the defects and changes that were tracked for our case. Six different defect types were used, namely: Defects related to deployment, development, generation, modeling, requirements and testing. A total of 631 defects are registered. Of these defects, 81% is directly related to a model. A total of 80 models (68.4%) has one or more defects associated with them. In this subset, on average, 6.4 defects are found per model. 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. The defects that are not related to a model mainly have to do with the code generator. A total of 96 change requests were registered. These change requests could not be traced back to specific models and are therefore not regarded for further analysis.

7. Discussion

The results of a bi-variate correlation analysis of 11 of the variables in the data set are presented in Tab. 3. The correlation between model changes, the development time per model and the other case variables are presented in Tab. 4. The following subsections will elaborate on the research questions and the various correlations found between the variables.

7.1. Model Size and Complexity

As expected, there exists a positive correlation between model size and average diagram size. This implies that models that contain more diagrams also contain bigger diagrams. In addition, as model size increases, the average complexity per diagram also increases. This means that certain models receive more attention than others and might imply that some models are more important than other models. This assertion is, however, not confirmed by the average defect priority of the models, which 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. This is also surprising as there is a known relation between module size and defect count at the source code level.

Not surprisingly, the greater the average diagram size in a model is, the greater the complexity of the diagrams becomes. This underlines our finding that larger models contain complexer diagrams. For the two separate diagram types we also find that class diagram size positively correlates with class complexity and that activity diagram size strongly
correlates with activity diagram complexity. Another interesting finding regarding diagram size is that when the average diagram size increases, the amount of diagrams in a model decreases. This leads to the contradiction that model size positively correlates to diagram size and the number of diagrams in a model but that diagram size correlates negatively to the number of diagrams found in a model. As we only regard class and activity diagram size, the explanation might be found in analyzing the size of the other diagrams.

7.2. Defects

In this subsection we discuss whether larger models contain more defects. The defect count per model positively correlates with the defect closing time. 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. This finding is further underlined by the strong positive correlation between defect count per model and the total activity diagram complexity per model. Models which contain complex activity diagrams, are relatively more prone to defects than models which contain less complex activity diagrams.

7.3. Effort

In this subsection we report on the relation between effort and model size. 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. The other effort categories and activities do not correlate with any of the other variables.

The positive correlation found between development time and changes is expected, as is the correlation between development time and model and diagram size. Larger models are worked on longer and are changed more frequently. Both the development time and the amount of changes correlate positively with model size. This is interesting as 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.

Analyzing the relation between model complexity and effort, we found that, the longer a model is worked on, the more complex the activity diagrams get. Contrarily, development time does not seem to be related to class complexity.
diagram complexity. Also, models that are changed more often seem to contain more complex class and activity diagrams.

7.4. Impact of MDD

What is the impact of using MDD techniques? Anda and Hansen [2] found that MDD has a positive impact on team communication. The project team members that were interviewed acknowledged that technical discussions relating to aspects of the systems are easier to conduct than they were used to in non-MDD projects because they could use models as a base for discussion instead of source code. Other benefits mentioned were an increase in productivity, benefits from a consistent implementation and their perception of improvement of overall quality. The amount of defects found per function point is equal to or less than 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 non-MDD development projects that were executed at the same organization is 0.52 ($\bar{n} = 0.46$). This implies that for similar size projects on average roughly 396 more defects are reported. This is a dramatic decrease in the amount of defects found.

Baker et al. [7] report an effort reduction of a factor 2.3, a 33% reduction in the time needed to develop test cases and a significant drop in the amount of time needed to correctly fix a defect. Team members acknowledge that they believe that defects are fixed faster. We could, however, not find data to support this claim. And although team members reported that inspection was easier because of the use of models, we could again not quantify this claim. Weigert et al. [9] report a three to 10-fold increase of the inspection time per hour, a five fold increase in source lines produced per person month during development.

In a student experiment Krogmann [18] found an 11% increase in productivity. For our project, the overall productivity could not yet be established because the project is not finished. After a final function point analysis the project can be benchmarked against the organization’s non-MDD projects.

7.5. Threats to validity

Because developers could not reach always each other immediately due to the distributed nature of the development process, more defects were reported. In cases where a developer could not reach someone to communicate a problem directly, he or she might have been more likely to report a defect. Furthermore, the extend to which the results can be generalized are limited to projects in which similar MDD processes, techniques and tools are used. Other context specific factors such as the size of the project, team composition and the fact that offshore collaboration techniques were used, could have had an influence on effort distribution, model size and the impact of MDD in general.

8. Conclusion

The main motivations for this study were to investigate (1) What do large scale, industrial MDD projects look like? and (2) What is the impact of using MDD tools and techniques? To this end, we presented an overview of what types of diagrams are used and of what types of diagrams are models made up. Activity diagrams were most abundant in this project, followed by class diagrams. Model size and complexity do not correlate with defect count. Model size and average diagram size are positively correlated. Also, model size is correlated with the average complexity per diagram but not to development effort. Furthermore, we found that the defect count per model positively correlates with defect closing time. 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.

Regarding the centrality of the models in the development process we can state that 59% of all effort was spent on developing models. This is a dramatic increase compared to non-MDD projects. In interviews with team members, benefits mentioned with regard to the MDD approach were an increase in productivity, benefits from a consistent implementation and their perception of improvement of overall quality. Also the use of MDD techniques was credited as reducing complexity. We could confirm the perceived
increase in the quality of the product in that the average amount of defects found is significantly lower than in similar size projects in which MDD was not employed.

The objective of the study was to add to the body of knowledge that exists regarding large scale industrial MDD projects. This study contains an analysis of only a single case. The results are therefore not generalizable to other projects. However, they do add to the scarce amount of empirical data that currently exists regarding software engineering projects of this type.

References


