An investigation on the impact of MDE on software reuse

Daniel Lucrédio
DC - UFSCar
Computing Department
Federal University of São Carlos
http://www.dc.ufscar.br/~daniel

Eduardo Santana de Almeida
DCC - UFBA
Department of Computer Science
Federal University of Bahia
http://homes.dcc.ufba.br/~esa/

Renata P. M. Fortes
ICMC - USP
Institute of Mathematical and Computer Sciences - University of São Paulo
http://www.icmc.usp.br/~renata/

***Abstract*—Software reuse and model-driven engineering (MDE) are two distinct approaches that attempt to improve quality and productivity in software projects. Much is said about how MDE can increase software reuse by reducing the amount of hand-written code, but few studies consider the fact that in MDE other artifacts – models, tools, transformations and code generators – come into play and need to be considered. How much more reuse can we achieve with MDE? How reusable are these MDE-specific assets? Motivated by these questions, this paper presents the observations made in three exploratory studies. In each study, the same software was developed with and without MDE, and a comparison based on reuse metrics was made. The results indicate that MDE can increase and/or improve software reuse in different ways, but with some associated costs, such as increased software complexity and decreased maintainability. In the context of our observations, complex technical domains have more to gain from the automation achieved with MDE, while business domains seem to be more suitable for simpler feature-based configuration. We conclude the paper pointing out more studies that could be performed to gain additional knowledge regarding MDE and reuse.**

I. Introduction

Software reuse can bring benefits in terms of quality and productivity. But code reuse is not enough, and only through higher level reuse we can start to see real benefits [1]. In model-driven engineering (MDE), reuse is achieved through generative programming, domain-specific languages (DSL) and software transformations, which not only save considerable effort in repetitive tasks, but can reduce the semantic gap between the problem and the implementation/solution [2], thus effectively increasing the abstraction level where reuse takes place.

Although MDE and software reuse seem to be a natural match, there is little evidence on the application of MDE and the observed benefits [3]. There are many literature reports describing gains in terms of productivity due to code generation, but only a few discuss the impact caused by the introduction of MDE-specific assets into the process. For example, modeling for MDE may be a complex task, sometimes as complex as writing code [3], and there are specific difficulties in coordinating distributed teams in a MDE project [4]. These and other factors may impact negatively on the process, causing the overall gains with MDE to be smaller than it looks like. Sometimes, these negative factors may cause a productivity loss [3], even if we discard the large upfront cost of developing the complete infrastructure for applying MDE.

Based on this scenario, in this paper we attempt to investigate such issues from a reuse perspective. We explore how MDE can influence software reuse, considering that MDE increases the abstraction level through modeling and code generation, but also introduces new assets – models, languages, transformations and code generators – that must be taken into account. We present the observations made in three exploratory studies [5], from both academia and industry, aiming to provide quantitative and qualitative data regarding the impact of MDE on software reuse. We believe our results can strengthen current evidence beyond intuition and subjective opinions, offering better support to future research.

II. Related work

Mohagheghi and Dehlen [3] review several reports of MDE application in industry. Among their goals, the authors wanted to search for evidence on the impact of MDE on productivity and software quality. After investigating 25 papers from 2000 to mid-2007, they concluded that only a few had quantitative data (two from practical application of MDE in Motorola [6], [7] and two comparative studies [8], one of them [9] from the EU-supported MODELWARE project, which is no longer active). Therefore, only four had quantitative data. Most of these report productivity gains when MDE is used, however there are some contradictory results. For example, one study reports a productivity gain of 24% on average. Another reported no difference in productivity. Another presented a productivity loss of 10%. And another had a productivity gain of 69% during coding. These gains are accredited mainly to code generation, simulation, test generation, avoiding defects, domain-specific languages and reuse. The productivity loss is accredited to immature tools, high start up costs, and the fact that modeling can be at least as complex as programming, in some cases.

Krogmann and Becker [10] present a comparison of two software developments, one using a code-centric approach, and the other using MDE. Despite the fact that the study has many uncontrolled factors, the authors observed that with MDE the same software could be developed with only 11% of the total effort with the conventional approach. These

are rough estimates, as they did not log development time precisely. Of course, these gains are only obtained after a large upfront investment to build the MDE support, which was not measured.

Kulkarni et al. [4] discuss their 15-year experience with MDE in an industrial scenario. They present their approach for MDE, focusing on issues that are critical if MDE is to scale up properly, such as the use of a repository and configuration/evolution management. Regarding the observed impact of MDE, they argue that productivity has doubled in some specific phases of development, mainly design and construction. However, significant effort has to be spent for coordination between distributed teams, eating some of the productivity gains. An overall effort savings of around 20% was observed in one particular project.

Lussenburg et al. [11] present a comparison of a non-MDE implementation with an implementation generated by a MDE tool, which is called Mod4J. Three criteria were studied: conformance to the reference architecture, functional requirement satisfaction and reduction of handwritten code. For this last criterion, the study observed that a reduction of 75% was achieved with Mod4J. Although these are not statistically validated numbers, it is hard data from a real case.

Zhang and Patel [12] employ agile techniques together with MDE to deliver software faster and with higher quality. They report the results of one project, where they observed that 93% of the entire code was automatically generated, leading to a threefold increase of productivity in terms of code lines per staff month, compared with hand coding. They also observed that the quality of the automatically generated code in terms of defects density is significantly higher. However, they do not present the actual data or any baseline.

In comparison with the related work, we believe that our study provides insightful quantitative data, which is still scarce in the literature [3]. Another point is that most studies measure software quality and productivity. This is also our goal, however we adopt a software reuse perspective. We are unaware of a study with a specific quantitative focus on the impact of MDE on software reuse. Also, most papers and reports focus on quantifying generated code and overall productivity gains. We try to provide a more detailed investigation of how reuse is being affected. In this sense, our study takes a more exploratory path, trying to offer a richer picture that may serve as basis for future research.

Regarding the results, as in most studies, we also observed high productivity gains, specially in terms of coding. One particular observation from a study reviewed by Mohagheghi and Dehlen [3], regarding modeling complexity, also coincides with our findings, i.e., for simpler domains, MDE can increase overall complexity and end up reducing productivity.

III. Studies Definition

All studies investigated MDE-based domain engineering. Domain engineering is the process of building reusable assets in a domain, so that applications can be more easily built reusing those parts. Here, the domain assets include not only reusable code, but also MDE-based code generation templates and transformations.

The definition of the studies followed the Goal-Question Metric paradigm [13]:

Goal G_1. ***To analyze*** *MDE* ***with the objective*** *of determining if it increases and/or improves software reuse, when compared to non-model-driven development,* ***with respect to*** *the produced domain assets* ***from the point of view*** *of the researcher* ***in the context*** *of domain engineering projects.*

- Q_1. By analyzing the same project developed with and without MDE, is there an increase/improvement in software reuse for the project that used MDE?
- Q_2. Are the software assets produced with MDE more reusable than those produced with a non-model-driven approach?

Question Q_1 tries to observe if there was reuse occurring between the projects, by measuring how much software is being reused, while Q_2 tries to observe the quality of the software assets produced with MDE, to see if they are more reusable than those produced without MDE.

Goal G_2. ***To analyze*** *MDE* ***with the objective of*** *determining its importance in the software life cycle* ***with respect to*** *the benefits and difficulties of using it* ***from the point of view*** *of the domain engineer* ***in the context*** *of domain engineering projects.*

- Q_3. Do the developers that use MDE perceive benefits in the development?
- Q_4. Do the developers that use MDE perceive difficulties in the development?

The MDE approach used in this research combines domain engineering with MDE to leverage software reuse. Its two main characteristics are: awareness of multiple subdomains, each one with a different potential for model-driven automation, and the combination of top-down and bottom-up development to better deal with complex domains. The approach is iterative and centered on the uncertainty regarding the ability to automate the subdomains. Initial iterations start with little knowledge about each subdomain, and a top-down process is followed to identify domain concepts and variability information. As further developments take place, more knowledge is acquired, and then a bottom-up process is followed, using design by example to derive higher level assets from a reference implementation. More details can be seen in previous works [14], [15], [16], although these are not essential for understanding the results presented here.

Regarding goal G_1, the comparison was made by developing the same software with the MDE approach and without it, i.e., in an ad hoc way. As described earlier, goal G_1 is to investigate only the **produced domain assets**. Thus, for the software developed **without MDE**, we analyzed the produced code components, such as classes, JSP, PHP and other code assets. In non-MDE (code-centric) development, these are the assets the developers must primarily build and maintain. For the software developed **with MDE**, we analyzed the produced code components for a single exemplar application in the

involved domain with the same requirements as the software developed without MDE. But we also included the models, languages, transformations and the code generators, because in MDE the development and maintenance of these assets is also part of the process.

There is a question to this setup that needs clarification, regarding the distinction between development *for reuse* and development *with reuse*. For the development without MDE, only a single application was developed. But for the development with MDE, a complete reusable infrastructure was developed, so we are including in the comparison both the assets developed *for reuse* and *with reuse*. We are discarding in our analysis the upfront costs for building this infrastructure, for we want to compare the reusability of the produced software. In theory, after the infrastructure is built, the developer only has to worry about specifying models and generating code (development with reuse), and so only the input models and hand-written custom code should be considered. However, in our experience, we agree with Tolvanen [17], who states that MDE and code generation should be used in a single company, for a very specific domain. In this scenario, we observed that maintenance in the languages, transformations and code generators is also necessary, even if less frequent, during development *with reuse* as well. So, for the development with MDE we decided to include these assets in the analysis.

Based on these considerations, for this evaluation four different developments (D) were made for each study:

- D_1. The domain was developed without the approach (in an ad-hoc way), producing reusable code assets and components;
- D_2. An application was developed reusing the product of D_1;
- D_3. The domain was developed with the approach, producing reusable code assets and components, as in D_1, but also MDE-specific assets, i.e., domain-specific languages, modelers, transformations and code generators; and
- D_4. An application was developed reusing the product of D_3.

In all studies, the implementation obtained without the approach (D_1) was used during the reference implementation development activity of the approach (D_3). Hence, the final products with and without the approach were very similar. The final products obtained from developments D_2 and D_4 were compared using some metrics, described in the next section.

The idea of the comparison is the following: with MDE, less code has to be handwritten because more code is reused due to code generation. But this does not mean the developer has less work to do. He must deal with new, different assets. In theory, these are higher-level and better to work with, improving overall productivity and quality. But as discussed in Section II, there is evidence suggesting that this is not always true. For example, in some cases, modeling may be as complex as writing code by hand, and therefore no benefit is perceived. We wanted to analyze this question in more details.

For goal G_2 we compared these four developments, with (D_1 and D_2) and without (D_3 and D_4) the MDE approach, from a more qualitative point of view, to complement our findings with feedback from the developers.

A. Metrics

Following the GQM paradigm, metrics were defined to help answering the questions. Literature describes some metrics to assess software reuse, but these are not capable of offering a complete picture of how reuse occurs in a system [18]. Thus, a careful analysis is necessary, combining different metrics that may lead to indications or evidence that are relevant to the problem at hand. With this in mind, and following our experience in a previous study [19], the following metrics were defined for Q_1:

M_1**.** *RP - Reuse percent*. Reused LOC / Total LOC. This is a classic reuse metric [20], which counts as reused any line of code that is reused without modification. Special care must be taken with this metric, because a large number of reused LOC does not necessarily means good reuse. There could be many lines of code being copied to a new context, but not being actually used.

M_2**.** *RR - Reuse ratio*. The same as M_1, but here artifacts that suffer only minor modification (less than 25% of its code) are considered reused [21].

M_3**.** *URP - Undesired reuse percent.* As discussed in M_1, the reuse of large unused pieces of code may distort the reuse percent metric, not to mention the extra work needed to maintain the unnecessary lines of code. For this reason, we defined metric M_3, which consists of calculating the lines of code that are reused (copied to the new context) but are not being used. Most IDEs and programming environments have functions that determine which methods are not being called, and can be used to collect this information. In our studies, we used Eclipse.

M_4**.** *GRP - Generated reuse percent*. We defined this metric to calculate how many LOC of generated software are being reused over the total number of reused LOC. It allows us to characterize reuse in terms of code generation.

M_5**.** *SCR - Specification/code ratio*. We defined this metric as a complement to M_4, to determine the relationship between one specification element and the corresponding generated code. For example, if the specification of a class, with attributes and relationships, produces 1000 lines of code, more reuse is happening, when compared to the specification of a 10-line configuration file that produces 20 lines of code. This metric is calculated by dividing the number of specification elements in a model by the sum of generated LOC for that model. A specification element may be a line of text (for a textual DSL) or a graphical element in a diagram (for a visual DSL): a box, an attribute, a line, or another graphical element.

Metrics M_4 and M_5 assume that code generation takes place, and therefore cannot be used to compare a project developed without the approach with a project developed with the approach. However, they are useful to help characterizing how reuse is occurring with the approach.

Metrics M_1, M_2 and M_3, related to question Q_1, can lead to problems because they do not consider the nature of the reusable software assets, nor the way they are being reused, penalizing, for example, large monolithic systems [18], [22], [19]. For this reason, an alternative is to try to measure

reuse through quality attributes related to the reusability of a software asset. In this sense, indirect measures like maintainability and complexity are suggested [23], [19]. Therefore, the following metrics were defined for question Q_2:

M_6. *Module instability*. A piece of software is rigid if it can not be easily modified, i.e., a single change starts a cascade of changes in different modules, thus making it difficult to reuse. The module instability metric (I) [24] aims at assessing this characteristic, by measuring the coupling between one module with others. The instability metric ranges from zero - a stable, easy to modify module - to one - an unstable, difficult to modify module. This metric can be used both in source code and MDE assets, like DSLs, models, transformations and generators, as it is based on the generic concept of dependencies. However, while there are automated tools that extract the needed values directly from the source code, for other assets data collection is manual.

M_7. *Cyclomatic complexity*. This is a classic metric [25] with a fundamental relation to software reuse, because more complex artifacts are more difficult to reuse [22], [23]. For being applicable to graphs, this metric can also be directly used in visual models. Textual models can also be transformed into graphs, by analyzing the corresponding metamodel.

M_8. *Maintainability index*. Maintainability is an indicator of reusability [18], since a domain is in constant evolution, with new features and products being developed [19]. The maintainability index (MI) [26] was used here as an indirect measure of reusability. It combines different structural properties, such as the Halstead volume [26], the complexity of a module and the number of operators and operands in it. Modules with MI smaller than 65 are difficult to maintain, between 65 and 85 have reasonable maintainability, and larger than 85 have good maintainability.

This metric is normally used in the source code, but can also be used in code generators, since these also contain operators and operands. Template-based generators, which are used in this approach, have embedded code and special tags that represent simple operations, like conditions and iterations. The following rules were used to calculate the Halstead volume of a code generation template:

- Variables, pieces of continuous code and constants are considered **operands**;
- Query tags, like a selection of some model elements or printing of some value, are considered **operators**;
- Conditional and iteration tags are considered **operators**; and
- Embedded pieces of code are analyzed in the same way as non-embedded piece of code.

Metric M_8 cannot be applied to models, since these do not necessarily have elements that can be associated to operands and operators. In these cases, metrics that are specific to models must be used.

The maintainability of a model is determined mainly from its understandability: it has been demonstrated that the structural properties of a model that make it more or less understandable to a human being are attributes and one-to-many relationships [27], [28]. The size of a model in terms of the number of entities does not seem to be relevant. Hence, for this evaluation, two metrics were defined:

M_9. *Number of attributes.* This metric involves counting how many attributes exist in a model. In visual models (diagrams), an attribute is a property of a visual element. In textual models (programs), an attribute is a property of a first class concept as defined by the language grammar.

M_{10}. *Number of relationships.* This metric involves counting how many relationships exist in a model. In visual models (diagrams), a relationship is represented as a line between two elements, or another relational representation between two or more elements[1]. In textual models, a relationship is a reference between two first class concepts as defined by the language grammar.

Although originally conceived for E-R and class diagrams [28], the concepts of attributes and relationships are present in many graph-like languages, and thus these metrics can be used for other similar models.

We defined a baseline for M_9 and M_{10} by calculating the average of the values observed in experiments involving E-R and UML diagrams [27], [28]. We considered that models with less than **41** attributes and less than **9** relationships are more maintainable than the average.

For Q_3 and Q_4, to evaluate if the approach can lead to some perceived benefits, and identify the main difficulties, an interview was conducted with the participants after the project was concluded. It was an open interview, where the participants were simply asked to report the observed benefits and difficulties.

Figure 1 summarizes the relation between the goals, questions and metrics.

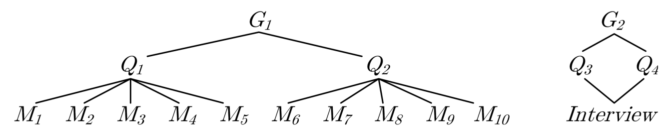


Fig. 1. Relation between the goals, questions and metrics

IV. Execution

This investigation consisted of three exploratory studies. The **first** study involved the web content authoring domain, a technical domain that includes applications for publishing information, such as news websites, forums and blogs. In this study, first a reference implementation was produced in an ad hoc, non-MDE process. Next a complete MDE infrastructure was developed, with four domain-specific languages and code generators: persistence, navigation, reports and feature configuration. Both implementations (non-MDE and MDE) were performed by one of the authors of this paper, in an academic environment.

[1]One example of such relational representation are sets in Vanderbilt's Generic Modeling Environment - available at http://www.isis.vanderbilt.edu/Projects/gme/

Study	1: Web Authoring	2: Cloud-based applications	3: Scientific domain
Participants	1 (author)	2 (including author)	2
Duration	3 months (full-time)	3 months (full-time)	2 months (part-time)
Number of DSLs	4	3	2
Number of generation artifacts	38	29	8
Size (LOC) generation artifacts	1610	2847	1375
Size (LOC) ref. implementation	5941	12127	71873
Implementation technologies	Tomcat, MySQL, Java, JSP, JSTL, Servlets, XML, SQL, Eclipse	V. Studio 2008, SQL Srv., C#, .NET Remoting, MS Volta, PNRP, DBML, Linq2SQL	Adobe Dreamweaver, PHP, MySQL
MDE technologies	Eclipse Modeling, JET (Java Emitter Templates)	V. Studio 2008, MS Text Templates, GME (Generic Modeling Environment)	Eclipse Modeling, JET (Java Emitter Templates)

TABLE I
SUMMARY OF THE STUDIES

The **second** study was performed in an industrial environment, in a large company, for the distributed cloud-based applications domain [29]. This domain presents a high degree of uncertainty regarding the underlying topology, heterogeneity of the infrastructure and environment and a high degree of cooperation between distributed components. This domain was already being explored by the company prior to this study, based on a business modeling theory featuring three domain-specific languages [30]: one to describe system data, one to describe the characteristics of the distributed elements, and one to describe the semantics of the data manipulation operations. It is a highly technical domain, involving distributed execution and cloud computing aspects. But the domain also has a business interest, as the business rules are defined using one of the DSLs. As in study 1, the non-MDE version was developed in an ad hoc process. Next, MDE-specific assets were developed for it. Both versions were developed by one of the authors of this paper together with one researcher from the company, both with experience in MDE.

The **third** study was also performed in an industrial environment, for the scientific events domain, a business domain comprising applications for paper submission, registration, secretary management, among other functions. It was performed in a company with a product line for this domain. At the time of this study, variability was being managed with the help of some reusable components, but mostly using configuration management. Each new product derived a new version that implemented the variants. Reuse occurred by retrieving a previous version that was similar to the customer requirements, and modifying it. The study was performed by a team composed of two employees with part-time availability for the project. They had no previous experience with MDE or systematic software reuse. They were both junior software engineers with around two years of experience, most of it spent in this particular company and the scientific events domain. A 24-hour training period preceded the project execution, covering the MDE approach. The existing product line, which was not developed under any particular method, was used as the non-MDE version of the domain. For the MDE version, two DSLs and corresponding code generators were produced: one textual language for feature configuration, and one textual language for the definition of certificates, an automatable subdomain identified by the participants.

Table I summarizes some relevant information regarding the three studies. All three studies took a similar amount of time, although the third study had smaller duration and only part-time availability of the participants. The third study was the largest in terms of lines of code of the reference implementation, but smaller in terms of the generation artifacts, which reflects the fact that the participants had less time to implement MDE automation.

A. Data collection

Data was collected with the help from the following tools: *Eclipse Metrics*[2] (Instability and Cyclomatic Complexity) and *JHawk*[3] (Maintainability Index). For JSP files, their correspondent Servlet code was used. We used JET templates for code generation, which are converted into Java before execution, and so these tools could also be used.

The PHP code used in the third study is not object-oriented, and no tool was found to extract these metrics for this kind of code. Since this is a large project, it was not possible to perform manual data collection for the instability, cyclomatic complexity and maintainability index metrics for the third study. So we did not collect these metrics for the third study.

For other types of assets and metrics, such as metamodels, models and code generation templates, manual data collection was carried out. Since the number of such artifacts is relatively small, this task did not take too long.

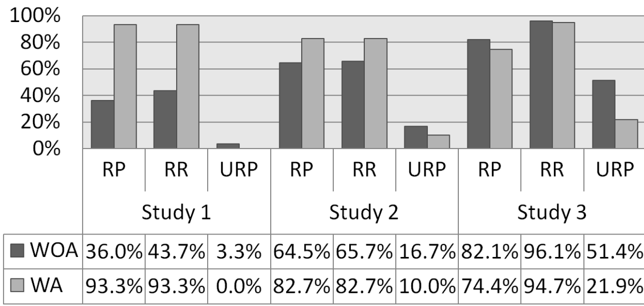
The qualitative analysis, using an interview, was only performed for the third study, because one of this paper's authors participated in the first two studies.

After collection, the data were analyzed using descriptive statistics using box plot graphics, following Fenton and Pfleeger's approach for whisker analysis [31]. Possible outliers were individually analyzed for inclusion/exclusion.

V. RESULTS AND DISCUSSION

In this section, we present the results of our observations. To facilitate the discussion, we divide it in three subsections: reuse levels, reusability and interview.

[2]http://metrics.sourceforge.net/
[3]http://www.virtualmachinery.com/jhawkprod.htm



	Study 1			Study 2			Study 3		
	RP	RR	URP	RP	RR	URP	RP	RR	URP
WOA	36.0%	43.7%	3.3%	64.5%	65.7%	16.7%	82.1%	96.1%	51.4%
WA	93.3%	93.3%	0.0%	82.7%	82.7%	10.0%	74.4%	94.7%	21.9%

Fig. 2. Reuse metrics comparison. *(WOA=Without MDE Approach, WA=With MDE Approach, RP=Reuse Percent, RR=Reuse Ratio and URP=Undesired Reuse Percent)*

A. Reuse levels

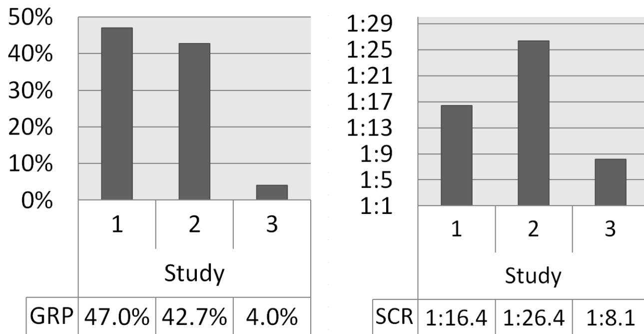
Figure 2 shows the comparison of the reuse metrics for the three studies. For study 1, considerable raises in reuse percent and ratio were observed when MDE was used, and the undesired reuse percent reduced to zero. Study 2 also presented, to a lesser extent, the same behavior. In study 3, we observed the contrary. Reuse percent and ratio decreased when the approach was used. But for the undesired reuse percent a considerable decrease was observed.

Figure 3 shows the results of the metrics for generated reuse characterization. Studies 1 and 2 had between 40-50% of generated reuse percent, while study 3 had only 4%. For the SCR metric, we observed a larger ratio in study 2 (1:26.4). Study 1 came in second (with 1:16.4), and study 3 had the smaller ratio (1:8.1).

The first observation we make is that all three projects managed to reach high reuse levels, around 80-90% (Figure 2). These values coincide, in a rough comparison, with some related work [10], [11], [12].

We observed an increase in the reuse levels when the MDE approach was used in studies 1 and 2 (Figure 2). A possible explanation is that generators produce many lines of redundant code, and thus the number of reused LOC is naturally high when the approach is used.

By further analyzing the software assets used in studies 1 and 2, we observed that, when MDE was not used, many pieces of code were produced through a manual "copy and paste" operation, followed by modifications to make them suitable to new contexts. This is very common in software projects. However, this kind of reuse did not show up in the analyzed metrics because the degree of modification was too high, above the 25% limit considered in the reuse ratio.



Study	1	2	3
GRP	47.0%	42.7%	4.0%

Study	1	2	3
SCR	1:16.4	1:26.4	1:8.1

Fig. 3. Generated reuse characterization for the three studies. *(GRP=Generated reuse percent, SCR=Specification/code ratio)*

Study 3, on the other hand, presented a small reduction of the reuse levels when MDE was used (Figure 2). By further analyzing it, we observed that the same "copy and paste" operation was being used, but the assembled product line required less modifications to configure a new product. Thus, even without MDE, the reuse level was already high. MDE was used mainly to automate small parts of this operation, but most of the code produced by the generators is fixed.

The generated reuse percent metric (Figure 3) helps to explain this difference between studies 1, 2 and 3. In studies 1 and 2 code generation is responsible for a large part of the reuse, while in study 3 only a small part of the reused code is generated. The specification/code ratio (Figure 3) also reflects the fact that the generators from studies 1 and 2 produce more code than those from study 3.

The fact that studies 1 and 2 involved technical domains, while study 3 involved a business domain also helps to explain this difference. To build new applications for studies 1 and 2, many small details need to be informed, while the business-oriented nature of study 3 allow for more similar products, which differ only by the presence or absence of some features. The technical domains of studies 1 and 2 had richer variability than the business domain of study 3.

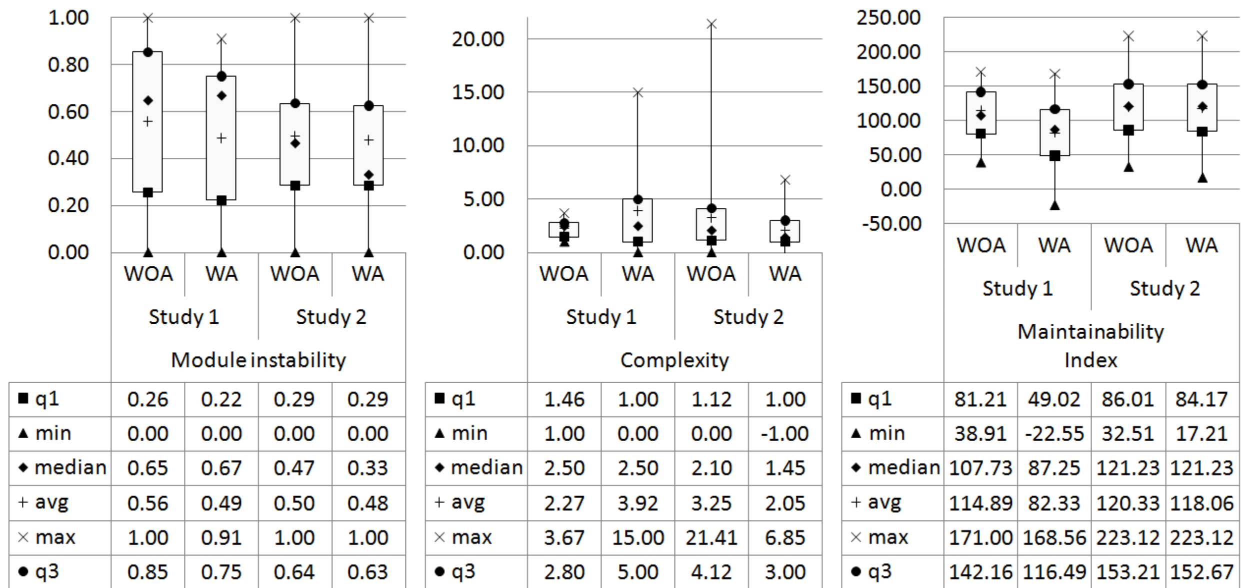
Considering only these observations, it may seem that the approach is more useful for technical domains, where it can help to increase reuse, and that it is not useful for business domains. However, the undesired reuse percent metric (Figure 2) showed a considerable reduction in study 3 when the approach was used. This is an indication that, for business domains, the approach can help to achieve more focused reuse. It also explains why the reuse levels for study 3 was reduced with the approach. The RP and RR metrics without the approach were being distorted by the presence of large pieces of unused code.

This in fact was one of the benefits highlighted by the participants during the interview (Section V-C). They stated that, without the approach, large pieces of code were just being copied into the product, even if not necessary. This code was actually prejudicing maintenance. With the approach, they managed to reduce this problem by creating code generation templates that perform a more intelligent "copy and paste" operation automatically.

B. Reusability

Figure 4 shows the results observed for the reusability metrics.

As discussed before, for study 3 it was not possible to collect such metrics. For the other studies, the MDE approach does not seem to have drastically influenced module instability. Complexity, on the other hand, presented a contradictory behavior. MDE seemed to increase the complexity in study 1



Module instability	WOA (Study 1)	WA (Study 1)	WOA (Study 2)	WA (Study 2)
■ q1	0.26	0.22	0.29	0.29
▲ min	0.00	0.00	0.00	0.00
◆ median	0.65	0.67	0.47	0.33
+ avg	0.56	0.49	0.50	0.48
× max	1.00	0.91	1.00	1.00
● q3	0.85	0.75	0.64	0.63

Complexity	WOA (Study 1)	WA (Study 1)	WOA (Study 2)	WA (Study 2)
■ q1	1.46	1.00	1.12	1.00
▲ min	1.00	0.00	0.00	-1.00
◆ median	2.50	2.50	2.10	1.45
+ avg	2.27	3.92	3.25	2.05
× max	3.67	15.00	21.41	6.85
● q3	2.80	5.00	4.12	3.00

Maintainability Index	WOA (Study 1)	WA (Study 1)	WOA (Study 2)	WA (Study 2)
■ q1	81.21	49.02	86.01	84.17
▲ min	38.91	-22.55	32.51	17.21
◆ median	107.73	87.25	121.23	121.23
+ avg	114.89	82.33	120.33	118.06
× max	171.00	168.56	223.12	223.12
● q3	142.16	116.49	153.21	152.67

Fig. 4. Reusability analysis for studies 1 and 2. *(WOA=Without MDE approach, WA=With MDE approach)*

and decrease it in study 2. Maintainability was slightly worse for study 1 with the approach, but practically the same, with or without the MDE approach, for study 2.

In order to understand the contradictory behavior observed for the complexity and maintainability metrics, we need to further analyze the studies. Intuitively, we can argue that the domain of study 2 - cloud-based applications - is naturally more complex than the domain of study 1 - web authoring - not only because it involves a wider plethora of implementation technologies (see Table I), but also because it involves complex tasks such as service discovery and peer-to-peer communication, among others. This can be seen in the complexities of the assets produced without the MDE approach, as shows Figure 4.

Apparently, MDE incurs in a complexity/maintainability cost. MDE assets seem to have an intrinsic complexity that is above average, which means that it can bring overall complexity up or down depending on the inherent complexity of the domain. In other words, with MDE, simpler domains, like in study 1, become less simple, and complex domains, like in study 2, become less complex.

To verify this explanation, we also analyzed the complexity and maintainability metrics for different kinds of assets produced by the MDE approach in each study: non-generated code, generated code and generation assets (code generation templates). Figure 5 shows this comparison.

We observed that, in study 1, the generated and non-generated code have similar complexities, but the generation assets - those produced specifically for MDE - are more complex and less maintainable. These are the assets the developer must work with when using MDE, and so overall complexity increases.

In study 2, the generated code is considerably more complex than the non-generated code and the generation assets. This is the code that has to be manually handled when MDE is not used. But when the MDE approach is used, the developer does not have to deal with it, as it is automatically generated. In this study, MDE is removing from the developer the responsibility of dealing with this complex code, and as a result, overall complexity decreases.

From this discussion we can conclude that MDE can also help in the development of inherently complex domains. However, in simpler domains, the incurred costs in complexity and maintainability must be considered.

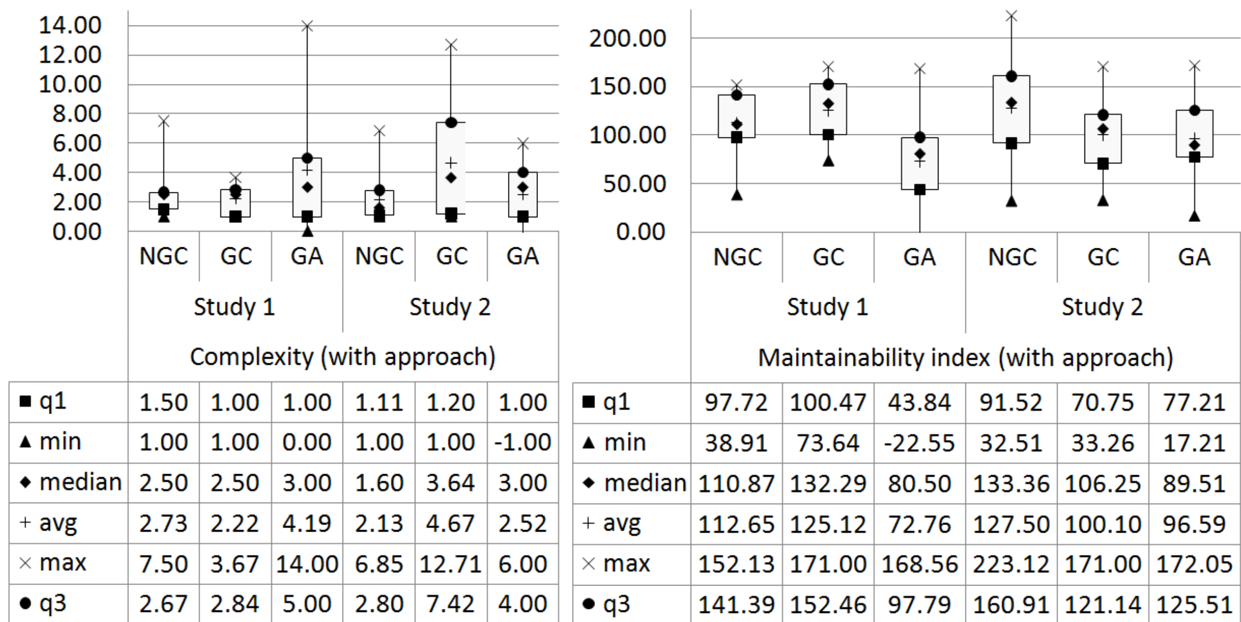
We also analyzed the models used in the studies, to determine their maintainability. Table II shows the observed results. For this analysis, we only show the average values, as the box plots did not contain any relevant information.

	NA (avg)	NR (avg)
Study 1	41.0	17.7
Study 2	9.4	4.6
Study 3	36.3	0.0

TABLE II
MODEL MAINTAINABILITY. *(NA=Num. attributes, NR=Num. relationships)*

Considering our reference values of 41 attributes and 9 relationships as the limit for "maintainable" models, in study 1 models had too many relationships, almost twice as many as the limit. For the other studies, the models were relatively simple, with a few attributes and relationships. The models for study 3 did not have any relationship, as they were simpler configuration-oriented models.

Again, for study 1 a problem with maintainability was observed when the MDE approach is used, which indicates that MDE has some associated costs that need to be analyzed when simpler domains are involved. Such result coincides with intuition: in MDE models are not just "pieces of paper" that



Complexity (with approach)	Study 1 NGC	Study 1 GC	Study 1 GA	Study 2 NGC	Study 2 GC	Study 2 GA
■ q1	1.50	1.00	1.00	1.11	1.20	1.00
▲ min	1.00	1.00	0.00	1.00	1.00	-1.00
◆ median	2.50	2.50	3.00	1.60	3.64	3.00
+ avg	2.73	2.22	4.19	2.13	4.67	2.52
× max	7.50	3.67	14.00	6.85	12.71	6.00
● q3	2.67	2.84	5.00	2.80	7.42	4.00

Maintainability index (with approach)	Study 1 NGC	Study 1 GC	Study 1 GA	Study 2 NGC	Study 2 GC	Study 2 GA
■ q1	97.72	100.47	43.84	91.52	70.75	77.21
▲ min	38.91	73.64	-22.55	32.51	33.26	17.21
◆ median	110.87	132.29	80.50	133.36	106.25	89.51
+ avg	112.65	125.12	72.76	127.50	100.10	96.59
× max	152.13	171.00	168.56	223.12	171.00	172.05
● q3	141.39	152.46	97.79	160.91	121.14	125.51

Fig. 5. Detailed complexity and maintainability analysis for studies 1 and 2. *(NGC=Non-generated code, GC=Generated code, GA=Generation assets)*

serve as a guide for the developer. They are actual part of the software and thus more effort is needed to create and maintain them.

C. Interview

The interview revealed some insight about the MDE approach. The following were the most important benefits and difficulties highlighted by the participants:

According to the participants, with MDE it was possible to more easily deal with two problems: (i) the high amount of work to perform many small changes to configure a product. For example, the inclusion of one feature required the modification of dozens of files just to include a link on the main page. With a simple code generation template and a DSL they were able to automate such task; and (ii) to facilitate the configuration, many unused files were always being included in the product. This was causing problems in maintenance. With the MDE approach, they managed to reduce the amount of unused code.

Another problem faced by the participants is that they wanted to generalize some parts of the domain, to facilitate configuration. They have tried before to create some configurable components, but in some parts of the domain, such as the generation of certificates, there are many change requests, and they ended up having to constantly modify the component after deployment. With MDE they managed to implement a DSL and code generation templates for it. The new implementation is also configurable but, after deployment, the code is more easily customized because it is specific for the product.

The participants reported difficulties to learn modeling and code generation technologies, because they had never used a generative approach before. According to them, it required a "shift in thought" from traditional software development to this new paradigm. They also had difficulties in correctly identifying the features, which is part of the model-driven reuse approach [14]. This was a conceptual problem of how to apply feature modeling, and was probably caused by inadequate training and lack of experience.

D. Summary of the observations

We summarize our observations by answering the formulated questions:

Q_1. By analyzing the same project developed with and without MDE, is there an increase/improvement in software reuse for the project that used MDE?

A_1. **Yes, for all studies we observed an increase and/or improvement in software reuse when MDE was used. For studies 1 and 2, reuse increased. For study 3, reuse decreased, but it became more focused, which we considered an improvement.**

Q_2. Are the software assets produced with MDE more reusable than those produced with a non-model-driven approach?

A_2. **It depends on the complexity of the domain. For simpler domains, MDE seems to increase complexity and decrease maintainability - hence lowering reusability. For complex domains, MDE can reduce the complexity - increasing reusability.**

Q_3. Do the developers that use MDE perceive benefits in the development?

A_3. **Yes, they perceived benefits in MDE, when dealing with product configuration and reduction of unused code in the products.**

Q_4. Do the developers that use MDE perceive difficulties in the development?

A_4. **The participants reported difficulty to learn modeling and code generation techniques.**

E. Threats to validity

First it is important to stress that the conclusions of this evaluation are weak [5], as it only involved exploratory studies. Also, the used metrics have many problems, as discussed earlier. Some are indirect measurements. Thus our observations could be distorted by some unknown effect.

A possible threat to validity is that, in all studies, the implementation obtained without the approach (D1) was used during the reference implementation development activity of the approach (D3). This may have introduced learning effects, as the development with the approach would be facilitated by previous experience. In other words, maybe it is not MDE that is leading to improvement, but experience alone. However, as discussed in the results section, most of the observed benefits are caused by code generation, and this is obviously provided by MDE, and not experience. Also, the development of a reference implementation prior to MDE-based development is supposed to happen. It is recommended in the literature [32], and is actually part of the approach [14], [15], [16]. Building code generators on the top of previous experience is an important part of MDE.

A second threat to validity is the participation of one of the authors of this paper in two of three studies, which could have influenced the results. He did not have any learning difficulties regarding the approach, and his knowledge about the related concepts was above average. Nevertheless many of our conclusions are based on objective metrics, which are not influenced by the researcher's influence. The subjective data were collected only from the third study, in which the researcher did not participate.

Another factor that could have influenced the results is the fact that, for JSP and JET assets, their corresponding generated Java code was used to collect the metrics. This is not the code dealt with by the developer, but an indirect, generated code, possibly introducing complex and auxiliary code that can be erroneously included in the quantification of complexity and maintainability metrics. The result is that the metrics are being distorted to show more complexity than there really is in these assets.

For JSP, this is not a problem, because these assets are being analyzed in comparison, so any extra complexity would be introduced in both sides being compared. For the JET assets, this extra complexity could be influencing the results of study 1, where we observed that the generation assets make simpler developments more complex, as observed in study 1. Maybe the development with MDE is not as complex as is being shown by the metrics. However, we cannot quantify this distortion. As for study 2, a possible distortion towards more complexity makes our conclusion even stronger, i.e., if code generation assets are not as complex as it is being shown, MDE is even more advantageous.

For the third study, the team's lack of experience in MDE and systematic software reuse could have influenced the results, as pointed out by the participants. For example, we observed that the MDE artifacts produced in this study was not as complex as in the other two, and the team reported difficulties in using techniques such as feature modeling. This raises the need for more intensive training before new studies are conducted, specially in larger domains.

The sizes of the projects involved in this evaluation, between small and medium, is reasonable, and thus was not considered as a serious threat to validity, although larger projects would certainly lead to much more valuable observations. The team sizes, however, was too small and did not allow us to evaluate the collaborative aspects of MDE.

Finally, these studies focused on an MDE approach where all the assets are built in-house for the reuse in a single company. Alternatives are not considered, such as reuse across projects of MDE assets built in-house or use of MDE solutions bought from vendors. While these factors may limit the generalization of the results, they represent a good practice in MDE, as demonstrated by Tolvanen [17].

VI. Conclusion

In this paper we present the results of three exploratory studies that investigated the impact of MDE on software reuse. To achieve this goal, we compared software developed with and without MDE. The most notable observation is that, in these studies, MDE helped to increase and/or improve software reuse, in different situations. We also observed that there are some costs associated with the MDE approach. Even if not supported by strong empirical evidence, this evaluation provided valuable insight regarding the combination of MDE and software reuse.

Future work involves the use of MDE in more projects, with larger teams, to explore different possibilities and other domains. In particular, we would like to perform an experimental study with a large business domain, because for the one we analyzed here it was not possible to collect all the metrics.

Currently, we have only a vague idea about when and how to use MDE to obtain better results with software reuse. There is more to be considered than the mere reduction of handwritten code caused by code generation. The introduction of MDE-specific assets may incur in extra effort and costs that need to be taken into account. We observed that MDE-specific assets are, in general, more complex and less maintainable than average. This indicates that apparently, complex technical domains have more to gain from the automation achieved with MDE.

The type of MDE automation also seems to influence the results. DSLs and more complex code generators seem to facilitate the job of the developer in technical domains, while for business domains simpler feature-based configuration seems to be enough. The repetition of these studies, with the same domains, but with a different setup, could further investigate these points.

ACKNOWLEDGMENT

The authors would like to thank the institutions that partially supported this work: FAPESP (process 2012/04549-4), CAPES (grant 0657/07-7), CNPq (grants 141975/2008-3, 305968/2010-6, 559997/2010-8, 474766/2010-1), FAPESB and the National Institute of Science and Technology for Software Engineering (INES[4]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08. We would also like to thank Microsoft Research and Aptor Systems, for the opportunity to perform our studies in their environments, and the reviewers for their insightful comments and suggestions for improving the paper.

REFERENCES

[1] C. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, no. 02, pp. 131–183, 1992.

[2] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *29th International Conference on Software Engineering 2007 - Future of Software Engineering*. Minneapolis, MN, USA: IEEE Computer Society, 2007, pp. 37–54.

[3] P. Mohagheghi and V. Dehlen, "Where is the proof? - a review of experiences from applying MDE in industry," in *ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 432–443.

[4] V. Kulkarni, S. Reddy, and A. Rajbhoj, "Scaling up model driven engineering - experience and lessons learnt," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, D. Petriu, N. Rouquette, and Ã. Haugen, Eds., vol. 6395. Springer Berlin / Heidelberg, 2010, pp. 331–345, doi: 10.1007/978-3-642-16129-2_24. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16129-2_24

[5] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721–734, August 2002.

[6] T. Weigert, "Practical experiences in using model-driven engineering to develop trustworthy computing systems," *Sensor Networks, Ubiquitous, and Trustworthy Computing, International Conference on*, vol. 1, pp. 208–217, 2006.

[7] P. Baker, S. Loh, and F. Weil, "Model-driven engineering in a large industrial context - motorola case study," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, L. Briand and C. Williams, Eds. Springer Berlin / Heidelberg, 2005, vol. 3713, pp. 476–491, dOI: 10.1007/11557432_36. [Online]. Available: http://dx.doi.org/10.1007/11557432_36

[8] Middleware Company, "Model driven development for j2ee utilizing a model driven architecture (mda) approach. productivity analysis," Report by the Middleware Company on behalf of Compuware, Tech. Rep., 2003, uRL: http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf.

[9] Modelware, "D5.3-1 industrial roi, assessment and feedback - master document. revision 2.2." Modelware - IST (no longer active), Tech. Rep., 2006.

[10] K. Krogmann and S. Becker, "A case study on model-driven and conventional software development: The palladio editor," in *Software Engineering (Workshops) - Beiträge zu den Workshops, Fachtagung des GI-Fachbereichs Softwaretechnik, 27.-30.3.2007 in Hamburg*, ser. LNI, W.-G. Bleek, H. Schwentner, and H. Züllighoven, Eds. GI, 2007, vol. 106, pp. 169–176.

[11] V. Lussenburg, T. van der Storm, J. Vinju, and J. Warmer, "Mod4j: a qualitative case study of model-driven software development," in *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part II*, ser. MODELS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 346–360. [Online]. Available: http://portal.acm.org/citation.cfm?id=1929101.1929136

[12] Y. Zhang and S. Patel, "Agile model-driven development in practice," *Software, IEEE*, vol. 28, no. 2, pp. 84 –91, march-april 2011.

[13] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering, Vol. II*, 1994, vol. 2, pp. 528–532.

[14] D. Lucrédio, R. P. d. M. Fortes, E. S. d. Almeida, and S. R. d. L. Meira, "Performing domain analysis for model-driven software reuse," in *10th International Conference on Software Reuse*, ser. Lecture Notes in Computer Science, vol. 5030. Beijing, China: Springer-Verlag, 2008, pp. 200–211.

[15] D. Lucrédio, R. P. de Mattos Fortes, E. S. de Almeida, and S. R. de Lemos Meira, "Designing domain architectures for model-driven engineering," in *4th SBCARS - Brazilian Symposium on Software Components, Architectures and Reuse - Salvador, BA, Brazil*, 2010, pp. 105–114.

[16] D. Lucrédio, "A model-driven software reuse approach (in portuguese)," PhD thesis, University of São Paulo, São Carlos, SP, Brazil, 2009.

[17] J.-P. Tolvanen, "Making model-based code generation work," *Embedded Systems Europe*, vol. 8, no. 60, pp. 36–38, Aug/Sept 2004.

[18] J. C. C. P. Mascena, E. S. d. Almeida, and S. R. d. L. Meira, "A comparative study on software reuse metrics and economic models from a traceability perspective," in *IEEE International Conference on Information Reuse and Integration (IRI)*. Las Vegas, Nevada, USA: IEEE/CS Press, 2005, pp. 72–77.

[19] E. S. Almeida, A. Alvaro, V. C. Garcia, D. Lucrédio, R. P. M. Fortes, and S. R. L. Meira, "An experimental study in domain engineering," in *33rd IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Component-Based Software Engineering Track*, 2007, pp. 93–100.

[20] J. S. Poulin and J. M. Caruso, "A reuse metrics and return on investment model," in *Proceedings of 2nd ACM/IEEE International Workshop on Software Reusability*, R. Prieto-Diaz and W. B. Frakes, Eds. IEEE Computer Society Press / ACM Press, 1993, pp. 152–167.

[21] P. Devanbu, S. Karstu, W. Melo, and W. Thomas, "Analytical and empirical evaluation of software reuse metrics," in *ICSE '96: Proceedings of the 18th international conference on Software engineering*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 189–199.

[22] J. C. C. P. Mascena, *C.R.U.I.S.E - Component Reuse In Software Engineering*. C.E.S.A.R. e-books, 2007, ch. 6 - Software Reuse Metrics, pp. 125–136.

[23] J. Poulin, "Measuring software reusability," in *Proceedings of the 3rd IEEE International Conference on Software Reuse (ICSR): Advances in Software Reusability, Rio de Janeiro, Brazil*, 1994, pp. 126–138.

[24] R. Martin, "Oo design quality metrics: An analysis of dependencies," 1994, disponível em: ¡http://www.objectmentor.com/resources/articles/oodmetrc.pdf¿. Acesso em: 14 jun 2009.

[25] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[26] K. D. Welker and P. W. Oman, "Software maintainability metrics models in practice," *Crosstalk, Journal of Defense Software Engineering*, vol. 8, no. 11, pp. 19–23, 1995.

[27] M. Genero, E. Manso, A. Visaggio, G. Canfora, and M. Piattini, "Building measure-based prediction models for uml class diagram maintainability," *Empirical Softw. Engg.*, vol. 12, no. 5, pp. 517–549, 2007.

[28] M. Genero, G. Poels, and M. Piattini, "Defining and validating metrics for assessing the understandability of entity-relationship diagrams," *Data Knowl. Eng.*, vol. 64, no. 3, pp. 534–557, 2008.

[29] D. Lucrédio, E. K. Jackson, and W. Schulte, "Playing with fire: Harnessing the hottest technologies for ultra-large-scale systems," in *15th Monterey Workshop - Foundations of Computer Software, Future Trends and Techniques for Development, September 24-26, 2008, Budapest University of Technology and Economics*, 2008.

[30] E. K. Jackson and W. Schulte, "Compositional modeling for data-centric business applications." in *Software Composition*, ser. Lecture Notes in Computer Science, C. Pautasso and Éric Tanter, Eds., vol. 4954. Springer, 2008, pp. 190–205.

[31] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, asdsad, Ed. Boston, MA, USA: PWS Publishing Co., 1998.

[32] M. Muszynski, "Implementing a domain-specific modeling environment for a family of thick-client gui components," in *The 5th OOPSLA Workshop on Domain-Specific Modeling, San Diego USA*, 2005, pp. 5–14.

[4]INES - http://www.ines.org.br