Anticipating Identification of Technical Debt Items in Model-Driven Software Projects

Ramon Araújo Gomes
ramon.gomes@trt5.jus.br
Universidade Federal da Bahia
Salvador, Bahia

Lars Thørväld
The Thørväld Group
Hekla, Iceland
larst@affiliation.org

Valerie Béranger
Inria Paris-Rocquencourt
Rocquencourt, France

ABSTRACT

Model-driven development (MDD) and Technical Debt (TD) are software engineering approaches that look for promoting quality on systems under development. MDD uses high-level abstraction models that can be transformed into application code, potentially improving system understanding and maintainability. TD, on the other hand, promotes quality through the use of strategies for detecting, quantifying, monitoring, and correcting software development problems that may hinder its maintenance and evolution. Most research on TD focuses on the application code as primary TD sources. In an MDD project, however, dealing with technical debt only on the source code may not be an adequate strategy because MDD projects should focus their software building efforts on models instead of source code. Besides, in MDD projects, code generation is often done at a later stage than creating models, then dealing with TD only in source code can lead to unnecessary interest payments due to unmanaged debts, such model and source codes artifacts desynchronization. The use of TD concept in an MDD context is also known as Model-Driven Technical Debt (MDTD). Recent works concluded that MDD project codes are not technical debt free, making it necessary to investigate the possibility and benefits of applying TD identification techniques in earlier stages of the development process, such as in modeling phases. This paper intends to analyze whether it is possible to use source code technical debt detection strategies to identify TD on code-generating models in the context of model-driven development projects. A catalog of nine different model technical debt items for platform-independent code-generating models was specified. Each catalog item provides a detection strategy to automatically find elements suspected to incur the corresponding TD type in the models. An evaluation was performed in order to observe the effectiveness of the proposed catalog compared to existing source code identification techniques found in the literature. Through three different open source software projects, more than 78 thousand lines of code were investigated. Results revealed that, although the catalog items present different precision rates, it is possible to identify and deal with these model-driven technical debts before source code is generated. We hope that sharing this initial version provides future contributions and improvements for this catalog.

KEYWORDS

model driven development, models, code smell, model smell, technical debt

ACM Reference Format:
Ramon Araújo Gomes, Lars Thørväld, and Valerie Béranger. 2020. Anticipating Identification of Technical Debt Items in Model-Driven Software Projects. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Considering the complexity of building quality software, several approaches have been proposed in recent years and have received attention from both industry and academia. Among these approaches we can highlight Model Driven Development and Technical Debt metaphor. Each one promotes software quality in a different way, while model-driven development does it through the use of high-level abstraction models as main artifact of software development, which improve understanding and maintainability of the system; technical debt does it through improvements in the software management process helping keeping in check problems that may hinder software maintenance and evolution [12].

Model Driven Development (MDD) is a software engineering approach in which modeling is a central activity over coding in building systems. Models can be transformed by automatic or semi-automatic transformations into lower and lower level abstraction models until the application source code is obtained. Therefore, MDD aims reduce the distance between problem domain and implemented solution and thus obtain software quality and maintainability gains [10].

Technical Debt (TD) is a metaphor used in software engineering to describe the costs associated with long-term implementation and design problems in software development projects [12]. For instance, rushing implementation and producing poor quality codes to meet tight deadlines is like getting a debt. Some debts can bring short-term benefits, but it needs to be paid one day through refactoring. Until the technical debt is paid, interest on it is charged in the form of greater difficulties in maintaining and evolving the system.

The technical debt metaphor was created concerning problems in the source code. However, the metaphor was gradually extended to other software artifacts [7]. There are current studies of technical debt management in architecture artifacts, design, requirements, testing and others [6]. These studies commonly refer to immature artifacts that do not meet a certain required quality level [13]. In

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

MDD project, the TD metaphor can be adapted, giving rise to the concept of model-driven technical debt (MDTD) [14]. Analogous to the original TD definition, MDTD can be defined as "the not exactly correct model that we postponed to make correct" [14]. MDTD is added to systems during the models creation and evolution, that is, in the modeling activities [14].

[13] have demonstrated that codes generated with MDD approach are not free of imperfections. Imperfections (technical debts) in models can naturally be replicated along the transformations chain until they reach code. Currently, using existing approaches to identifying technical debt, these imperfections would only be revealed after a source code review. Just as interest-bearing debt increases debt, late identification could consume projects resources in the form of greater difficulty in evolving and maintaining models simply because the debt exists for some time unnoticed. Applying TD to MDD context may anticipate problems that would only be noticed later during the coding step.

Another problem with identifying technical debts only at coding steps is that this encourages corrections to imperfections (TD payment) to be made directly in the source code. Direct code changes go against MDD principles, as they generate inconsistencies between applications code and models, so that the model no longer reflects the development system. Identifying the problems still in modeling steps would allow them to be corrected even in the models, avoiding breaking the synchronization between code and model.

Several approaches have been proposed for identifying source code technical debts [5]. Other research initiatives propose model quality metrics and patterns that identify modeling problems, such as lack of comprehensibility, completeness or consistency [20]. However, anticipating well-known source code TD identification so that they could be detected in models in MDD projects remains an open issue. In order to use the TD metaphor within this context, however, techniques need to be adapted.

As mentioned before, there is a great variety of proposals in the literature to identify TD in the source code. One of the most famous is through code smells detection. Code smells are defined as surface indications that usually correspond to a deeper problem in the system [9]. In the MDD context, it would be interesting to know what types of code smells also make sense at the modeling level. In other words, it is worthwhile to know what code smells could be identified in advance at code-generating models activities.

An effective way to automatically detect TD-incurring suspects is through the use of detection strategies. A detection strategy is defined by [19] as a quantifiable expression of a rule by which design fragments that are conforming to that rule can be detected in the source code. It means that it is a formal way to define rules that could identify patterns on the source code (or a model, on an extended definition) based on metrics or indicators. Detection strategies are an important mechanism in an attempt to automate the identification of some characteristics of a design fragment, such as technical debt or code smells.

The objective of this paper is to analyze whether it is possible to use source code technical debt detection strategies to identify TD in code generating models in the context of model driven development projects and to provide a catalog of technical debt identification strategies to detect TD in platform-independent code generator models. The catalog is composed by a set of model technical debt items. Each item consists of a technical debt type that makes sense to be identified in models and was originated from some well-known source code TD type. A model technical debt item also contains a model detection strategy, which is an adaption of its corresponding source code detection strategy for the domain of code-generating models. The model detection strategy is what allows the static analysis of models to automatically identify elements suspected to incur that TD type.

To build this catalog, we first performed a literature review on existent code smells, which are surface indications that usually corresponds to a deeper problem in the system [9]. Code smells were chosen because they are well-accepted types of source code technical debt suspects and there is a variety of proposals and tools to automatically identify them [15].

After defining the code smells to compose the catalog, the strategies to automatically identify them needed to be adapted so that they could be used in code-generating models. The adapted strategies focused on EMF technology [1], which is the Eclipse official framework for software modeling. EMF was chosen because it is one of the most popular modeling patterns for MDD projects and has a set of mature tools to support modeling activities [13]. Although the catalog focuses on EMF models, the proposed identification strategies are platform independent and therefore could be adapted to be used with other class model languages, such as UML, that is the superset of Ecore metamodel in which EMF models are based.

An exploratory study was conducted to evaluate the effectiveness of the proposed catalog in identifying the technical debts in code generating models compared to the original source code detection strategy. The study showed that, although presenting various effectiveness rates, the model technical debt items can anticipate TDs that otherwise would only be detected later in the source code. We hope to contribute to both MDD developers and researchers by providing information about what types of TD make sense to be detected in code-generating models and how they can be detected. Likewise, different software process approaches could need specific TD management.

The paper is organized as follows: section 2 presents the related works; section 3 presents the catalog of technical debt identification strategies for code-generating models; section 4 presents the exploratory study, describing the scope, planning, operation, results analysis, and presentation steps; finally, section 5 presents the final considerations, highlighting the contributions, limitations and future work.

2 TECHNICAL DEBT ASPECTS IN MODEL DRIVEN DEVELOPMENT

This section presents strategies that are related to this paper proposal. The works described here were found by a literature review. Although many studies have been conducted on technical debt studying area, there are few works that propose to apply the TD metaphor on a model driven context.

MDD advocates that models should be the primary artifact on system development. However, traditional software engineering

[1] https://www.eclipse.org/modeling/emf/

practitioners and researchers commonly focus technical debt management on coding activities. Identifying technical debt only in the source code encourages developers to pay the debts, i.e., maintain and evolve the software, also in the application code. This tends to make models obsolete and forces new changes to be made in the source code instead of the models, what goes against MDD main goal. Besides, according to the TD metaphor, the longer a technical debt inadvertently remains in an application, the higher can be the interests paid on that debt in the form of greater difficulties in maintaining and evolving the software. Thus, it would be worthwhile that TD could be identified and managed earlier in models rather than in the source code, so less interest could be paid.

MDTD (Model Driven Technical Debt) is added to systems during the creation and evolution of models, ie, in modeling activities [14]. Design failures in models caused by poor modeling can be reflected in the source code generated from the models. Because in MDD models tend to be more complete than in traditional development strategies, it is possible that many of the source code technical debts can be found in models as well. For example, a technical debt caused by a God Glass [9] existing in a code generated from a model was probably present in the original model as well and could be identified and corrected before code generation.

Many studies have been conducted on technical debt, especially in the last decade, with the area being in evidence both in academia and industry [17]. Most of the works is focused on managing architectural, design or implementation. TDs, as can be seen in [17]. Many of them (such as [5] for example) basically consist of new strategies for identifying and measuring technical debt through static application code analysis approaches. These papers, therefore, do not usually investigate how similar strategies can be used to manage technical debt in system models.

The model quality area highlights typical problems of modeling as a domain analysis tool, that is, they evaluate the ability of a model to adequately represent an abstraction of the system. It does not, therefore, address design or implementation issues and will not necessarily translate into technical debt in the application code.

Concerning models, several works have been proposed to evaluate and improve the quality of the created models. [20] presents a systematic literature review in this area. The authors identified 6 classes for model quality goals: correctness, completeness, consistency, comprehensibility, confinement, and the ability to change. Much work focuses on metrics to assess how well these goals are met and ways to improve models to meet those goals.

[22] presents an approach called model smells defined as manifestation of code smells in models. The authors present some of the model smells along with examples where they happen, and which indicators are used to identify them in the models. However, they do not present strategies for automatic identification of these smells, and furthermore, they do not perform any studies that validate whether they can even be anticipated in code-generating models in model-driven projects.

[4] presents a metric-based approach that also has the idea of anticipating the identification of design problems for system models. The paper presents metric-based mechanisms for detecting two well-known code smells (*God Class* and *Data Class*) in class diagrams. The developed mechanisms allow automatic detection of smells through the use of a tool developed for static analysis of models. The authors experiment to evaluate the accuracy of the proposed approach, but the projects in which the experiment is applied are not necessarily model driven.

Both [22] and [4] present the idea of identifying coding problems in development models. But neither explicitly treat smells as technical debts and thus does not take advantage of the benefits that metaphor use can bring in decision making about debt management (monitoring, prevention, payment). The results meantime can be used as input and starting point of this research.

Among the few papers found addressing technical debt in the model driven development context, [14] presents a position paper that discusses important issues needed to be addressed when applying the technical debt metaphor to models. The authors note that the literature lacks work on the detection, quantification, and decision making of technical debt during modeling and mapping activities and that this is an open problem.

In [12] the authors propose a strategy for technical debt calculation in models in a model driven development context. They use model quality principles to generate metrics to validate models for the presence of technical debt. The strategy created is integrated with the SonarQube [2] tool to automatically calculate technical debt on projects using EMF/GMF. The metrics used to calculate the technical debt in [12] come from the model quality area. They seek to detect problems such as the high number of metamodel elements, the high complexity of models, the excess of OCL [3] rules, etc. They are therefore more related to the understanding of models and not to coding problems. Differently, this work proposes to use already established source code TD identification techniques to detect technical debts in code generating models, based on the idea that in MDD modeling should replace coding.

Among all related works found, [23] presents the most similar approach to this paper proposal. The authors aim to investigate effects that refactoring of conceptual models have on technical debt of corresponding code. For this, they first provide a map between Model Smells of UML class diagrams (presented in[2]) and java source code issues (listed in [24]). They then provide experiments to evaluate the effects that refactoring's on a model have on the issues of the corresponding generated source code. The authors conclude that there is a significant disconnection between the TD measured on the models and the one measured on the generated source code. [11] presented a replicated study of this proposal and confirmed the conclusions.

Although our paper also intends to investigate the connection between TD on models and TD through code smells detection, there are main differences between the works that could lead to distinct conclusions. First of all, instead of using already defined model smells, we have proposed our own detection strategies to identify code smells in advance on the development models. This resulted in a distinct catalog of model smells, with only one overlapping smell between both works (*Large Class*). Another main difference is the experimental evaluation of both proposals. While [23] uses one model provided as an example by EMF Refactor tool for their analysis, we propose to study several real EMF/GMF open source

[2] https://www.sonarqube.org/
[3] https://www.omg.org/spec/OCL/2.4/PDF

projects found on github. We believe that this way we could generate a significantly higher amount of data, potentially making a deeper analysis possible.

It must also be borne that in MDD the treatment of TD in models is important since the models also work as code generators and, technical debt could go directly from model to source code. If only source code is refactored, when it is generated again, the technical debt will be incurred again, as the model is the input to model to code transformation.

3 CATALOG

This wok provides a catalog of model technical debt items created from well-known code smells, usually associated to source code technical debt. This catalog has the objective to help anticipating technical debt identification, dealing with TD at the model level instead of the source code in MDD projects. In order to facilitate the automation in the process of identifying the TD in models, for each model technical debt item, source code detection strategies were adapted and model detection strategies were proposed.

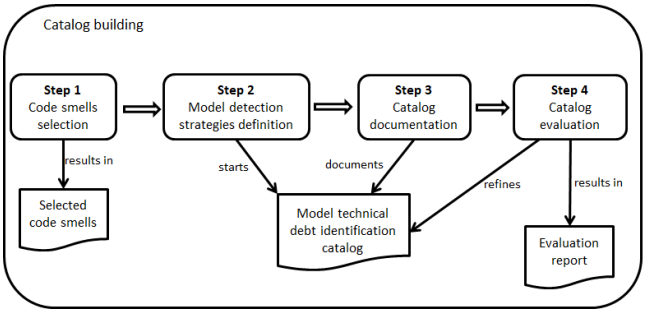

Figure 1: Steps for catalog building and evaluation.

Figure 1 presents the steps performed to build and evaluate the model technical debt catalog. The first step performed to specify the catalog was to select source code TD types that could be adapted to model elements. Our rationale was to select those one which have detection strategies that use information which could also be found in code-generating class models. In the step 2, model detections strategies were defined. While some detection strategies were already proposed by other works (such as [4]) and then could be reused or refined, others needed to be adapted from the original source code detection strategies. Step 3 consisted in documenting the catalog items. A presentation format template was defined and all items were documented according to this format. Finally, in step 4, an exploratory study was performed with the purpose of evaluating and refining the previously proposed detection strategies.

The following subsections detail the catalog specification process. Section 3.1 details how the model technical debt items that compose the catalog were selected. Section 3.2 presents the strategy used to adapt the source code detection strategies so that they could identify technical debt on the code-generating models. Section 3.3 shows what composes a catalog item and how these items are documented. Step 4 is presented in section 4, Catalog Evaluation.

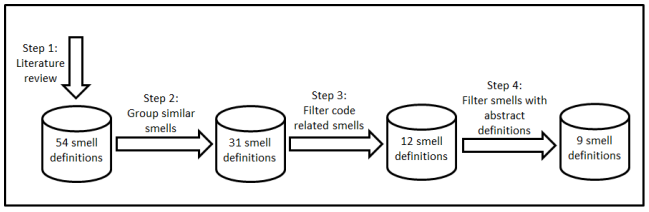

Figure 2: Steps to select catalog items.

3.1 Catalog items selection

Figure 2 presents the tasks performed to select the catalog items. The first step to build the catalog was to select source code technical debt types to be analyzed and then adapted. A literature review was performed to seek works that proposes code smells definitions and detection strategies. Works which provide either a detection strategy or well-defined description to identify code smells were selected. At the end of this step, there was a list with 54 candidate smell definitions comprising all smells defined by the selected works. The full table with all evaluated code smells and the reason why they were discarded/selected can be found in LINK.

After listing the code smells found in the literature, it was performed an analysis to find code smells that represent the same design problem, but were defined slightly different by different authors. These cases were grouped into one only candidate code smell so that we could later decide which author definition we would use to base our model detection strategy. For instance, both *Duplicated Code* [9] and *Significant Duplication* [16] define code smells to identify unwanted duplications on the source code and were grouped into a single code smell named *Duplicated Code*. The initial list of 54 smells from step 1 was then reduced to 31 smells.

Then an analysis was performed on the 31 code smells resultant list so that we could discard those that were inherently associated with metrics that could only be collected from the source code. *Long Method* is an example of discarded code smell as its considers metrics like number of code lines or cyclomatic complexity to determine whether a method is so complex that it becomes hard to understand and maintain. As this metrics cannot be found on class models, this code smell was not considered adaptable and then discarded. Other examples of code smells discarded by the same reason are *Comments*, *Duplicated Code* and *Switch Statements*.

Finally, in the last filtering step, code smells that were defined on very abstract ways and thus don't have formal detection strategies to identify them were also discarded. For instance, *Speculative Generality* [9] was discarded in this step. Although it could be potentially identified on models since its definition doesn't consider any source code specific information, it was not found on the literature any formal detection strategy to identify them in source code besides its generic definition.

After all these steps, nine items remains as candidates to compose the catalog. Table 1 (columns 1 and 3) shows the list of selected code smells along with the works that provide definitions for them.

3.2 Detection strategies definition

After selecting the model technical debt item candidates, the model detection strategies were defined. Some of the selected code smells

Table 1: Selected code smells, and their related works

TD Catalog item	Design problem description	Reference
Large Class	A class that becomes too big. In class models, usually are classes with many methods or attributes.	[9], [3]
Long Parameter List	A method has too many input parameters.	[9], [3], [4]
Shotgun Surgery	A class in which a single change has the potential to result in many small changes in other classes. It is usually associated to highly coupled and low cohesive classes.	[9], [18], [16]
Data Class	A class that presents as only responsibility storing data, which goes against object orientation best principles. The class usually has many accessor methods (getters and setters) and few service methods.	[9], [18],[4], [16]
Misplaced Class	A class has more relationships with classes of another package than its own.	[18], [4]
God Package	A package becomes too big and low cohesive. Usually happens when a package contains too many classes and this classes are low cohesive.	[18], [4]
God Class	A class that centralizes a great part of the system intelligence. This class usually are too complex (eg, has many methods) and is low cohesive.	[18], [4], [16]
Lazy Class	A class does not have enough responsibilities to justify its existence. These classes usually have few attributes and methods.	[9], [8]
Tradition Breaker	A subclass provides a high amount of services that are not related to its superclass and then breaks the inheritance purposes.	[16]

already have detection strategies for automatic identification in class models defined by related works (table 1). That was the case for *Long Parameter List*, *Shotgun Surgery*, *Data Class*, *Misplaced Class*, *God Package* and *God Class*. For these cases we used the same detection strategy defined by the original work, performing some refinements or adaption to use them on EMF domain, which is the focus of our work.

For the technical debts that do not have yet a detection strategy for models, we proposed a new strategy definition. That was the case for *Large Class*, *Lazy Class*, and *Tradition Breaker*. For these cases, sometimes there is no immediate equivalence from the attributes used on their source code detection strategy and some attribute of EMF Models. Adaptions were needed on the detection strategies to use attributes that represents similar characteristics in the models than that attributes of the original source code detection strategy.

Large Class detection strategies found in literature commonly use the number of code lines to identify the TD. Line of codes is a metric that is not present in EMF Models. However, the attributes and methods number could be used as an indicator that a model class is too large, which is the main objective of *Large Class* code smell, just like the number of code lines indicate that a Java class can be a *Large Class* instance. Then, for the catalog, we use the sum of these two metrics on the detection strategy to identify this TD on the models.

Tradition Breaker uses, in its source code detection strategy, metrics to indicate the class complexity based on the complexity of its methods [16]. As this kind of metric depends on the method implementation, which is only available at the source code, an adaption was needed and we considered each method having the same complexity, and adjusted the thresholds. *Lazy Class*, on the other hand, uses metrics that are all available in code-generation EMF models and the model detection strategy became quite similar to that of source code.

3.3 Catalog items documentation

This section details the structure required for each catalog item. Each catalog item represents a technical debt adapted to identify TD on models. The proposed catalog structure is similar to that used in [22] and contains the following information:

- `Name`: defines the technical debt identified by the detection strategy. We have used the same name as the corresponding source code smell from which the detection strategy was originated.
- `Description`: contains a textual description of the problem represented by the technical debt. This section describes when the TD occurs and what are the main impacts it can causes on the application development and maintenance.
- `Source code detection strategy`: describes which source code detection strategy was used to base the model adaptation. The detection strategy is always based on metrics that could be collected automatically and be used to identify the debt on the source code.
- `Model detection strategy`: details how we adapted the original source code detection to generate a detection strategy that could automatically detect the technical debt on the code-generating model. Also presents the resulting formal detection strategy.
- `How to pay the debt`: describes which refactoring could be applied to the models so that the TD could be removed from the software application. The refactoring strategies proposed are based on that suggested to pay the corresponding debt in the source code. They were presented in the catalog and will be subject to evaluation in future works.

Figure 3 details the *Large Class* technical debt item. The threshold limit of 20 was empirically defined and is just a suggestion for general purposes. For different project characteristics or quality requirements, other threshold values may present better results and

Name	Large Class
Description	
Large Class is described by (FOWLER, 2004) as a class that tries to do too much. On object oriented models, a class should represent a single abstraction of a given domain. When a class becomes too large, it is often an indication that this OO good practice is not respected. Then a Large Class is often traduced as a class with too many attributes or too many methods.	
Source code detection	
Large Class commonly use one of the following information to identify the code smell: • The class number of attributes and methods (NOAM), where a Large Class is considered to be a class with NOAM higher than a pre-defined threshold. • The class number of code lines (LOC), where a Large Class is a class with LOC higher than a pre-defined threshold.	
Model detection	
The detection strategy is then defined as: LargeClass = (NEC > 20) * suggested threshold Where: • Number of Elements in Class (NEC): represents the sum of the amount of attributes and methods in a class.	
How to pay the debt	
Break the class into new classes while NEC is higher than 10.	

Figure 3: Large Class catalog item

be more suitable. As well as the majority of source code detection tools perform, we admit that this threshold is configurable to attend to project specific quality requirements.

The refactoring strategy usually suggested to pay this debt is to divide the *Large Class* into smaller classes, with fewer responsibilities. It could be made by identifying other abstractions on the domain, defining classes for these abstractions and extracting methods and attributes from the *Large Class* to the new classes.

All the model technical debt items were documented with the same format. The whole catalog, with all these items, is all available at LINK.

4 CATALOG EVALUATION

As mentioned before, in MDD projects code can be generated from models. If a generated code has technical debt it is possible that this TD was originated from the corresponding code-generating model. In this case, the TD could probably be identified and even paid before the code generation by refactoring the model itself, and so preserving model and code synchronized.

An exploratory study was designed to evaluate the proposed catalog performance in anticipating technical debts in code generating model artifacts. We wish to investigate whether can be established some relationship between the code smell present in source codes generated by model transformation and the model driven technical debt catalog items found in the models.

For this evaluation we need to detect the TD from both models (using the proposed model detection strategies) and source code (using some well-accepted source code detection strategy) and compare the results to evaluate this anticipation effectiveness. As each catalog item has its particular characteristics and represents different design problems, each model detection strategy may produce different effectiveness levels and then item must be evaluated independently. The result obtained for each evaluated catalog item must be interpreted as the effectiveness of that adapted strategy in identifying specific TD. A possible low level effectiveness achieved by any catalog item does not necessarily mean that this item cannot be identified in code-generating model, but could mean that the detection strategy needs refinements.

Several laborious and non-trivial tasks had to be performed. First, projects that contains the artifacts from which the data were collected (models and code) were selected. Afterwards, elements suspects to incur TD needed to be identified both in model and source code. Due to the high number of model and source code elements to be analyzed for each catalog item evaluation, some automation was necessary. Both model and source code TD identification were then automated and as well as were the comparison between the obtained results. Altogether, 94 classes and more than 78 thousands lines of code were analyzed to this evaluation (table 2).

The study focused on projects developed using EMF/GMF models. EMF and GMF technologies were chosen because they are two of the most popular and mature tools on the MDD context and there are many public software projects developed with these technologies that can be used on the study [13].

Model detection strategies were implemented in a tool named EMF Refactor[4]. EMF Refactor is a tool that provides automatic identification of the so called Model Smells [2]. It provides a module that allows the implementation of new detection strategies to identify some design problem. Then the models can be automatic analyzed with the specified detection strategies and the elements that fit the strategy's rules are accused as suspects of having the design problem it wants to detect. We chose this tool because it was the only one found in literature that allows not only the automatic identification of model smells on EMF models but also the specification of new detection strategies. Besides, EMF Refactor was built as an Eclipse Plugin (what makes it easier to use) and has an extensive documentation.

This section details the exploratory study and discusses its results.

4.1 Study scope

We used the Goal-Question-Metric (GQM) methodology to formalize the exploratory study goal and derive metrics to help the evaluation of each catalog item effectiveness: The goal of this work is ***to analyze*** *the proposed model technical debt detection strategies* ***for the purpose of*** *evaluation* ***with respect*** *to its effectiveness on anticipating source code technical debts* ***from the point of view of***

[4]https://www.eclipse.org/emf-refactor/

*the researcher **in the context** of model driven projects developed with EMF/GMF technology.*

This goal was further refined into the following research questions:

- **RQ1**: is a code smell identified at the source code also identified as a model smell on the corresponding code-generating model?
- **RQ2**: does a model smell identified at a model result on a code smell at the corresponding generated source code?

The first research question (RQ1) is related to the capacity of the proposed detection strategies to detect technical debts at the models that would also be identified as technical debt at the generated source code. On the other hand, the second research question (RQ2) relates to evaluate whether the technical debts identified by the proposed catalog will result in technical debt at the source code. The answer to these questions can indicate how close the results obtained by the model detection strategies are to those obtained by the source code detection strategies and then provide evidence whether the source code TD can be anticipated.

For these research questions, we derived the following research metrics:

- **M1**: *Recall = True Positive / True Positive + False Negative*, , where *True Positive* are the elements that incurs on TD at both source code and model levels, and *False Negative* are elements which were pointed as TD-incurring at the source code but not at the corresponding generating model. Closely related to RQ1, recall is an accuracy metric sensitive to false negative, i.e., elements identified as code smells at the source code but that were not detected at the model level.
- **M2**: *Precision = True Positive / True Positive + False Positive*, , where *True Positive* has the same definition that it has on M1 and *False Negative* are the elements pointed as TD-incurring at the model level, but not at the source code level. Precision metric is sensitive to false positives, i.e., elements identified as model smells but that do not result on code smells at the generated source code. This metric is closely related to RQ2.
- **M3**: *Overall Agreement*. This metric is related to both RQ1 and RQ2. It measures how much of agreement (in percentage) there is on TD identification at model and source code related artifacts. For measuring it, given a catalog item, each element (model or source code artifact) is classified as TD-incurring by the model and source code detection strategy. The percentage of elements classified at the same category consists on the overall agreement.

4.2 Study design

The study comprises the following steps (figure **??**):

(1) Select the software projects where the technical debt metrics will be collected.
(2) Define a tool to help automating model technical debt identification.
(3) Select tools to perform automatic source code smells detection.
(4) Perform each item individual evaluation.

In step 1, MDD project selection, we defined a set of selection criteria to choose projects for the study. Selected projects must meet the following requirements:

- Be developed using the EMF/GMF technology.
- Be opensource and versioned at the github* public repository so that we could access the software artifacts.
- Have both model and source code available so that we could collect the metrics.
- Must have at least 100 commits to exclude most trivial projects.

Using these criteria three projects were selected to be used in the evaluation. We chose projects with different characteristics (for instance, size, number of models, number of commits) to reduce the result bias. The selected projects were Henshin*, EMF-Profile* and Service Technology*. Table 2 shows some statistics of each of them.

In step 2, model detection tool definition, we defined EMF Refactor as the tool to automate the model smell identification process as stated before. Thus the detection strategies proposed by our catalog were specified and implemented on that tool and then could be used to automatically identify the TD-incurring suspect elements in the models .

In step 3, code detection tools definition, we seek to find source code analysis tools to automatically identify the code smells adapted by our catalog. From the 9 elements of the catalog, 5 can be automatically identified at the source code by the tools found by this research. Table 3 shows the tools found and which code smells adapted by our catalog they are able to detect. PMD[5] is source code analyzer to find common programing flaws. SonarQube[6] is an automatic code review tool to detect bugs, vulnerabilities and code smells. Checkstyle[7] is a tool to help programmers write Java code that adheres to a coding standard. JSpirit[8] is a tool to assist developers in identifying and prioritizing code smells.

Finally step 4 aims to evaluate each of the evaluated catalog item. For each item, metrics M1, M2, M3 were collected and a discussion was made to answer RQ1 and RQ2. Next subsection details this process.

4.3 Catalog items evaluation

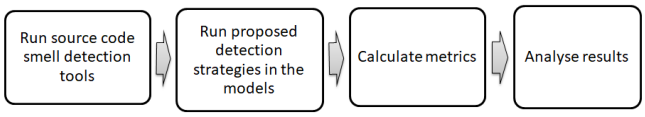

Figure 4: Item evaluation steps.

Figure 4 presents the steps performed to each evaluated catalog item. The first step was to run the source code smell automatic detection tools. As seen in table 3, for each evaluated catalog item, there is a list of tools able to detect it in the source code. Each of these tools was executed and the results were documented. The

[5]https://pmd.github.io/
[6]https://www.sonarqube.org/
[7]https://checkstyle.sourceforge.io/
[8]https://sites.google.com/site/santiagoavidal/projects/jspirit

Table 2: Information about selected projects

Projects	Number of models	Number of classes	Analyzed lines of code	Number of commits
Henshin	1	34	40871	2024
EMF-Profile List	2	7	7277	146
Service Technology	3	53	30309	8017
Total	6	94	78457	10187

Table 3: Code smells and tools to identify them.

Code smells	PMD	SonarQube	Checkstyle	JSpirit
Large Class	x	x	x	
Long Parameter List	x	x	x	
Data Class	x			x
God Class	x			x
Shotgun Surgery				x

source code detection was performed only in the code directly generated from the EMF models.

In step 2 we ran EMF-Refactor with our proposed detection strategies in the code-generating EMF models. For some code smells, we varied the thresholds so that we could compare the results and find the one who present the highest effectiveness on anticipating the code results.

In step 3, metrics M1, M2 and M3 were collected for each catalog item and for each source code detection tool. When more than one threshold was tested, we had to calculate these metrics for each of them. For the research questions assessment, we considered as good performance metric values about than 0.7.

Finally, in step 4 a discussion was made to analyze the obtained results for each evaluated catalog item. The next subsections will present this discussion along with the results obtained by the metrics calculation.

4.3.1 Large Class. The detection strategy to find Large Classes suspects in the models was defined as follows:

LargeClass = (NEC > 20), where NEC represents the sum of the amount of attributes and methods in a class.

Originally, Large Class smell is identified in source code based on lines codes. As the number of code lines is not available at the model level, we used the amount of attributes and methods to define our strategy to detect a Large Class on code-generating models. The rationale in using attributes and methods in substitution to lines of code is that usually the higher are the number of attributes and methods of a class, higher is the its source code number of lines. Particularly, in EMF domain, each attribute generates not only a corresponding attribute in the generated Java class, but also accessor methods (getters and setter), what contributes to increase the class's number of code lines. The threshold value 20 is just a suggestion and was empirically defined by testing different values in this study and analyzing the obtained results.

As presented in table 3, Large Class can be detected at the source code by three different tools: PMD, SonarQube and Checkstyle.

- `PMD`: Uses a detection strategy based on the number of lines of code to detect a Large Class. The tool allows users to set the threshold for the number of lines of code, but suggests 1000 as the default value. Thus each class with more than 1000 lines of code is considered a Large Class with this configuration. For this experiment, we ran the tool with this default value.
- `SonarQube`: Although this tool does not specifically define Large Class as a code smell, it is capable of detecting classes that have too many attributes or too many methods. For this study we considered a Large Class a class that violates any of these rules.
- `CheckStyle`: this tool also does not defines a code smell named Large Class, but defines one named FileLenght that also have the objective to detect classes that are too big. The detection strategy is the same as PMD, and we used the same threshold for this study.

Table 4 shows the results obtained by comparing the *Large Classes* found by our detection strategy with those found by each available source code analysis tool in the evaluated projects. As we can see in the table, the threshold value 20 presented better results for all metrics in the three projects.

One important thing to highlight is that none of the tools identified *Large Classes* in EMF-Profile and Service projects. Thus, there was no true positives for these projects, which made metrics M1 (recall) and M2 (precision) non applicable (N/A). Although in this case these metrics could not be collected, M3 (overall agreement) was calculated and presented 100 percent agreement, showing that our detection strategy also did not detected any suspect and thus, did not presented any false positive.

For Henshin, all tools found instances of *Large Classes*. The result analysis shows that the recall varied from 0.5 to 1, which shows that our detection strategy presented a good performance regarding false positives. It also presented medium performance for the precision metric (all of them resulted in 0.5). For M3 (overall agreement) metric, the results were very good (all close to 1), showing that the model detection strategy presented high effectiveness in anticipating the code smells detection.

In summary, *Large Class* answered satisfactorily to both research questions, but has better rates to RQ1 than RQ2. It means the model detection strategy seems to deal better with false negatives than false positives.

4.3.2 Long Parameter List. The detection strategy to find *Long Parameter* Lists in the models was defined as follows:

Table 4: Large Class results.

Project	Source code tool	NEC = 10			NEC = 20		
		M1	M2	M3	M1	M2	M3
Henshin	PMD	1	0.2	0.88	1	0.5	0.97
	SonarQube	0.5	0.2	0.85	0.5	0.5	0.94
	Checkstyle	1	0.2	0.88	0	0.5	0.97
EMF-Profile	PMD	N/A	0	0.71	N/A	N/A	1
	SonarQube	N/A	0	0.71	N/A	N/A	1
	Checkstyle	N/A	0	0.71	N/A	N/A	1
Service	PMD	N/A	0	0.92	N/A	N/A	1
	SonarQube	N/A	0	0.92	N/A	N/A	1
	Checkstyle	N/A	0	0.92	N/A	N/A	1

LongParameterList = (NOP > 6), where NOP represents the number of parameters of a method.

NOP is a metric that can be collected both from source code and code-generating models. Besides, each parameter of a method in an EMF model is usually reflected in one parameter in the generated code, which means that this metric result is expected to be the same in the model and the corresponding source code. Then, the model detection strategy didn't need any adaptation and is the same used by other authors, such as [4] and [18].

Table 3 shows that 3 different tools provide automatic identification of this code smell: PMD, SonarQube and Checkstyle. All tools use a similar detection strategy to detect suspects of *Long Parameter List*, which is based on the number of input parameters of a method. They all have configurable thresholds.

None of the projects selected for this study presented methods with the Long Parameter List either in the models or in generated code, which lead to non-applicable values for the metrics M1 and M2 (as there were no true positive values). As the model detection strategy also did not detect any *Long Parameter Lists* in the models, there was 100% of overall agreement, leading to the value 1 for M3 in all cases.

Although the selected projects did not presented any *Long Parameter List* in the source code, which impairs our evaluation, the fact that the proposed model detection strategy also did not detect any TD of this type can be seen as positive. Besides, as mentioned before, parameters of generated methods usually present a one-to-one relationship with the parameters of their original model, what make us expect that this catalog item can reach good results with projects that present this *Long Parameter Lists*.

4.3.3 Data Class. The detection strategy to identify *Data Classes* in the models was defined as follows:

DataClass = (WOC<0.33) and ((NOAM>2) and WMC<3) or ((NOAM>5) and WMC<4), where:

- `Weight Of a Class (WOC)`: corresponds to the total amount of service methods (methods that are not access only, like getters and setters) divided by the amount of interface members of the class. Classes that too many access methods will have a high value for this metric.
- `Number Of Accessor Methods (NOAM)`: corresponds to the number a access methods (getters and setters) that a class contains.

Table 5: Data Class results.

Project	Source code tool	M1	M2	M3
Henshin	PMD	0.67	0.5	0.82
	JSpirit	0.5	0.25	0.76
EMF-Profile	PMD	1	0.33	0.71
	JSpirit	N/A	0	0.57
Service	PMD	1	0.24	0.4
	JSpirit	1	0.12	0.31

- `Weighted Method Count (WMC)`: corresponds to the sum of static complexity of all methods of a class. In the source code detection strategy, this metric is given by its cyclomatic complexity, for instance. As this information cannot be calculated before code generation, an adaptation was needed for this metric. We used the same adaptation as [4], both in the metric calculation and the detection strategy thresholds. According to it, the static complexity of a method can be calculated by using the number of input parameter it has. The rationale is that methods that have too many parameters tend to be more complex, then the higher the amount of input parameters, higher is WMC.

This item can be detected by two different tools: PMD and JSpirit (table 3). They both claim to use the strategy proposed by [19], but presented slightly different results for the projects used in this study. This is probably due to different implementation details by each tool.

The results of the comparison between the detection in the models and in the generated source code can be seen in table 5. As we can see in the results of metrics M1, M2 and M3, this detection strategy presented better results for Henshin project (in which the overall agreement presented values higher than 0.7) than for the others.

In general *Data Class* did not present as good performance as the other catalog items. It presented reasonable results for metric M1, which means the model detection strategy did not detect a large amount of false negative. Then, we can say that it answers RQ1 positively. However, the proposed detection strategy did not perform well in metric M2, which means it detected several suspected *Data Classes* in the models that were not confirmed in the generated source code, i.e., answer to RQ2 was not positive.

4.3.4 God Class. The detection strategy to find *God Classes* in the models has the following definition:

GodClasses = (ATFD>4) and (NOM>=5) and (CAM<0.33), where:

- `Access To Foreign Data (ATFD)`: corresponds to the number of external classes to which one class has access. For EMF models we consider that a class A is accessing a class B if A has any attribute or method parameter of B type.
- `Number Of Methods (NOM)`: corresponds to the number of methods of a class. The original source code detection strategy used the metric Weighted Method Count (WOC), which calculates the methods static complexity, instead of NOM. As the method's static complexity cannot be calculated with a class model only, [4] in their work adapted the use of this WOC to consider that each method has a weight of 1, which leads this metric to be equals to NOM. As they obtained good results with this adaptation, we decided to use the same in our catalog.
- `Cohesion Among Methods (CAM)`: calculates the cohesion of a class based on the number of methods that have the same parameter within a class. The original source code detection strategy uses Tigh Class Cohesion (TCC) to calculate the cohesion of a class, but it uses information that is not available in a class model. As with NOM, we decided to use the same adaptation of [4] and change the use of this metric by the use of CAM.

As same as *Large Class*, the thresholds of this strategy were refined during this study. Originally, the threshold for NOM method has the value 25, but it did not obtain good results, specially presenting a lot of false negatives. After some tests, we defined the threshold for this metric as 5 as it presented better results. Table table 6 presents the results obtained by both thresholds.

This TD type can be detected by 2 different tools (table 3): PMD, and JSpirit. Just like *Data Class*, both tools claim to use the method defined by [19] to detect this code smell. Although they use the same detection strategy, they present slightly different results, what can be assigned to different implementation details.

As can be observed in table 6, the model detection strategy performance results varied a lot among the projects. For Henshin and EMF-Profile, the detection strategy presented a good performance. The model detection strategy presented some false negative, what made the recall metric (M1) vary from 0.33 to 0.6. It also presented only one false positive, what lead the precision (M2) to present high values (0.5 in one case and 1 in the rest). Regarding the overall agreement, the adapted strategies presented a good level of agreement, with the values of M3 varying from 0.57 to 0.94.

However, for the Service Technology project, the adapted detection strategy was not able to find any instance of *God Class* with any of the threshold value, while the source code detection tool found several. One possible explanation for this is the way models of Service Technology were built. They don't specify service methods in the models, which prevents the model detection strategy to find *God Classes* for this project.

4.3.5 Shotgun Surgery. It is usually associated to highly coupled and low cohesive codes. The detection strategy to find *Shotgun Surgery* in the models has the following definition:

ShotgunSurgery = (CM+CA>10) and ((CM+CA), TopValues(20%)) and (CC>5), where:

- `Changing Methods (CM)`: corresponds to the number of methods possibly affected by a change in one class. It is measured by the number of times that a class is referenced by external methods. In the models, we considered that a method refers to a class when it has one parameter of that class' type or when it overrides a method of that class.
- `Changing Classes (CC)`: corresponds to the number of client classes that may be affected by a change in the measured class. In the models, we considered that a class B can be affected by some change in a class A when B depends on or is associated with A.
- `Changing Attributes (CA)`: corresponds to the number external classes attributes with the type of a given class. It is a metric suggested by [4] to complement the effect of CM metric, that is limited in the models since we don't have access to the methods implementation in order to find all external references it makes.

The only tool capable of detecting *Shotgun Surgery* in the source code is JSpirit. It claims to use the detection strategy defined by [18]. This detection strategy is the same in which we based ourselves to propose our model detection strategy.

Table 7 shows the results of applying both the model and source code automatic detection to the projects used in the evaluation. As we can see, only Henshin presented classes suspected to be *Shotgun Surgery* instances. For this project, the model detection strategy presented several false negative, which lead to recall rate (M1=0.22). However, it did not present any false positive and then presented a high precision (M2=1). The model detection strategy also presented a good overall agreement when compared to JSpirit results (M3=0.79).

EMF-Profile and Service Technology did not present any instance of *Shotgun Surgery* either in the models or in the source code. As well as with *Large Class*, it means that metrics M1 and M2 are non-applicable, but that the model and source code detection strategies presented 100% overall agreement, what is a positive fact.

The results showed that it is possible to detect *Shotgun Surgery* in EMF-Models, but there is a need for refinements to the detection strategy to reduce the number of false negative. We could say that this catalog item answered positively RQ2 but not RQ1, however, there is a need for further tests of this detection strategy with projects that present *Shotgun Surgery* instances in the source code, so that the model detection strategy could be better evaluated.

4.4 Result discussion

In general, the catalog items answered satisfactorily to RQ1. Large Class, Data Class and Shotgun Surgery achieved recall rates that, associated to their overall agreement result, could lead is to believe that it is possible to identify these TD items in models when they are present in the source code. God Class, on the other hand, presented not so good rates for M1, what makes us believe that this detection strategy may be refined in order to decrease the number of false negatives. Long Parameter List presented inconclusive results and further research is necessary to evaluate this item.

Table 6: God Class results.

Project	Source code tool	NOM = 5			Threshold NOM =25		
		M1	M2	M3	M1	M2	M3
Henshin	PMD	0.6	1	0.94	0.4	1	0.91
	JSpirit	0.27	1	0.76	0.18	0.67	0.71
EMF-Profile	PMD	0.5	1	0.86	0	N/A	0.71
	JSpirit	0.33	0.5	0.57	0	N/A	0.57
Service	PMD	0	N/A	0.96	0	N/A	0.96
	JSpirit	0	N/A	0.75	0	N/A	0.75

Table 7: Shotgun Surgery results.

Project	Source code tool	M1	M2	M3
Henshin	JSpirit	0.22	1	0.79
EMF-Profile	JSpirit	N/A	N/A	1
Service	JSpirit	N/A	N/A	1

RQ2 also presented good results for the evaluated catalog items. Large Class, Shotgun Surgery and God Class presented very high precision rates and showed that many times an element identified as TD-incurring at a model will result in a corresponding element in the source code that also incurs TD. Data Class, on the other hand, presented a bad performance and needs refinements to decrease the false positives. As well as for RQ1, Long Parameter List also presented an inconclusive answer to this research question.

From the answers to RQ1 and RQ2, we can conclude that the exploratory study demonstrated that it is possible to anticipate source code TD detection to code-generating models. In general, although having various effectiveness rates, the evaluated catalog items answered positively the research questions. For those that presented lower effectiveness we see opportunities to refine the detection strategy and then present better results in future studies.

In our point of view, the different effectiveness rates between the catalog items do not discourage the usage of any of them. As stated in works such as [REFERENCIA] , even the code smells automatic detection tools present a strong disagreement in their results sometimes and it does not mean that they are not useful to help developers to identify suspects TD-incurring elements. Different precision rates then must be faced as opportunities to refinements so that in the future we could replace source code detection tools with code-generating model tools in the context of model driven development projects.

4.5 Threats to validity

4.5.1 ***Validity.*** One possible threat to the study validity is the great difference found by source code detection tools results. As presented in [1] and [21], source code detection tools have a high degree of disagreement in the identification code smells. This makes difficult to determine our proposed strategies effectiveness because there is no clearly correct result to which compare our detection strategies' results. As we could see in the study, even tools that claim to use the same detection strategy could present different results due to various implementation details. To mitigate this validity, a qualitative analysis was made in each tool execution so that we could explain the differences between the results and its meaning to our effectiveness performance evaluation.

4.5.2 ***Conclusion validity.*** Another possible threat to the validity is the low number of projects in which the evaluation was performed. In order to mitigate this threat projects with different characteristics (like size, number of models and number of commits) were chosen from the list that attended to the defined selection criteria. Although this mitigation action was taken, we know that the small number of projects used in the evaluation prevents a statistically relevant generalization of the results.

5 CONCLUSION AND FUTURE WORKS

Technical debts occur in every stage of system development, from requirements-related TDs to software coding, having specific causes that can be studied in isolation [17]. Likewise, as software development approaches have different activities, different TD may occur depending on the approach specificity. Direct code refactoring activities in MDD software projects could lead to break the synchronization between code and model, and so a TD to be paid on MDD projects. TD Identification is a primary activity in TD management processes.

This paper presented a catalog of model technical debt items for model-driven software projects. Our goal was to investigate whether it is possible to provide identification techniques that could anticipate the detection of well-known code smells in code-generating models, i.e., before source code generation. If a model technical debt item can be detected in a code-generating model, model refactoring could be performed then keeping model and code synchronization, which is a crucial aspect for MDD proposal.

The catalog was presented and then evaluated with 3 real open-source projects to measure the proposed model detection strategies effectiveness in anticipating the code smells. Results showed that, although presenting different effectiveness rates, the model detection strategy were able to anticipate a high amount of a TD that otherwise would be detected only in the source code.

The catalog is a starting point to other possible future works and we are planning to repeat study with a larger number of projects. This will allow for a quantitative analysis with more statistical relevant data, what can lead to some generalization possibilities. Initially, we intend to perform the same evaluation with the other items of the catalog. As there are not available automatic identification tools to all catalog items, a possible strategy to get around this problem could be to use the opinion of code smell specialists instead of automatic code smell identification tools. Other possible

future work is to perform an experiment to evaluate the proposed refactoring strategies and see if a debt payed in the model really results in a source code without the associated debt.

Finally, we intend to extend the catalog items to other types of technical debt rather than code smell only. Some possible types of TD that could be evaluated and adapted to the model driven context are, for instance, Modularity Violations and Design Patterns Grime [15] but there is a need for further research. The evaluation study methodology is detailed so other works could repeat and more items can be added to the catalog. We hope to contribute towards TD specific for MDD projects providing information about what types of TD make sense to be detected in code-generating models and how can we detect them.

REFERENCES

[1] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology* 11 (01 2012). https://doi.org/10.5381/jot.2012.11.2.a5

[2] Thorsten Arendt. 2010. *UML model smells and model refactorings in early software development phases.* Master's thesis. Philipps-University Marburg.

[3] Thorsten Arendt. 2014. *Quality Assurance of Software Models - A Structured Quality Assurance Process Supported by a Flexible Tool Environment in the Eclipse Modeling Project.* Ph.D. Dissertation. Philipps-University Marburg.

[4] Isela Macía Bertrán. 2009. *Avaliação da Qualidade de Software com Base em Modelos UML.* Master's thesis. Pontifícia Universidade Católica do Rio de Janeiro.

[5] Johannes Bohnet and Jürgen Döllner. 2011. Monitoring code quality and development activity by software maps. *Proceedings - International Conference on Software Engineering* (01 2011). https://doi.org/10.1145/1985362.1985365

[6] John Brondum and Liming Zhu. 2012. Visualising architectural dependencies. (06 2012). https://doi.org/10.1109/MTD.2012.6226003

[7] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. 2010. Managing technical debt in software-reliant systems. 47–52. https://doi.org/10.1145/1882362.1882373

[8] Phongphan Danphitsanuphan and T. Suwantada. 2012. Code Smell Detecting Tool and Code Smell-Structure Bug Relationship. *2012 Spring World Congress on Engineering and Technology, SCET 2012 - Proceedings*, 1–5. https://doi.org/10.1109/SCET.2012.6342082

[9] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA.

[10] R. France and B. Rumpe. 2007. Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering (FOSE '07).* 37–54.

[11] Sávio Freire, Amanda Passos, Claudio Sant'Anna, Rodrigo Spínola, and Manoel Neto. 2019. Influence of Model Refactoring on Code Debt: A Replicated Study. 452–456. https://doi.org/10.1145/3350768.3350793

[12] Fáber D. Giraldo, Sergio España, ManuelA Pineda, WilliamJ Giraldo, and Oscar Pastor. 2015. Conciliating Model-Driven Engineering with Technical Debt Using a Quality Framework. 199–214. https://doi.org/10.1007/978-3-319-19270-3_13

[13] Xiao He, Paris Avgeriou, Peng Liang, and Zengyang Li. 2016. Technical debt in MDE: a case study on GMF/EMF-based projects. https://doi.org/10.1145/2976767.2976806

[14] Clemente Izurieta, Gonzalo Rojas, and Isaac Griffith. 2015. Preemptive Management of Model Driven Technical Debt for Improving Software Quality. 31–36. https://doi.org/10.1145/2737182.2737193

[15] Clemente Izurieta, Antonio Vetro, Nico Zazworka, Yuanfang Cai, Carolyn Seaman, and Forrest Shull. 2012. Organizing the technical debt landscape. *2012 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings*, 23–26. https://doi.org/10.1109/MTD.2012.6225995

[16] Michele Lanza and Radu Marinescu. 2010. *Object-Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems.* Springer Verlag.

[17] Zengyang Li, Paris Avgeriou, and Peng Liang. 2014. A Systematic Mapping Study on Technical Debt and Its Management. *Journal of Systems and Software* (12 2014). https://doi.org/10.1016/j.jss.2014.12.027

[18] Radu Marinescu. 2002. *Measurement and Quality in Object-Oriented Design.* Ph.D. Dissertation. "Politehnica" University of Timisoara.

[19] Radu Marinescu. 2004. Detection startegies: Metrics-based rules for detecting design flaws. *IEEE International Conference on Software Maintenance, ICSM*, 350–359. https://doi.org/10.1109/ICSM.2004.1357820

[20] Parastoo Mohagheghi, Vegard Dehlen, and Tor Neple. 2009. Definitions and approaches to model quality in model-based software development – A review of literature. *Information and Software Technology* 51 (12 2009), 1646–1669. https://doi.org/10.1016/j.infsof.2009.04.004

[21] Thanis Paiva, Amanda Damasceno, Juliana Falcão Padilha, Eduardo Figueiredo, and Cláudio Sant'Anna. 2015. Experimental Evaluation of Code Smell Detection Tools.

[22] Parul and Brahmaleen K. Sidhu. 2016. Model Smells In Uml Class Diagrams.

[23] Gonzalo M. Rojas, Clemente Izurieta, and Isaac Griffith. 2017. Toward Technical Debt Aware Software Modeling. In *CIbSE.*

[24] SonarSource. 2020. . https://www.sonarsource.com/