Toward Technical Debt Aware Software Modeling

Gonzalo Rojas[1], Clemente Izurieta[2], Isaac Griffith[2]

[1]Department of Computer Science, University of Concepción, Concepción, Chile
gonzalorojas@inf.udec.cl
[2]Department of Computer Science, Montana State University, Bozeman, USA
{clemente.izurieta,isaac.griffith}@msu.montana.edu

Abstract. Over the last decade, the technical debt metaphor has gained in popularity, and many tools exist today that can calculate the debt associated with a miscellany of source code. However, no corpus of studies has investigated the effects that creation and refactoring of conceptual models have on technical debt of corresponding code. Our work addresses this fundamental gap by first providing a map of correspondences between recognized model smells of UML Class Diagrams and Java source code issues. We then describe a set of empirical studies to calculate the technical debt of generated source code as a result of refactorings performed on their corresponding models. Our results reveal a significant disconnect between model smells and technical debt values of resultant generated source code, and little effect of model refactorings on reducing these values. However, once correspondences between model smells and code issues are defined, model refactoring proves helpful in preventing technical debt from a high abstraction level. We exemplify this scenario by providing an in-depth example, and conclude with a discussion of results.

Keywords: Technical Debt; model driven development; software quality; software maintenance; model smells and refactoring.

1 Introduction

Technical debt accumulates over time in the form of source code that is difficult to work with and can surface as a variety of disharmonies. This concept has been the subject of numerous studies over the last few years. To date, most of the research has concentrated on management approaches –most performed at code and implementation levels through various static analysis tools. However, if practitioners are to adopt model driven techniques, then the management of technical debt also requires that we address this problem during the specification and architectural phases. According to [1], most technical debt is not incurred during the implementation phase, but from poor architectural decisions made during the design stages of software formation.

In 1992, Ward Cunningham introduced the notion of *technical debt* [2], and although the metaphor has been the subject of significant research, most of the literature is concerned with the analysis of source code issues. In 2016, however, the definition of technical debt [3] also combined implementation and design as potential sources of debt. Further, the research roadmap and vision for technical debt [4] promotes "support for upfront and continuous architectural work (vs. emergent architecture) and evidence that it helps avoid and manage technical debt". Software systems accumulate technical debt when short-term goals in software development are traded for long-term goals [5]. Many code analysis tools and techniques have been proposed to identify source code-level debt accumulated in a system [6][7][8].

As a result of the rise in popularity of this metaphor, we set about studying whether there exists a correspondence between model refactoring and overall reduction of technical debt in the resulting source code. We seek to raise the abstraction level of technical debt management, taking advantage of the claimed benefits of model-driven development (less time-consuming and error-prone code generation, easier analysis in graphical models, and predictability). By considering the "model smell" concept as a high level counterpart of code issues that incur technical debt, and model refactoring (originally explored by Sunyé et al. [9]) as the high level equivalent of the code refactoring actions [10] to reduce it, our final goal is *to apply model refactorings that are technical debt aware*.

There exist a number of factors that affect model quality [11][12]. The focus of this work is on the quality of source code generated as a result of well-known model refactoring changes made to fix model smells in UML class diagrams. Our overarching research question can be stated as: *What is the influence of Model Smells found in UML class diagrams on the technical debt of generated source code?*

To investigate, we have broken this question down into the following sub-questions:

RQ1: Which model smells have corresponding source code issues?
RQ2: How do model refactorings influence the technical debt of generated source code?
RQ3: Are there any code generation aspects that affect technical debt in the resulting source code?

To answer RQ1, we compared the definitions of well-known model smells of UML class diagrams with specifications of Java code issues, thus portraying an initial scenario where few rigorous correspondences between both groups were found. To address RQ2 and RQ3, we carried out an empirical study to analyze the effects that model refactoring on UML class diagrams have on resultant technical debt of automatically generated Java code. The study was oriented toward both removing model smells and removing code issues; while another experiment analyzed a specific model smell with a strong correspondence to a code issue, evaluating the impact of three model refactoring alternatives on technical debt. In both cases, influence of model transformation was assessed.

We choose UML class diagrams because they are a de-facto standard used to model structural aspects of systems, and Java because of its widespread use and simple syntax.

To measure the quality of class diagrams, we used 27 model smells natively supported by the EMF Refactor tool [13], while the quality of the generated source code was based on the technical debt measured by SonarQube [7].

This paper is organized as follows: in Section 2 we provide background on model driven engineering and attempts to link it to technical debt measurements. Section 3 describes our research design, and in Section 4 we present our results in three separate sub-sections (i.e., comparative study results, experimentation, and in-depth example). In Section 5 we discuss our results. We discuss the threats to the validity of the study in Section 6, and conclude with further remarks on future work and relevant new problems.

2 Background

Code smells [10] represent warnings that something may be wrong with source code. Model smells represent disharmonies at higher abstraction levels that occur in the context of a model driven process. Both are interconnected, and the concept of model smell suggests a corresponding disharmony in source code –the major source of technical debt. As stated in [14], "model smells can be defined as elements within the model that are potential candidates for improvements, being either symptoms of design defects [15] or bad alternatives to recurring design problems in OO design also known as anti-patterns" [16].

There have been previous attempts to measure the quality of models in the context of technical debt. The work performed by Giraldo et al. [19] used Moody's rule definitions [20] to characterize technical debt at the model level. Work by Izurieta and Bieman [21] compare realizations of design patterns to pattern metamodels written in RBML [22] used to characterize the abstractions of design patterns. Unfortunately, there is a dearth of available research when investigating the generation and quality of source code produced from models. In previous work presented by the authors [23] we identified the need to perform empirical experiments on models to understand how model driven disharmonies may affect code generated from said models.

According to Mens et al. [24], model refactoring "*is a specific kind of model transformation that allows us to improve the structure of the model while preserving its quality characteristics*," however, these quality characteristics may be at odds with source code quality when source code is analyzed for its technical debt. Krogtie et al.'s framework [25][26] measures physical, empirical, syntactic, semantic, and pragmatic quality, but does not measure the quality of models according to code generation capabilities that are technical debt aware, and missing from Mens et al. suggestions [24] to improve model quality is the relationship that may exist between model refactorings and the corresponding quality of the source code.

3 Research Design

Our research design consisted of three major steps; which analyze the correspondences between model smells detected in UML class diagrams and code issues of their automatically generated Java code. We consider the concept of code issue as a direct indicator of technical debt, by adopting a code analysis tool (described in Section 3.2) that assigns a precalculated amount of technical debt to each occurrence of a code issue. The steps are:

i. To systematically identify possible correspondences between model smells and code issue descriptions.
ii. To analyze a case study of a UML class diagram with several occurrences of model smells, measuring the Technical Debt of successive versions of generated code after performing model refactorings on each version.
iii. To analyze a case study with a specific model smell, to confirm and validate the correspondences made in step *i* with the same model refactoring strategy of step *ii*.

Step *i* aims at diagnosing the actual implementation of model smells in code analysis rules, under the assumption that structures were subject to similar smells at different abstraction levels, apart from the differences in the goals of models and source code. This diagnosis helped depict a baseline scenario for further analysis. We carried out the systematic review of descriptions of 27 model smells [13] and 240 code issues [7].

Once the correspondences between model smells and source code issues were made, step *ii* was aimed at complementing the static picture obtained from step *i*, by incorporating the analysis of the effect of model refactorings on technical debt associated with the detected code issues. For this, two alternative strategies for model refactoring were applied: in the first one, we chose those model refactorings that directly helped remove detected model smells; in the second one, we modified the model by refactoring the possible source of detected code issues. For both alternatives, we iteratively applied one model refactoring, measured the model smells of the new model version, generated the corresponding code automatically, and measured its code issues and technical debt.

Finally, and in order to gain more precise insights into the effects of model refactoring on technical debt, in step *iii* we isolated one model smell with a strong correspondence to a code issue (detected in step *i*), and applied the model refactoring strategy of step *ii* that removes model smell occurrences.

We used the EMF (Eclipse Modeling Framework) Refactor tool [13] to measure the model smells of UML class diagrams, while code issues and technical debt of the generated code were measured with SonarQube [7]. The SonarQube operationalization of the SQALE quality model [27] calculates technical debt by focusing on the maintainability aspect of quality; where the technical debt ratio is the remediation cost divided by the development cost, and the development costs are measured in days by multiplying the LOC by the (parameterizable) cost per line. This choice is highly justified by its wide-

spread use in code analysis, but no dominant alternative exists in the implementation of model smells. As an exception, the SDMetrics tool [28] supports the calculation of model metrics and the checking of design rules, but its proprietary specification of models is hard to integrate with UML editors with code generation capabilities. EMF Refactor is distributed as a plugin of Eclipse IDE, and thus can be easily integrated with modeling and code generation facilities provided by EMF.

4 Results

4.1 Comparative Analysis

We analyzed model smell descriptions provided by EMF Refactor and code issues from the SonarWay profile of SonarQube for the Java language. According to this analysis, we identified three distinct groups of model smells: (a) model smells with a strong correspondence to a source code issue, (b) those with a weak or indirect correspondence to a source code issue, and (c) those with no corresponding or associated source code issue. Table 1 shows the results of this analysis: from the 27 model smells natively provided by EMF Refactor, 3 were classified into group (a), 2 into (b), and most (i.e., 22) into (c).

Table 1. Classification of EMF Refactor model smells by their corresponding SonarQube issues

Group (a)	**Model Smell**	**Code Issue**
	Attribute name overridden	*Child class members should not shadow parent class members*
	Long parameter list	*Methods should not have too many parameters*
	Unnamed package	*The default unnamed package should not be used*
Group (b)	**Model Smell**	**Code Issue**
	Large class	*Classes should not have too many methods*
	Unused class	*Classes should not be empty*
Group (c)	**Model Smells with no code issue in SQube**	**Model Smells without information at code level**
	Data clumps (attributes) *Data clumps (parameters)* *Primitive obsession (constants)* *Primitive obsession (primitive types)* *Speculative generality (abstract class)* *Speculative generality (interface)*	*Abstract package* *Diamond inheritance* *Equal attributes in sibling classes* *No specification* *Specialization aggregation* *Unused enumeration* *Unused interface*
	Model Smells not considered as a code issue	**Model Smells as indicators of an incomplete model**
	Concrete superclass *Equally named class*	*Empty package* *Unnamed attribute* *Unnamed class* *Unnamed data type* *Unnamed interface* *Unnamed operation* *Unnamed parameter*

4.2 Technical Debt Measurement Experimentation

In order to assess the effects of model smell and source code refactorings on technical debt, we selected an example design of a model that could seize enough major disharmonies in its first version, so as to allow for successive refactorings in subsequent designs. We chose an example (see Figure 1) obtained from EMF Refactor's documentation.

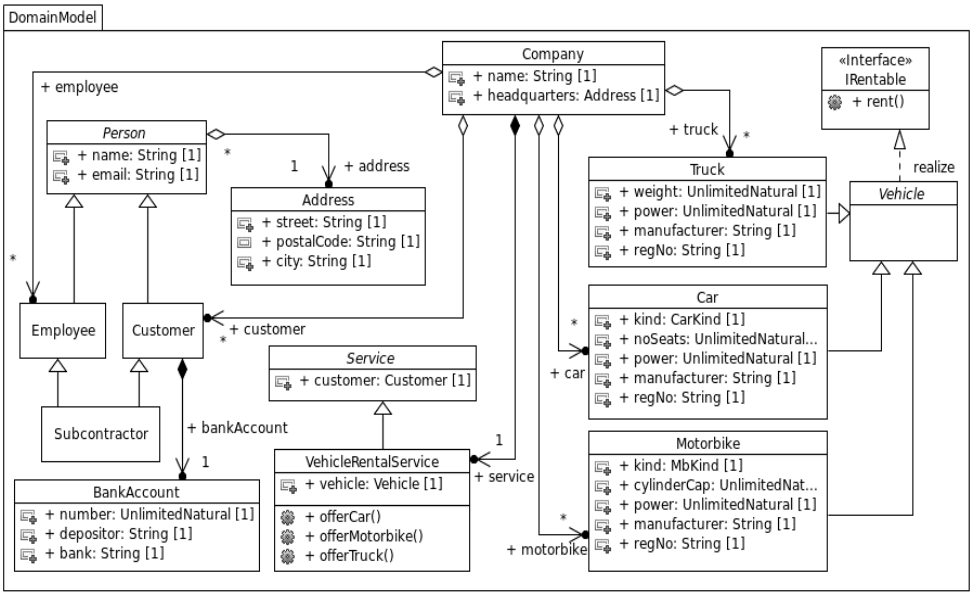

Fig. 1. Model diagram of example used for formal experimentation [29].

Tables 2 and 3 show the measures of the number of occurrences of model smells and code issues (and associated technical debt, in minutes) for this example, obtained by applying model refactoring either to remove detected model smells (Table 2), or to remove code issues (Table 3). The upper section of both tables consists of 9 model smells detected in the diagram, while the lower consists of 6 code issues of its corresponding Java code.

4.2.1 Model Smell Aware Refactoring.

Table 2 shows the results of refactoring model smells. From the original version of the class diagram (v1) to v4, we applied the *Pull up attribute* refactoring to remove the 9 occurrences of *Equal attributes in sibling classes* model smell (*regNo*, *power*, and *manufacturer*). Removal of redundant attributes also helped reduce *Data clumps* and *Primitive obsession* smells. For the same reason, generated code decreased in *Class attributes should not be public* issues, but the total number of code issues remain almost unchanged.

Table 2. Model refactoring based on model smell removal

		v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	v11
Model Smells	Equal attributes in sibling classes	9	6	3	0	0	0	0	0	0	0	0
	Large classes	8	8	7	6	6	6	6	6	6	6	6
	Primitive obsession (data types)	5	5	2	3	2	1	0	0	0	0	0
	Data clumps (attributes)	3	3	0	0	0	0	0	0	0	0	0
	Diamond inheritance	1	1	1	1	1	1	1	1	1	1	1
	Speculative generality class	1	1	1	1	1	1	1	0	0	0	0
	Speculative generality interface	1	1	1	1	1	1	1	1	0	0	0
	Unused class	1	1	1	1	1	1	1	1	1	0	0
	Unused interface	1	1	1	1	1	1	1	1	1	1	0
Code Issues	Class attributes should not be public	34 (340)	32 (320)	30 (300)	28 (280)	29 (290)	30 (300)	31 (310)	31 (310)	31 (310)	31 (310)	31 (310)
	Naming convention for packages	18 (360)	18 (360)	18 (360)	18 (360)	19 (380)	20 (400)	21 (420)	20 (400)	19 (380)	18 (360)	17 (340)
	Use "@Override" annotations	3 (15)	3 (15)	3 (15)	3 (15)	3 (15)	3 (15)	3 (15)	3 (15)	0 (0)	0 (0)	0 (0)
	Classes should not be empty	1 (5)	1 (5)	1 (5)	1 (5)	1 (5)	1 (5)	1 (5)	1 (5)	1 (5)	0 (0)	0 (0)
	Tab chars should not be used	13 (26)	14 (28)	14 (28)	14 (28)	15 (30)	16 (32)	17 (34)	16 (32)	15 (30)	15 (30)	15 (30)
	TODO tags should be handled	6 (120)	6 (120)	6 (120)	6 (120)	6 (120)	6 (120)	6 (120)	6 (120)	4 (80)	4 (80)	4 (80)
	Total	**75 (866)**	**74 (848)**	**72 (828)**	**70 (808)**	**73 (840)**	**76 (872)**	**79 (904)**	**79 (882)**	**70 (805)**	**68 (780)**	**67 (760)**

From v4 to v7, we applied the *Replace data value with object* refactoring to remove the remaining *Primitive obsession* smells, splitting affected classes. This refactoring increased the technical debt, by augmenting the *package naming convention* and *tabulation characters* code issues. The same effect occurred with the *Remove superclass* refactoring of the only *Speculative generality class* smell (*Service* class in Fig. 1), applied from v7 to v8.

From v8 to v9, the application of *Remove interface* to delete the only *Speculative generality interface* smell (*IRentable* interface in Fig.1) also removed the three occurrences of the *Use @Override annotations* smell, from the methods inherited by the three subclasses of *Vehicle*. However, the diagram lost the specification of a method. Finally, from v9 to v11, *Remove Unused Class* and *Remove Unused Interface* refactorings were applied to remove model smells caused by empty structures that were included in the model, but not in the diagram. This is conventional in model editors, and can introduce technical debt when hidden structures are not detected. In the case of *Unused Class*, the hidden class was empty, so the refactoring also removed the *empty class* issue. The remaining 7 smells (post v11) were not refactored because their removal caused new model smells to arise.

This experiment reinforces the lack of correspondence between model smells and code issues reported in the previous analysis. The effects of model refactorings on technical debt were mainly associated with the number of attributes and java files generated, but with no correspondence with the goals of the applied model refactoring.

4.2.2 Code Issue Aware Refactoring.

Table 3 shows the results of model refactoring applied to remove detected code issues. Refactoring options were chosen by identifying modeling decisions that likely caused these issues. In this way and differently from the previous case, model refactoring was not oriented towards removing detected model smells. Three model refactorings were applied, from v1 to v4, with a decrease of 53 code issues and 705 mins of technical debt. From v4, remaining code issues could not be removed from the class diagram, and code refactorings were performed, decreasing the technical debt by 41 mins. In the entire process, only one model smell was removed.

Table 3. Model refactoring based on code issue removal

		v1	v2	v3	v4	v5	v6
Model Smells	Equal attributes in sibling classes	9	9	9	9	9	9
	Large classes	8	8	8	8	8	8
	Primitive obsession (data types)	5	5	5	5	5	5
	Data clumps (attributes)	3	3	3	3	3	3
	Diamond inheritance	1	1	1	1	1	1
	Speculative generality class	1	1	1	1	1	1
	Speculative generality interface	1	1	1	1	1	1
	Unused class	1	1	1	0	0	0
	Unused interface	1	1	1	1	1	1
Code Issues	Class attributes should not be public	34 (340)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
	Naming convention for packages	18 (360)	18 (360)	0 (0)	0 (0)	0 (0)	0 (0)
	Use "@Override" annotations	3 (15)	3 (15)	3 (15)	3 (15)	0 (0)	0 (0)
	Classes should not be empty	1 (5)	1 (5)	1 (5)	0 (0)	0 (0)	0 (0)
	Tabulation chars should not be used	13 (26)	13 (26)	13 (26)	13 (26)	13 (26)	0 (0)
	TODO tags should be handled	6 (120)	6 (120)	6 (120)	6 (120)	6 (120)	6 (120)
	Total	**75 (866)**	**41 (526)**	**23 (166)**	**22 (161)**	**19 (146)**	**6 (120)**

From v1 to v2, visibility of all public attributes was switched from public to private, removing the most numerous code smell detected. In this way, the generation of v2 greatly diminishes the technical debt (by 340 mins). From v2 to v3, a simple renaming of the package name (from *DomainModel* to *domainmodel*) of the diagram was enough to remove 18 code issues and 360 mins of technical debt. From v3 to v4, a previously undetected empty class was deleted from the diagram, removing the *Unused Class* smell from the diagram and 5 mins of technical debt associated with it (i.e. 1 issue).

Adding missing *@Override annotations* and replacing *tabulation characters with whitespaces* were the code refactorings applied to corresponding issues. Even when 41 mins of technical debt were reported as paid by SonarQube, it took significantly less time to perform these actions. We chose not to refactor *TODO tags* issues, because the tags provide an aid to the developer, thus we do not consider them as technical debt.

4.3 Causal Analysis Experiment

In order to validate the strong match classification between source code issues and model smells, and to better understand the relationship between model refactorings and technical debt measurements in source code, we performed an in-depth causal analysis of these smells. For each model smell in Group (a) of Table 1, a sample Class Diagram was created, aiming to obtain several occurrences of that smell only. As an example, we discuss the results of the experiment applied to the *Attribute Name Overridden* model smell associated to the code rule "*child class members should not shadow parent class members*".

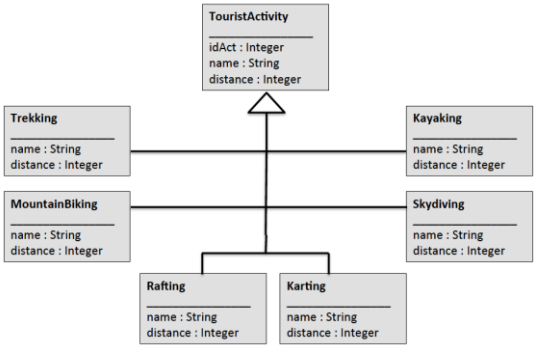

Fig. 2. Initial diagram for *Attribute name overridden* study

Figure 2 shows the initial version of the tested UML class diagram, which contains 12 occurrences of the *Attribute Name Overridden* smell (*name* and *distance* attributes of subclasses). To address their removal, three different options of model refactoring were iteratively applied: *i) Remove attribute*, in which overriding attributes were removed from subclasses one by one; *ii) Pull up attribute*, where overriding attributes were pulled up to the superclass; and *iii) Push down attribute*, where overridden attributes were pushed down from the superclass to its subclasses.

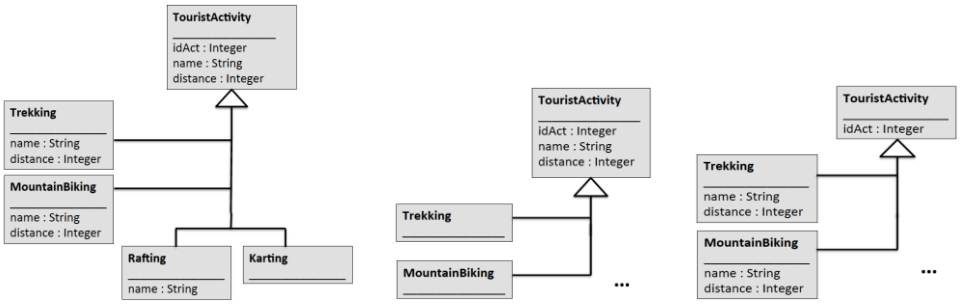

Fig. 3. Option i): Remove attribute Option ii) Pullup attribute Option iii) Pushdown attribute

The application of *Remove attribute* was predictably the option that took most iterations. Figure 3 Option i) shows version 8 of the refactored diagram, when 7 attributes from subclasses were already removed, one for each version. Option ii) shows the final version of the diagram, with two *Pull up attribute* refactorings performed, and Option iii) shows the final version for the application of two *Push down attribute* refactorings.

Table 4. Results of *Remove attribute* refactoring for *Attribute Name Overridden* model smell

	v1	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**	**v8**	**v9**	**v10**	**v11**	**v12**	**v13**
ms1	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ms2	12	6	0	0	0	0	0	0	0	0	0	0	0
ci1	**12 (60)**	**11 (55)**	**10 (50)**	**9 (45)**	**8 (40)**	**7 (35)**	**6 (30)**	**5 (25)**	**4 (20)**	**3 (15)**	**2 (10)**	**1 (5)**	**0 (0)**
ci2	24(120)	22(110)	20(100)	18 (90)	16 (80)	14 (70)	12 (60)	10 (50)	8 (40)	6 (30)	4 (20)	2 (10)	0 (0)
ci3	7 (14)	7 (14)	6 (12)	6 (12)	5 (10)	5 (10)	4 (8)	4 (8)	3 (6)	3 (6)	2 (4)	2 (4)	1 (2)
Total	43(194)	40(179)	36(162)	33(147)	29(130)	26(115)	22 (98)	19 (83)	15 (66)	12 (51)	8 (34)	5 (19)	1 (2)

Table 4 shows the progression of occurrences of model smells and code issues for the corresponding code, with the technical debt amount associated in minutes. Model smells *ms1* and *ms2* correspond to *Attribute Name Overridden* and *Equal Attributes in Sibling Classes*, respectively. Code issue *ci1* corresponds to the rule "*Child class members should not shadow parent class members*," whose correspondence with *ms1* we studied, while *ci2* and *ci3* correspond to the rules related to *@Override annotation* and *Tabulation characters*, respectively. In order to provide an example with several occurrences of *ms1*, the diagram also incurred several occurrences of *ms2*. The second column (i.e., v1) shows the values of the initial version. This example shows the exact correspondence between occurrences of the *ms1* model smell and the *ci1* code issue (bold values in Table 4).

Table 5. Results of *Pullup attribute* (left) and *Push down attribute* (right) refactoring for *Attribute Name Overridden* model smell

	Pull up attribute			**Push down attribute**		
	v1	**v2**	**v3**	**v1**	**v2**	**v3**
ms1	**12**	**6**	**0**	**12**	**6**	**0**
ms2	12	6	0	12	12	12
ci1	**12 (60)**	**6 (30)**	**0 (0)**	**12 (60)**	**6 (30)**	**0 (0)**
ci2	24 (120)	12 (60)	0 (0)	24 (120)	12 (60)	0 (0)
ci3	7 (14)	7 (14)	1 (2)	7 (14)	7 (14)	7 (14)
Total	43 (194)	25 (104)	1 (2)	43 (194)	25 (104)	7 (14)

Table 5 shows the values for both refactorings, respectively. In both cases, removing the occurrences of *ms1* only took two refactorings, corresponding to the attributes being pulled up or pushed down, respectively. In all three cases, the correspondence between the

Attribute Name Overridden model smell and the code issue associated to the rule "*Child class members should not shadow parent class members*" has been confirmed.

Evaluated model refactorings differ in the number of actions required to completely remove the measured technical debt, and the final state of the refactored model. In the first two cases, the final diagram shows all model smells were removed, and its code reflected a low technical debt principal; while in the third case, occurrences of *ms2* remained unaltered, with a technical debt amount significantly higher. Consequently, from these three alternatives of model refactoring for the *Attribute Name Overridden* smell, *Pull up attribute* allows reducing the highest amount of technical debt principal in less iterations, thus being the best technical debt aware model refactoring option.

5 Discussion

We set out to answer three research questions:

RQ1: ***Which model smells have corresponding source code issues?***

We carried out a comparative analysis study (c.f. 4.1) and found that most model smells have no corresponding code issues in SonarQube. The lack of correspondences can be explained by different reasons: *i)* the Sonar Way profile does not implement the rule because their implementation is independent of model smell definitions, *ii)* there exists no code issues associated with the model smell, *iii)* the model smell is orthogonal to good or correct implementation practices, and *iv)* the model smell is incorrectly stated.

RQ2: ***How do model smell refactorings influence the technical debt and quality of generated code?***

To answer this question we refer to Table 3 focused on model refactorings aimed at eliminating model smells, and Table 4 focused on model refactorings aimed at eliminating code issues. From Table 3, we can observe that as model smells are progressively eliminated, technical debt principal does not vary greatly, and in some cases we observe a slight growth. The results shown in Table 4 can be thought of as an exercise in reverse engineering where we tried to detect which modeling decision was responsible for the code issue detected. Once the corresponding model refactoring was applied we observed (as expected) a reduction in technical debt; however there were no changes in model smell counts. This clearly points to a need for improving the correspondence between model smells and code issues in order to support software modeling technical debt awareness.

Using an example, we showed the correspondence between model smells and code issues that have a one-to-one relationship, which allows estimating the technical debt and performing the necessary refactoring at a high abstraction level. In this way, estimation and prevention of technical debt can be performed before code is generated.

Further, we should reiterate that model smells can be removed using different model refactoring alternatives, and although all effective, some may progress quicker than others. For example, the same model (c.f. Figure 3) can be refactored using two *Pullup*, two *Pushdown*, or multiple *Remove Attribute* refactorings, however; from a technical debt point of view, some strategies are better than others. For example, *Pullup* and *Remove Attribute* perform better than *Pushdown*. This underlines that technical debt awareness is a new criterion of model refactoring that must be considered when removing model smells, and that it should be taken into consideration when developing new tools that alert software developers of possible alternatives and the effects they have on technical debt.

RQ3: ***Are there any code generation aspects that affect technical debt in the resulting source code?***

In both experiments, we detected some code issues associated to the executed model-to-text transformation rules, implemented by Papyrus' Java Code Generator plugin [30]. This tool uses tab characters instead of whitespaces, which is considered a code issue by Sonarqube, and the technical debt associated increased as the number of generated files grew. The addition of TODO tags in incompletely defined classes is also a feature implemented in model transformation rules that generate technical debt, but which greatly help developers. In this sense, the generation of @Override annotations could also be helpful, but the used tool does not include it. In summary, a technical debt-aware model transformation should implement code generation rules that consider not only the semantic correspondences between modelling and implementation artifacts, but also the code issues that can be introduced in the transformation.

6 Threats To Validity

We used Wohlin et al. [31] definitions of threats. To mitigate for *internal threats* such as unintended relationships between experiment code issues and model smell treatments with the outcome, we made sure that we did not have factors that we had no control over or had not been measured. We controlled the treatments of model refactorings (code and model) by applying them one at a time and carefully measuring the dependent variable (i.e. technical debt principal). We also performed a comparative study against well-known model smells to help with causal analysis. In order to lessen the *construct validity* of our study, we compared the model smells from EMF Refactor to some well-known smells from the current literature and found that there is variation in definitions and the strength of the correspondence to the source code issue. We documented such relationships. We also carefully choose a model example that had known model smells so we could observe their impacts on technical debt. *External validity* is threatened by the usage of a single set of rules (i.e., The Sonar Way profile) and their default values, although we are highly suspect that results would be no different under other code issue measuring techniques.

7 Conclusion

We set out to understand how model smell refactorings influence the technical debt of source code generated from models. Results showed that a significant gap exists. We suspect that this is due to three reasons. First, model smells do not have source code counterparts implemented as rules in existing tools, second, the idea of technical debt measurements performed at the modeling level is still relatively new, and third, model smell refactoring and source code issue refactoring are intended to achieve different quality goals.

Development of new rules at the source code level, and the development of more mature tools will address the first two points. To make progress in the third reason, we must tackle the well-known problem of synchronization between models and generated source code. To exemplify this problem, consider the evolution [32] of a UML class diagram whose quality is measured subject to usability characteristics, whereas the quality of the corresponding source code is measured subject to the maintainability of the latter. Both objective functions are orthogonal, thus synchronization can only progress so far until repeated amendments to the model or the source code are required as a result of changes to its counterpart. Model refactorings that improve usability may cause changes that negatively impact source code maintainability, and vice versa. We posit that refactorings performed during modeling can be technical debt aware, and that synchronization can become more manageable because the objective function is shared.

References

1. Ozkaya, I.: Agility and Software Architecture: Why Successful Teams Should Master Both. Software Engineering Institute, CMU. Presentation at MSU, Bozeman MT (2016)
2. Cunningham, W.: The WyCash portfolio management system. In: OOPSLA '92, pp. 29-30. ACM, New York, NY, USA, (1992)
3. Schloss Dagstuhl: Managing Technical Debt in Software Engineering, Dagstuhl Reports, Vol. 6, Issue 4, April 17-22 (2016). http://www.dagstuhl.de/16162
4. Izurieta, C., Ozkaya, I., Seaman, C., Kruchten, P., Nord, R., Snipes, W., Avgeriou, P.: Perspectives on Managing Technical Debt. A Transition Point and Roadmap from Dagstuhl. In: 1st Int. Workshop on Technical Debt Analytics (TDA), December 6-9, Hamilton, New Zealand (2016)
5. Zazworka, N., Vetro, A., Izurieta, C., Wong, S., Cai, Y., Seaman, C., Shull, F.: Comparing Four Aproaches for Technical Debt Identification. Software Quality J. 2,3, 403-426 (2014)
6. Findbugs (2016) http://findbugs.sourceforge.net/ Accessed 2016
7. SonarSource (2016) http://www.sonarsource.com Accessed 2016
8. Strasser, S., Frederickson, C., Fenger, K., Izurieta, C.: An automated software tool for validating design patterns. In: CAINE '11, Honolulu, HI, USA (2011)
9. Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.M.: Refactoring UML models. In: UML 2001. LNCS, vol. 2185, pp. 134-138. Springer, Heidelberg (2001)
10. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)

11. Mohagheghi, P., Aagedal J.: Evaluating Quality in Model-Driven Engineering. In: MISE '07. IEEE Computer Society, Washington, DC, USA (2007)
12. Jalbani, A., Grabowski, J., Neukirchen, H., Zeiss, B.: Towards an integrated quality assessment and improvement approach for UML models. In: SDL 2009, pp. 63-81 (2009)
13. Arendt, T., Taentzer, G.: UML model smells and model refactorings in early software development phases. Tech. Rep., Philipps Universitat Marburg, Germany (2010)
14. Misbhauddin, M., Alshayeb, M.: UML model refactoring: a systematic literature review. Empirical Software Engineering. 20, 206-251 (2015)
15. Hasker, R.W., Rowe, M.: UMLint: Identifying defects in UML diagrams. In: 2011 Annual Conf. of the American Society for Engineering Education, Vancouver, BC, Canada (2011)
16. Brown, W.J., Malveau, R.C., Brown, W.H., McCormick, H.W., Mowbray, T.J.: AntiPatterns: refactoring software architectures and projects in crisis. John Wiley & Sons, Hoboken (1998)
17. Siau, K., Tian, Y.: The Complexity of Unified Modeling Language: A GOMS Analysis. In: 22nd International Conference on Information Systems, pp. 443-447 (2001)
18. Solheim, I., Neple, T.: Model Quality in the Context of Model-Driven Development. In: MDEIS'06, pp. 27-35 (2006)
19. Giraldo, F., España, S., Pineda, M., Giraldo, W., Pastor, O.: Integrating Technical Debt into MDE. In: CAISE '14 Forum (2014).
20. Moody, D. L.: The physics of notations: Toward a scientifc basis for constructing visual notations in software engineering. IEEE Trans. Softw. Eng., 35(6), 756-779 (2009)
21. Izurieta, C., Bieman, J.: A multiple case study of design pattern decay, grime, and rot in evolving software systems. Software Quality Journal. 21, 1-35 (2013)
22. Kim, D.: A meta-modeling approach to specifying patterns. CSU PhD Dissertation (2004)
23. Izurieta, C., Rojas, G., Griffith, I.: Preemptive Management of Model Driven Technical Debt for Improving Software Quality. In: QoSA '15, pp. 31-36 ACM, New York, NY (2015)
24. Mens, T., Taentzer, G., Mueller, D.: Model Driven Software Refactoring. In: Rech, J., Bunse, C.(eds.) Model Driven Software Development: Integrating Quality Assurance. Inf. Science Reference, Hershey, NY (2009)
25. Krogstie, J., Lindland, O.I., Sindre, G.: Defining Quality Aspects for Conceptual Models. In Information Systems Concepts (ISCO3), pp. 216-231 (1995)
26. Krogstie, J.: Evaluating UML Using a Generic Quality Framework. In: UML and the Unified Process, Idea Group Publishing, pp. 1-22 (2003)
27. Letouzey J. L., Coq, T.: The SQALE Analysis Model: An Analysis Model Compliant with the Representation Condition for Assessing the Quality of Software Source Code. In: VALID 10, pp. 43-48, IEEE (2010)
28. SDMetrics (2016) http://www.sdmetrics.com Accessed 2016
29. Arendt, T., Kehrer, T., Taentzer, G.: Understanding Complex Changes and Improving the Quality of UML and Domain-Specific Models. In: MoDELS 2013. Miami, FL (2013)
30. Java Code Generation Plugin. https://wiki.eclipse.org/Java_Code_Generation Accessed 2017
31. Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A.: Experimentation in software engineering. Springer Science & Business Media pp. 102–104 (2012)
32. Khalil, A., Dingel, J.: Supporting the Evolution of UML Models in Model Driven Software Development: A Survey. Technical Report 2013-602 School of Computing, Queen's University Kingston, Ontario, Canada (2013)