The Impact of Design and UML Modeling on Codebase Quality and Sustainability

Omar Badreddin
University of Texas El Paso
obbadreddin@utep.edu

Rahad Khandoker
University of Texas El Paso
karahad@miners.utep.edu

ABSTRACT— The general consensus of researchers and practitioners is that upfront and continuous software design using modeling languages such as UML improve code quality and reliability particularly as the software evolves over time. Software designs and models help in managing the underlying code complexities which are crucial for sustainability. Recently, there has been increasing evidence suggesting broader adoption of modeling languages such as UML in open source. However, our understanding of the impact of using such modeling and design languages remains limited. This paper reports on a study that aims to characterize this impact on code quality and sustainability. We identify a sample of open source software repositories with extensive use of designs and modeling and compare their code qualities with similar in size and number of contributors code centric software repositories. Our evaluation focuses on various code quality attributes such as code smells and technical debt. We also conduct code evolution analysis over five-year period and collect additional data from questionnaires and interviews with active repository contributors. This study finds that repositories with significant use of models and design activities are associated with reduced critical code smells but are also associated with increase in non-critical code smells. The study also finds that modeling and design activities are associated with significant reduction in measures of technical debt. Analyzing code evolution over 5 years reveals that UML repositories start with significantly lower technical debt density measures but decline over time.

Keywords— Model Driven Software Development, Open Source repositories, code-centric development, UML, Technical debt, software design, software maintenance, Empirical Investigation, code smells, sustainability.

CCS Concepts—Software and its engineering → software creation and management; Designing software, Software post-development issues, Software development techniques, Software evolution.

1 INTRODUCTION

Software systems are continuing their exponential growth in complexity. This is mirrored with broad flux in the supporting middleware and platforms. These two trends, among others, are fueling the interests in managing code complexity to improve software systems maintainability and sustainability.

As in many engineering disciplines, investments in upfront designs are the primary and fundamental mechanism to ensuring sustainability and verifying key systems' quality attributes. In the software engineering spheres, there is near consensus that upfront and continuous software design using modeling languages such as UML is the foundation for the engineering of software and software-intensive systems. This consensus is supported by broad and large number of studies that demonstrate, often unequivocally, the broad benefits of design practices [1]. These benefits include improvements in program comprehension, software maintenance and refactoring tasks, as well as improvements in engineers' productivity and the quality of the code they develop. Investigations of organizations that had made the strategic decision to adopt modeling as core to their software development practices largely confirm the in-lab studies [13][14].

Infiltration of UML and design practices to the open source community has opened new pathways for investigations. Open source repositories make artifacts and their history revisions, developers contributions and communications available for investigations. This makes for a unique opportunity to assess the impact of design practices and UML on software sustainability and code quality.

In this paper we focus our investigations on assessing the effects, if any, of using UML on code quality and sustainability. We identify five code repositories with evidence of significant designs and modeling activities and compare their code quality to comparable five repositories that adopt code-centric approach. Code

quality assessment is based on code smells and technical debt. The findings suggest that 1) design activities has a role in reducing technical debt and critical code smells, and 2) counterintuitively, repositories with significant design practices demonstrated elevated levels of non-critical code smells. The findings also suggest that UML and related design activities tend to have the biggest impact early in the project lifecycle.

The paper is organized as follows. In the next Section we provide an overview of the related work. In Section 3, we introduce the study design and research questions. In Section 4, we introduce the study assessment criteria that includes code smells and technical debt. In Section 5, we report on the results and analysis. We present the results of analyzing code evolution in Section 6. In Section 7 we discuss the quality characteristics of software design and in section 8 Threats to Validity. We discuss implications of the findings and conclude the paper in Section 9.

2 RELATED WORK

Fernández-Sáez et al reports on a family of experiments to assess the impact of the Level of UML Models on code quality [1]. They conducted 8 controlled experiments in total and their findings suggest that code quality is slightly better when using low level UML diagrams, especially if used for the modification of the source code. They also find that high level models appear to be more helpful in understanding the system. All experiments were conducted using small examples with student participants. Interestingly, they found that participants tried to minimize or avoid using the models. Their interpretation that this is a result of using small size examples and code.

Another study investigated the effectiveness of forward engineering and reverse engineering on maintenance tasks [2]. They find that models produced at design time (forward designed diagrams) are more effective than reverse engineered diagrams. Their studies were conducted with undergraduate and graduate student participants, and the artifact they used were comprised of small systems.

Izurieta et al advocate for primitive management of technical debt by addressing it early in the design and architecting activities [3]. In their study, they explore different aspects that must be handled to better manage technical deb.

Scanniello et al conducted long term investigations that included 333 observations of both PhD and practitioners in different contexts as they performed comprehensibility tasks [4]. Results suggest that the use of UML models affects source-code comprehensibility, but in two opposite directions. In particular, models produced in the analysis phase reduce source-code comprehensibility and increase the time to complete comprehension tasks, while models produced in the design phase improve source-code comprehensibility and reduce the time to complete comprehension tasks.

In Model-Driven Engineering (MDE) projects, most - or at least some- code is automatically generated from models. He et al. analyzed 16 MDE projects and found that the generated code contains more code smells that what software developers would normally produce [15]. Besides this study by He et al., there seems to be limited literature on technical debt in the context of modeling and MDE [17]. Izurieta et al. in a position paper discussed basic concepts of technical debt in the context of MDE but did not provide code-level analysis [16].

3 STUDY DESIGN

The Goal of this study is to investigate the impact of UML and associated design activities on quality attributes reflecting software sustainability, maintainability, code smells, and technical debt. We investigate this question by analyzing code repositories with significant modeling and design activities as evident in the presence of various design and modeling artifacts and confirmed by the repository most active code contributors. We compare these model-heavy repositories with other comparable repositories that followed a code-centric approach as evident in their artifacts and confirmed by their contributors. In the process of selecting the sample repositories, we took into consideration code size, programming language, number and experience of repository contributors, popularity of the codebase, and the overall codebase contributions activities.

3.1 Research Question

The study has the following research questions.

RQ1: Does UML and associated design activities have an impact on code quality attributes?

Our quality attributes focus on measurements of code smells and other measurements of technical debts.

RQ2: What are the key quality attributes that are affected by the design and modeling activities? If there is in fact a measurable effect of design and modeling on code quality in practice, we aim to understand the primary quality measurements that are affected. In addition to results from code analysis, we investigate this question by analyzing codebase evolution overtime and collecting data from repository contributors.

RQ3: What is the perceived effects of UML and design activities (or lack thereof) on maintainability and technical debt? We want to investigate the perceptions of

repository participants and contributors on whether UML and design activities have an impact on maintainability and technical debt.

3.2 Subject Systems

The study identifies 10 subject systems, 5 identified as model-heavy repositories [6], and 5 identified as code-centric repositories. In the following, we describe our subject systems selection process.

The first 5 subject systems are selected from a pool of 4,237 identified in [6] to be model-heavy repositories. These 4,237 repositories were selected by mining all GitHub repository artifacts and selecting those that included UML and modeling elements. From this list, we selected 5 repositories that meet the following criteria, code size is greater than 150K lines of code written predominantly in an object-oriented language.

The second 5 subject systems are selected by identifying comparable repositories. We queried GitHub for repositories with similar object-oriented code size, number of active contributors, and similar programming languages. We ensured that the average expertise of the active contributors is comparable to the expertise of the contributors of the identified repositories. The table 1 below lists all 10 subject code repositories, their number of commits, code size, primary programming language, and number of active contributors. The analyzed LoC column lists the lines of code that were analyzed in this study. This excludes non object-oriented code and documentations.

In the analysis, we only considered major programming languages that fall within the following list: C/C++, JavaScript, C#, Java, COBOL, TypeScript, PL/SQL, PL/I, PHP, ABAP, T-SQL, VB.NET, VB6, Python, RPG, Flex, Objective-C, and Swift. Other languages were excluded. This exclusion was performed at the code analysis step, not repository selection step. The rational is to minimize bias in repository selection particularly that most large repositories often include lines of code outside of the identified major languages. As a result, there is an important distinction between code size that includes the entire repository codebase, and the analyzed LoC, which is limited to the lines of code that belong to one of the aforementioned programming languages.

Table 1: Subject Software repositories

	Repository	Commits	Code Size	Primary Programming Lang.	Analyzed LoC		Active Cont'rs
					Count	%	
UML	Marble	9,090	265,546	C++	95,157	36	100
	Oryx-editor	2,022	640,127	JavaScript, Java	543,704	85	10
	101repo	2,312	183,083	PHP, JavaScript, Java, C#	154,437	84	25
	Activiti	7,741	207,339	Java	192,812	93	151
	Poi	9,157	450,906	Java	427,326	95	11
Non-UML	Selenium	21,788	875,267	JavaScript, Java, C#, C++	775,268	89	341
	Fastjson	2,673	168,880	Java	149,186	88	69
	Mal	2,249	178,870	Visual Basic, Swift	166,296	93	62
	Deeplearning4j	9,301	283,711	Java, JavaScript	221,711	78	139
	Phabricator	15,001	508,264	PHP, JavaScript	470,232	93	198

4 ASSESSMENT CRITERIA

Our assessment criteria are based on two primary measurements; measurements of code smells and measurements of technical debt. In the following we describe these measures.

4.1 Code Smells

Code smell is any symptom in the source code of a program that possibly indicates a deeper problem. A code smell is a surface indication that usually corresponds to a deeper problem in the system [7]. To identify code smells in the subject repositories, we used SonarQube supported by CodeSmell plug-in for additional detailed code smell analysis. SonarQube is considered the defacto to industrial standard in identification of code smells and quantification of technical debt and has been used in numerous code analysis studies [5]. We also utilized SonarQube GitHub plug-in to facilitate integration with GitHub repositories. We utilized the plug-in 3D code metrics for improved visualizations of code metrics.

SonarQube classifies code smells into 5 categories; critical, blocker, major, minor, and information. Studies report that developers agree in general with the SonarQube code smells and their severity [8]. We excluded

information category from our analysis because it is not closely related to code quality. The table 2 below reports on the code smells for all 10 code repositories. To calculate the weighted total, we assign the following weights: critical = 4, blocker = 3, major = 2, and minor = 1.

We calculate weighted smell density by dividing the weighted total over the analyzed lines of code and multiplied by 100. This measure is key to our study as the goal is to assess code quality, and not simply the total number and severity of code smells. But since the code smell weights are arbitrary, we also calculate code smell density for each category of code smells using the analyzed lines of code as shown in Table 2.

Table 2: Code Smells

	Code Smells Density								Weighted Total	Weighted Smell Density
	Critical		Blocker		Major		Minor			
	Count	Density	Count	Density	Count	Density	Count	Density		
UML	1	0	0	0	13,000	13.66	507	0.53	26,511	27.86
	1,100	0.20	350	0.06	7,800	1.43	15000	2.76	36,050	6.63
	632	0.41	59	0.04	5,700	3.69	5600	3.63	19,705	12.75
	1,600	0.83	97	0.05	1,300	0.67	2000	1.04	11,291	5.8
	4,300	1.01	228	0.05	5,200	1.22	8500	1.99	36,784	8.6
Mean	**1,526.6**	**0.49**	**146.80**	**0.04**	**6,800**	**4.14**	**6,321.4**	**1.99**	**26,068.2**	**12.34**
Non-UML	876	0.11	317	0.04	7,200	0.93	1900	0.25	20,755	2.6
	962	0.64	33	0.02	3,100	2.08	4600	3.08	14,747	9.8
	685	0.41	95	0.06	6,200	3.73	6300	3.79	21,725	13.06
	1,300	0.59	175	0.08	11,000	4.96	8200	3.70	35,925	16.20
	5,400	1.15	1	0.0	4,900	1.04	3200	0.68	34,603	7.3
Mean	**1,844.6**	**0.58**	**122.33**	**0.04**	**6,480**	**2.55**	**4,840**	**2.30**	**25,551**	**9.84**

4.2 Technical Debt

Technical debt (also known as code debt) is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer time. There is a fundamental difference between code smells and technical debt. First, not all code smells are certain to contribute to maintenance and quality problems. Moreover, the effort required to address code smells is largely independent of the code smells itself [9].

SonarQube computes technical debt based on the Software Quality Assessment which is based on Lifecycle Expectations methodology (SQALE) [10]. The SQALE is a methodology that organizes non-functional requirements related to code quality. Non-functional requirements are realized in terms of coding rules and issues in the SonarQube implementation of the SQALE method. Table 3 below lists the calculated Technical Debt (TD), and the TD density obtained by dividing the TD value over the analyzed LoC and multiplied the value by 100.

Table 3: Technical Debt

	Repository	TD	TD Density
UML	Marble	163	0.17
	Oryx-editor	486	0.09
	101repo	386	0.25
	Activiti	122	0.06
	Poi	322	0.08
Mean		**298.8**	**0.13**
Non-UML	Selenium	217	0.03
	Fastjson	196	0.13
	Mal	417	0.25
	Deeplearning4j	720	0.21
	Phabricator	501	0.11
Mean		**410.2**	**0.15**

4.3 Code Evolution Analysis

We analyzed the codebase evolution over the 5-year period. Over this period, we analyzed 25 repository revisions equally spaced over the 5-year duration. The goal is to understand the code quality evolution over an extended period of time.

4.4 Contributors Perceptions

We designed an open questionnaire that focuses on identifying the design techniques and methodologies of the subject repositories. We solicited responses from the

set of most active contributors of the 10 subject repositories. The questionnaire was anonymous and included 11 main questions with follow up sub-questions. Some questions focus on profiling the contributors themselves and other questions are related to the subject repositories under study. To minimize bias, we distributed equal number of questionnaires to each repository. Solicitation emails were customized for each contributor and each repository to maximize responses. In addition, we offered $100 gift card for the first five responses. Follows are the 11 main questions in the distributed questionnaires.

Q1: How many years of software development experience do you have?

Q2: What is the highest level of education you have obtained?

Q3: Rank your skills from high (5) to low (1) in software development?

Q4: Describe formal software development education and training you may have completed?

Q5: How you communicate with other contributors of the project?

Q6: Do you develop or use design models (using UML for example) for this project?

Q7: Do you use models to understand the system?

Q8: Are system models up to date with the code? Do you maintain system models as the code evolves?

Q9: Is there an overall design, architecture, or model that you often refer to? And what impact does those design have on the project and its code quality?

Q10: In your opinion, does the presence of design models affect how new developers contribute to the project?

Q11: Do you consider your software to be well designed, and the code is of high quality?

The first goal of the questionnaire is to validate the code repository classification; specifically, to ensure that those repositories identified to have significant level of modeling do indeed exhibit significant levels of modeling, and ditto for non-UML code repositories. We also aim at ensuring that code repositories that included modeling artifacts reflect significant level of modeling and design activities (i.e. to exclude projects that stored modeling artifacts without engaging in design activities). This serves to minimize threats to validity of inadequately classifying code repositories.

The second goal of the questionnaire is to investigate contributors' perceptions of the repositories understudy. Specifically, the aim is to uncover whether repositories most active contributors attribute code quality characteristics to the presence, or lack thereof, of design models of the code.

The third goal of the questionnaire is to collect additional profiling information from the primary contributors that are not available from GitHub platform. The profiling information related to years of experience, formal training, and academic education contribute to the 'causality' investigations; i.e., whether the resulting code quality characteristics is most likely caused by the use of modeling, or due to other factors such as expertise of the primary contributors, or other factors.

We also conducted interviews and questionnaires with contributors who agreed to be contacted for further information. In those sessions, we shared summary results of the study, and detailed results for the contributors' code repository.

5 RESULTS AND ANALYSIS

The results are organized along the three main assessment criteria; code smells, technical debt, and repository contributors' perceptions.

5.1 Code Smells

We distinguish between two measures of code smells. The first is the code smell count that reports on the number of various types of code smells. The second is code smell density, which is the code smell count divided by the analyzed LoC. Table 4 and Table 5 illustrate the results for these two measures respectively.

Table 4: Code Smell Count results

Repository	Code Size	Code Smell (Count)				Weighted Code Smell Count
		Critical	Blocker	Major	Minor	
All UML Repositories	1,747,001	7,633	734	33,000	31,607	130,341
All non-UML Repositories	2,014,992	9,223	621	32,400	24,200	127,755
Gap (UML – non-UML)	-267,991	-1,590	+113	+600	+7,407	+2,586
Gap %		**-20.83%**	**+15.93%**	**+1.81%**	**+23.43%**	**+1.98%**

Table 5: Code Smells Density results

Repository	Analyzed LoC	Code Smell (Density)				Weighted Smell Density (/m. LoC)
		Critical	Blocker	Major	Minor	
All UML Repositories	1,606,248	2.45	0.21	20.68	9.94	38.42
All non-UML Repositories	1,782,693	2.90	0.20	12.74	11.50	27.59
Gap (UML – non-UML)	-176,445	-0.46	0.01	7.94	-1.56	+10.83
Gap %		**-18.78%**	**4.76%**	**38.4%**	**-15.69%**	**28.18%**

We observe that the number of critical code smells is significantly reduced for UML code repositories (20% reduction in count and 19% reduction in density). However, there is some increase in blocker (16% in count and 4.8% in density), major (2% in count and 38% in density), and minor code smells (23% in count). 15.7% reduction of minor code smell density is also noticeable.

5.2 Technical Debt

Analysis of technical debt data suggests a significant reduction in total technical debt in UML repositories. The data shows a 572-day reduction in total technical debt, or about 114 per code repository. This translates to 1.9 hours on average per line of code. Compared to total TD in non-UML repository, this account for about 39% TD reduction (Table 6). In terms of TD density per LoC, data shows a 0.08 reduction (12%).

5.3 Contributors' Perceptions

We sent out the questionnaire to the fifteen most active contributors of each subject code repository totaling 150 solicitations. To increase the response rate, each solicitation included the project name and the name of the contributor. We received nine responses, five from UML projects and four from non-UML projects. We excluded one response from the UML pool and two responses from non-UML pool who indicated they are not qualified to answer the questionnaire. We followed the grounded theory methodology to analyze the data [12]. We present the results of the questionnaire along the three aforementioned questionnaire goals.

5.3.1 Responses from UML repositories. Three respondents from Marble, 101repo, and Activiti confirm high level of modeling artifacts and design activities as part of the development activities. All three also confirm that models are used routinely in their comprehension activities. One respondent for Poi also confirms use of modeling, but reports using only architectural and high-level models for documentation purposes only. Poi respondent did not report using models as part of the comprehension activities. Two respondents report putting efforts to maintain synchronization between the models and code (Marble and 101repo). Respondents from Poi and Activity do not report efforts to maintain the synchronization. Three code repositories (Marble, 101repo, and Activity) maintain high-level software design that aids in the on-boarding of new contributors. Poi does not report any role for models in aiding in the on-boarding of new contributors. All four responses perceive their code repository to be of high quality. The only exception was a few legacy code segments in Marble.

Marble reported the automated generation of code from models. However, the respondent confirmed that generated code is never committed to the repository, which is the reported best practices for generative programming.

Respondents from Activity, Poi, and 101repo reported more than 10 years of software development experience, with educational background of Masters, Bachelor, and PhD respectively. Marble reported 7 years of experience and a bachelor degree. All reported formal software engineering training except Marble who reported self-training exclusively.

None of the respondents reported using models as a primary way to communicate with other developers. Instead, mailing list, chats, and in-person communications were stated.

5.3.2 Responses from non-UML repositories. We included three responses from the non-UML repository pool; specifically, from Fastjson, Mal, and Deeplearning4j. It is possible that other contributors may have been discouraged because the questionnaire included questions on modeling. The respondent for FastJson and Deeplearning4j confirm that there is no use of design or modeling as part of the repository or the development activities. The respondent for Mal reported the use of modeling and designs in other projects, but not in Mal, the repository under study. While there is no overall design or architecture, the respondent reported using code analysis to help in code comprehension. The responded has 12 years of experience, a masters' degree, and self-assessed software engineering skills at 4/5. Notably, the respondent self-assessed code quality as low.

The Mal code repository do not use any UML or modeling, but rather relies on extensive documentation, including architecture specifications and down to step-by-

step guidelines for developers. Inspecting those artifacts reveals that these guidelines focus on coding and aims at instilling consistent styles throughout the code base. Contributions to the code base are often rejected unless these guidelines are strictly applied. The respondent confirms that these documentations are routinely maintained. Mal repository respondent has more than 10 years of software development experience with significant formal training.

5.3.2 Summary of Contributors' Perceptions. All contributors of this interviews confirm the 'UML/non-UML' classification. A notable observation is the Mal code repository where they use extensive documentations that focused on coding styles that did not include any formal modeling or UML artifacts. All responses suggest that code is perceived to be of very high quality in general, except Poi, a UML repository. Profiling information collected about years of experience, education, and formal training do not suggest any imbalance in the data set. All respondents had more than 5 years of development experience. This is at least in part because we selected active and popular code repositories and solicited responses from the most active repository contributors. Two respondents report not having any formal software engineering training, one from UML and one from non-UML.

Table 6: Technical debt results

Repository	TD (Days)	TD Density (Days / LoC)
All UML Repository	1,479	0.65
All Non-UML Repository	2,051	0.73
Gap count	-572	-0.08
Gap %	**-38.67%**	**-12.31%**

5.4 Analysis

We conduct pair-wise mean analysis [11] as follows. We calculate the mean code smell and TD densities for all non-UML repositories. We then compare the non-UML mean to each of the UML repository count of code smell and TD measure. In other words, this analysis compares each of the UML repository quality measures to the average quality measures of non-UML repositories. The results of this analysis are shown in Table 7. Positive numbers in the table indicate values above non-UML repository mean. Similarly, negative numbers indicate values below non-UML repositories mean (shaded cells in the table).

Table 7: Pair-wise analysis

Repository	Code Smells Mean Density				TD Mean
	Critical	Blocker	Major	Minor	
Marble	-0.58	-0.04	11.11	-1.77	0.03
Oryx-editor	-0.38	0.02	-1.11	0.46	-0.06
101repo	-0.17	0.00	1.14	1.33	0.1
Activiti	0.25	0.01	-1.87	-1.26	-0.08
Poi	0.43	0.01	-1.33	-0.31	-0.07

Three out of the 5 UML repositories shows improvement in critical code smell density compared to non-UML repository mean. For TD analysis, three UML repositories TD measure is below non-UML repository TD mean.

6 CODEBASE EVOLUTION

Investigations of code evolution over the five-year period shows a clear pattern where UML repositories started with significantly low measures of TD. However, after these initial low values, code repositories continued to accumulate more TD over the 5-year period. On the other hand, code-centric repositories started with significantly higher TD density and declined overtime.

The interpretation of these results may suggest that software design may have the biggest impact early in the development activities but this impact tends to decline over time, possibly because code contributors do not maintain the models and may focus instead of code contributions. We interpret the phenomenon of TD declination overtime in the case of code-centric repositories is primarily due to the high TD values in early revisions. However, the declination in TD measures was not significant enough to make up for the initial high TD values.

The data from contributors' questionnaires and interviews support these interpretations. UML repository contributors agree with the interpretation that code and model tend to evolve independently and that the contributors and their colleagues do not adequately maintain the synchronization. Contributors also agree that the models tend to become more irrelevant particularly to contributors who have been actively involved in the repository for long time. Models tend to be primarily

useful for new contributors who need to understand the codebase structure.

7 QUALITY CHARACTERISTICS OF SOFTWARE DESIGN

This study assesses the quality of software design using static code analysis tool SonarQube and their potential impact on the collected data. SonarQube has the feature of detecting design code violations to assess the design of a software. In this study, we computed the number and types of design code violations in both UML and Non-UML based repositories. The study suggests that the average design code violations in two UML repositories out of 5 increased. On the other hand, in two Non-UML repositories, it is decreased. The results of design characteristics of selected UML and Non-UML repositories is presented in Table 8. The design code violations related to html and CSS files have been excluded from this study.

It is possible that different qualities of the design models to have different impacts on code quality.

Table 8: Software Design Code Violation

		Design Code Violation Rules			
		Inheritance tree of classes should not be too deep	String literals should not be duplicated	Two branches in a conditional structure should not have exactly the same implementation	Utility classes should not have public constructors
UML	1	0	0	0	0
	2	0	4	7	0
	3	2	368	4	58
	4	0	1162	0	64
	5	5	1271	0	101
Mean		**1.4**	**561**	**2.75**	**44.6**
Non-UML	1'	0	703	3	0
	2'	0	0	11	0
	3'	0	0	7	0
	4'	0	0	0	0
	5'	0	3624	63	0
Mean		**0**	**865.4**	**16.8**	**0**

8 THREATS TO VALIDITY

We discuss the key threats to validity in this study.

8.1 Internal Validity

There is the risk that the model-heavy and code-heavy repositories that were selected for the study have code quality attributes that are affected by factors other than the design and modeling practices in the repository. These factors may include the practices of code reviews and refactorings and the process by which code contributions are reviewed, approved, and committed. Repository contributors experience and training background may also impact code quality. To minimize this risk, we included an external reviewer of the selected repositories to make sure that they are comparable in various aspects, including bug tracking and reviewing process. We understand however that some repositories may include extensive code reviews without it being evident in the repository published information and related artifacts. We also collected profiling information from the most active repository contributors and included it in the analysis.

Because all code repositories were selected from GitHub open source platform, conclusions from this study should be comprehended within the context of open source software.

8.2 External Validity

There is the risk that the selected repositories are not a good representation of the general practices and other open source repositories. This risk is introduced in the selection process. To minimize this risk, we selected repositories of sizes close to the median repository size

in GitHub. We also excluded outlier repositories that are too small or too large in code size, repositories with very few contributors, or repositories that demonstrated unique commit patterns (i.e., repositories with very few commits compared to their code size). Despite this effort, external validity remains.

A second important external validity risk is related to our pool of open source repositories. Open source repositories tend to be developed using an agile approach and are often not from domains where safety and reliability are critical.

8.3 Bias in Code Quality Measurements

There is the risk that our approach to assess code quality is deficient. First, our assessment of code quality is not comprehensive. Similarly, other measures of technical debt may yield different results. Second, we did not consider the domain or nature of the application and its potential effect on code quality and technical debt. To minimize these risks, we considered multiple methods for the same measurements of code smells. Moreover, the conclusions we draw are based on the holistic quality measurements; hence, minimizing the risk that a single measurement may have introduced significant bias.

8.4 Applicability of code smells measures on assessing the quality of the generated code

The study employs well-established measures to assess code quality. The question is raised as to whether such measures are applicable to model-generate code. While the literature is clear that these measures (code smells for example) are a good indication of the quality of the underlying code, it is not clear to what extent the same measures are applicable for the automatically generated code. In our study, we did not find any evidence for generated code. One repository respondent reported generating code from UML models but confirmed that such code is never committed to the repository.

9 CONCLUSION

This study aims at investigating the impact of design activities and UML on code quality and sustainability. In this study, code quality is assessed by measuring 1) code smells, and 2) technical debt. There is, however, potential benefits from using designs and models that go beyond code quality, such as improved code documentation, facilitation of on-boarding of new contributors, and generating other artifacts such as test cases and related documentations. These additional benefits are not investigated in this work.

The results suggest that designs and UML models is likely to have a positive effect on reducing critical code smells and reducing the overall technical debt. The results, however, reports some increase, albeit small, in the number of non-critical code smells. Feedback from contributors seems to confirm these findings. As for codebase evolution, the effects of designs and modeling tend to fade as the codebase continues to evolve, particularly as contributors ignore maintaining software designs and related models.

REFERENCES

[1] Fernández-Sáez, Ana M., Marcela Genero, Danilo Caivano, and Michel RV Chaudron. "Does the level of detail of UML diagrams affect the maintainability of source code?: a family of experiments." *Empirical Software Engineering* 21, no. 1 (2016): 212-259.

[2] Fernandez-Saez, Ana M., Marcela Genero, Michel RV Chaudron, Danilo Caivano, and Isabel Ramos. "Are Forward Designed or Reverse-Engineered UML diagrams more helpful for code maintenance?: A family of experiments." *Information and Software Technology* 57 (2015): 644-663.

[3] Izurieta, Clemente, Gonzalo Rojas, and Isaac Griffith. "Preemptive management of model driven technical debt for improving software quality." In Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, pp. 31-36. ACM, 2015.

[4] Scanniello, Giuseppe, Carmine Gravino, Genoveffa Tortora, Marcela Genero, Michele Risi, José A. Cruz-Lemus, and Gabriella Dodero. "Studying the Effect of UML-Based Models on Source-Code Comprehensibility: Results from a Long-Term Investigation." In International Conference on Product-Focused Software Process Improvement, pp. 311-327. Springer, Cham, 2015.

[5] Kazman, Rick, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, and Andriy Shapochka. "A case study in locating the architectural roots of technical debt." In Proceedings of the 37th International Conference on Software Engineering-Volume 2, pp. 179-188. IEEE Press, 2015.

[6] Ho-Quang, Truong, Regina Hebig, Gregorio Robles, Michel RV Chaudron, and Miguel Angel Fernandez. "Practices and perceptions of UML use in open source projects." In Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, pp. 203-212. IEEE Press, 2017.

[7] Cedrim, Diego, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. "Does refactoring improve software structural quality? a longitudinal study of 25 projects." In Proceedings of the 30th Brazilian Symposium on Software Engineering, pp. 73-82. ACM, 2016.

[8] Al Mamun, Md Abdullah, Christian Berger, and Jörgen Hansson. "Explicating, understanding, and managing technical debt from self-driving miniature car projects." In Managing Technical Debt (MTD), 2014 Sixth International Workshop on, pp. 11-18. IEEE, 2014.

[9] Kazman, Rick, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, and Andriy Shapochka. "A case study in locating the architectural roots of technical debt." In Proceedings of the 37th International Conference on Software Engineering-Volume 2, pp. 179-188. IEEE Press, 2015.

[10] Letouzey, Jean-Louis. "The SQALE method for evaluating technical debt." In Proceedings of the Third International Workshop on Managing Technical Debt, pp. 31-36. IEEE Press, 2012.

[11] Ma, Sheng, Genady Grabarnik, Joseph L. Hellerstein, and Changshing Perng. "Systems and methods for pairwise analysis of event data." U.S. Patent 6,697,802, issued February 24, 2004.

[12] Hoda, Rashina, James Noble, and Stuart Marshall. "Using grounded theory to study the human aspects of software engineering." In Human Aspects of Software Engineering, p. 5. ACM, 2010.

[13] Petre, Marian. "UML in practice." In Proceedings of the 2013 International Conference on Software Engineering, pp. 722-731. IEEE Press, 2013.

[14] Ho-Quang, Truong, Regina Hebig, Gregorio Robles, Michel RV Chaudron, and Miguel Angel Fernandez. "Practices and perceptions of UML use in open source projects." In Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017 IEEE/ACM 39th International Conference on, pp. 203-212. IEEE, 2017.

[15] He, Xiao, Paris Avgeriou, Peng Liang, and Zengyang Li. "Technical debt in MDE: a case study on GMF/EMF-based projects." In Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, pp. 162-172. ACM, 2016.

[16] C. Izurieta, G. Rojas, and I. Griffith. Preemptive management of model driven technical debt for improving software quality. In Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, pages 31–36. ACM, 2015.

[17] Seaman, Carolyn, Robert L. Nord, Philippe Kruchten, and Ipek Ozkaya. "Technical debt: Beyond definition to understanding report on the sixth international workshop on managing technical debt." ACM SIGSOFT Software Engineering Notes 40, no. 2 (2015): 32-34.