Received: 12 February 2020 | Revised: 27 October 2020 | Accepted: 14 January 2021

DOI: 10.1002/spe.2957

RESEARCH ARTICLE

WILEY

The human in model-driven engineering loop: A case study on integrating handwritten code in model-driven engineering repositories

Khandoker Rahad | **Omar Badreddin** | **Sayed Mohsin Reza**

Computer Science, University of Texas at El Paso, El Paso, Texas, USA

Correspondence
Khandoker Rahad, Computer Science, University of Texas at El Paso, El Paso, TX, USA.
Email: rahadiit@gmail.com

Abstract

In model-driven engineering (MDE) software projects, large portions of the executable code are automatically generated from designs and models. This generated code may or may not be edited by the developers to achieve their development objectives. MDE projects also include a significant amount of handwritten code (HC). This handwritten code is developed under unique constraints, as it must integrate with generated artifacts and code elements that are not directly developed by the engineers. These constraints adversely affect codebase quality and maintainability. This case study aims to investigate the hypothesis pertaining to the handwritten code quality developed in the context of MDE. The study analyzes these unique code fragments and compares their characteristics to handwritten code in repositories where code generation is not present. The study finds that handwritten code quality in the MDE context suffers from elevated technical debt and code smells. We observe key code smells that are particularly evident in this handwritten code. These findings imply that code generators must optimize for human comprehension, prioritize extensibility, and must facilitate integration with handwritten code elements.

KEYWORDS

automated software engineering, code analysis, code comprehension, code generation, code smell, forward engineering, handwritten code, model driven engineering, technical debt, UML

1 | INTRODUCTION

Model-driven engineering (MDE) envisions software development teams that focus primarily on developing models that would generate all executable artifacts. This vision seems to have been realized only in organizations that have invested in infrastructures to support domain -specific modeling languages and custom code generators that produce all or most of the required executable. These organizations can afford the overhead to support the development of compilers, code generators, and custom-built design languages. Software modeling is undoubtedly a core activity in software development. The precise form of modeling varies from whiteboard sketches to models that support code generation. Further, modeling in some form is a fundamental part of designing, understanding, communicating, and analyzing software heavy systems.[1]

Today, many MDE practitioners generate only a portion of the required executable artifacts. In these cases, engineers often write code that integrates with and extends the generated code. This handwritten code is unique for many reasons. The code must integrate with generated artifacts that may not be well-suited for integration. Code generators often do not follow coding conventions and frequently generate counter-intuitive code that may not be comprehensible.[2] Moreover, the originating models and their code generators may not be designed to prioritize extensibility; further complicating the engineers' tasks.[3]

In addition to the generated code and the handwritten code categories in MDE projects, developers often modify code that was originally generated from models. This modified code category is also unique; the code is neither written from scratch or purely generated. Software engineers are often constrained in the way they manipulate this code.

The goal of this study is to understand the quality characteristics of handwritten code. Specifically, the study aims to characterize the maintainability of handwritten code fragments in MDE projects. We investigate the hypothesis that handwritten code in MDE contexts suffers from unique deficiencies that have a significant impact on itsmaintainability.

The rest of this article is organized as follows. The next section presents a background pertaining to MDE projects and key related methodologies and technologies. The study design is presented in Section 3. Results and Analysis are presented in Sections 4 and 5. Related works are presented in Section 6. The threats to validity is discussed in Section 7. We conclude this article in Section 8.

2 | BACKGROUND

The potential benefits of MDE are clear; models are much easier to comprehend and provide a better platform to support collaborations. Models tend to be more visual and can support designs at variable levels of abstractions.[4] Moreover, there is significant potential in improving software engineers' productivity and the quality of the code they develop by automatically generating executable artifacts.

Today, only a few organizations have succeeded in achieving this vision. Many MDE adopters generate some artifacts and rely on software developers to extend the generated code. This handwritten code often consumes the majority of the maintenance efforts.[5] As such, understanding this code quality is fundamental to understanding the MDE value proposition.

The handwritten code in MDE projects is subject to unique constraints that can affect code quality both positively and negatively. First, to integrate with generated artifacts is a negative impact of MDE. But on the other hand, having well-formed unambiguous designs that are part of MDE artifacts would affect code quality positively.[6] Therefore, in this study, we analyze the handwritten code in the MDE context with comparable code from two sets of repositories; those that include designs and those that do not. In this study, we collect graphical modeling framework (GMF) and eclipse modeling framework (EMF) based MDE projects because both of these categories are popular, mature, and stable MDE platforms with extensive code generating engines and customized templates.[6] The graphical modeling framework (GMF) is a framework within the Eclipse platform. It provides a generative component and runtime infrastructure for developing graphical editors based on the Eclipse Modeling Framework (EMF).[7] The EMFs purpose is to allow data models to be created and then stored in an "ecore" file. However, GMFs purpose is to translate existing EMF models and utilize GEF (graphical editing framework) to build a graphical editor automatically based on the content.[8] Projects that are developed using GMF/EMF platforms include three unique classes of code. (1) Generated, code that is generated exclusively from models. (2) Generated and modified, code that is generated but then later modified by engineers. (3) Handwritten code, this is code developed manually by engineers that either extend or integrate with the previous two classes of code.

In this study, we hypothesize that code quality characteristics such as code smells (CS) and technical debt (TD) are elevated in the MDE environment. CS is any surface symptom in the source code that suggest deficiencies related to maintainability.[9] CS appear because of bad software design and programming practices and indicate that code refactoring may be required.[10,11] TD is a metaphor that provides short term benefits but may hurt long term software maintainability. TD has both positive and negative impacts on software systems. When TD is incurred intentionally to achieve short-term benefits can be beneficial if the cost associated with TD is made visible and kept under control. However, unintentional TD could be detrimental to the maintenance of the software systems.[12,13]

3 | STUDY DESIGN

The goal of this case study is to investigate the quality characteristics of handwritten code in MDE Projects. We followed Basili et al.[14] guidelines to formulate the goal of the study. Specifically, the case study investigates Technical Debt and Code Smells in MDE handwritten code. Six types of code smell such as large class, large method, excessive imports, god class, cyclomatic complexity, and duplicate code are identified in this study. For reference, we analyze this handwritten code to comparable code fragments from non-MDE repositories. Non-MDE repositories include design-driven (DD) and non-design driven (non-DD) repositories. We followed Runeson et al. guidelines for this case study.[15]

3.1 | Research question

The research is motivated by the following research questions.

RQ1: What are the quality characteristics of handwritten code in the MDE context? How do these characteristics compare to handwritten code in non-MDE contexts?
RQ2: What are the key code deficiencies in handwritten code in MDE projects? What are the most prevalent code smells and their severity?
RQ3: How does the technical debt accumulated in handwritten code in the MDE context compare to non-MDE contexts?

3.2 | Repository selection and artifacts identification

This study identifies 15 sub-systems (sources are listed in Table 1), 5 identified as MDE repositories (based on GMF/EMF framework), and 10 identified as non-MDE repositories that are further classified under two classes. These repositories selection process is visualized in Figure 1. All these 15 repositories can be accessed using the URLs provided in Table 1.

TABLE 1 Repository info

	Project	URL at GitHub
MDE	WSO2 Tools	https://www.github.com/wso2-attic/tools.git
	aspirerfid	https://www.github.com/mouillerart/aspirerfid.git
	pldoctoolkit	https://www.github.com/spbu-se/pldoctoolkit.git
	UNICASE	https://www.github.com/unicase-ls1/unicase.git
	Reuseware	https://www.github.com/DevBoost/Reuseware.git
DD	cdt-tests-runner	https://github.com/xgsa/cdt-tests-runner
	Oryx-editor	https://github.com/andreaswolf/oryx-editor/tree/02e4c0930742137de5f0dcbf604872624ba91bde
	101repo	https://github.com/101companies/101repo
	Activiti	https://github.com/Activiti/Activiti
	Poi	https://github.com/apache/poi/tree/47fb3691f76b6f1286a41faea09019eac843119b
Non-DD	Selenium	https://github.com/SeleniumHQ/selenium
	Fastjson	https://github.com/alibaba/fastjson
	Mal	https://github.com/kanaka/mal
	Deeplearning4j	https://github.com/deeplearning4j/deeplearning4j
	Presto	https://github.com/prestodb/presto?fbclid=IwAR3fa-x1cdM2m62544S65j6Cugix-ubR_AOHsLtBJjI1sr0IyOM3ebYbofY

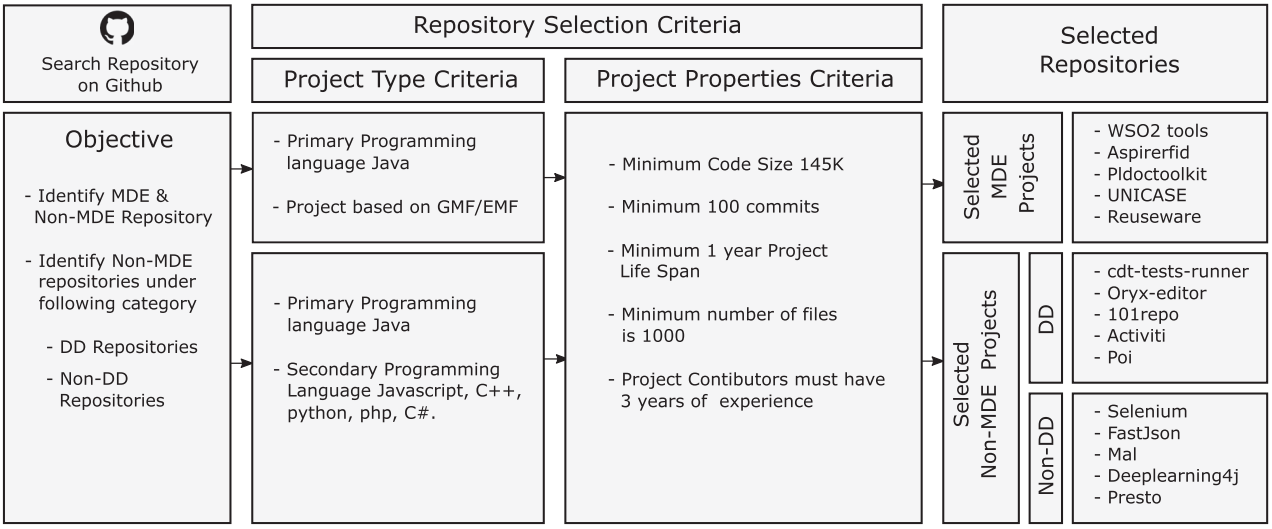

FIGURE 1 Repository selection process

The first five MDE sub-systems are selected from a pool of 16 MDE repositories that are reported in the study by He et al.[6] We select these repositories that meet the following criteria, each repository is GMF/EMF framework based, code size greater than 145k lines of code (LoC), predominantly written in the Java object-oriented programming language, and the number of commits in GitHub is at least 100. We select specific code size and number of commits to exclude trivial projects.

To determine whether a project lies within the GMF/EMF category, we checked whether the project includes files with the extension **gmfgen**. The **gmfgen** extension is the generator model of GMF and from which source code is derived. Since a GMF project may contain many sub-projects, only the sub-projects that are based on GMF/EMF are included in this study. The details of selecting criteria of these five repositories are described in Reference 6. The next five repositories are identified as design-driven sub-systems (DD) which are selected from a pool of 4650 identified in Reference 16 to be model heavy repositories. These 4650 repositories are selected by mining all GitHub repository artifacts that include UML and modeling elements.[17] From this list, we select the top five repositories that meet the following criteria: code size is greater than 145K lines of code, written predominantly in the Java object-oriented language, and have at least 100 commits in the GitHub repository.

The third set of five repositories are selected as reference repositories. These repositories are identified as non-design driven (Non-DD). They are selected from the study by Badreddin et al.[18] These repositories include similar object-oriented code size, number of commits, and similar programming language and contributors' profiles. We ensure that the average expertise of the active contributors in this set is comparable to the expertise of the contributors of the identified repositories. For this, we collect profiling information of active contributors such as the history of their edits, years of contribution in GitHub. Table 3 lists all 15 subject code repositories and the number of their identified files, commits, code size, and analyzed lines of code (LoC). The analyzed LoC column lists the lines of code that were analyzed in this study. This excludes non-object-oriented code and documentation. File category identifies files where code is model generated, modified generated, and handwritten without using any tool.

3.3 | Data collection

MDE repositories contain three types of files: Generated files (GF), modified generated files (MGF), and handwritten code (HC). In this study, we extract handwritten code files from the selected MDE repositories by carefully excluding GF and MGF. This process is achieved by a script[19] where results were independently verified.

3.3.1 | Study variables

For each project, we consider 12 variables that directly relate to our research questions. The variables description and relation with the research question are listed in Table 2. The first six variables ($\#F_{MDE}$, $\#F_{DD}$, $\#F_{NDD}$, $\#LOC_{MDE}$, $\#LOC_{DD}$,

TABLE 2 Variable description

Variables	Description	Research question (RQ)
$\#F_{MDE}$	Total number of handwritten code files in model driven engineering repositories	RQ1
$\#F_{DD}$	Total number of files in design driven repositories	RQ1
$\#F_{NDD}$	Total number of files in non-design driven repositories	RQ1
$\#LOC_{MDE}$	Total number of lines of code in model driven engineering repositories	RQ1, RQ2 and RQ3
$\#LOC_{DD}$	Total number of lines of code in design-driven repositories	RQ1, RQ2, and RQ3
$\#LOC_{NDD}$	Total number of lines of code in non-design driven repositories	RQ1, RQ2 and RQ3
$\#CS_{MDE}$	Total code smells in model driven engineering repositories	RQ1 and RQ2
$\#CS_{DD}$	Total code smells in design-driven repositories	RQ1 and RQ2
$\#CS_{NDD}$	Total code smells in non-design driven repositories	RQ1 and RQ2
$\#TD_{MDE}$	Total technical debt in model driven engineering repositories	RQ3
$\#TD_{DD}$	Total technical debt in design-driven repositories	RQ3
$\#TD_{NDD}$	Total technical debt in non-design driven repositories	RQ3

$\#LOC_{NDD}$) are selected under file and code metrics to compare the relationship between MDE and non-MDE repositories code quality.

The variables ($\#CS_{MDE}$, $\#CS_{DD}$, $\#CS_{NDD}$) represent CS value for each MDE, design-driven (DD), and non-design driven (Non-DD) repositories. These variables provide total occurrences of all CS in handwritten code to help answer the second research question (Table 3).

The last three variables ($\#TD_{MDE}$, $\#TD_{DD}$, $\#TD_{NDD}$) are related to the third research question and refers to TD in the selected repositories in MDE, DD, and non-DD repositories, respectively.

We construct two complex variables related to density for further analysis in this study. These variables are ($\#CSD_{MDE}$, $\#CSD_{DD}$, $\#CSD_{NDD}$, $\#TDD_{MDE}$, $\#TDD_{MDE}$, $\#TDD_{MDE}$) and are described as code smell density (CSD) and technical debt density (TDD) in MDE, DD, and non-DD repositories, respectively. The variables are constructed by using these equations below.

$$\#CSD_X = \frac{\#CS_X}{\#LOC_X} \tag{1}$$

$$\#TDD_X = \frac{\#TD_X}{\#LOC_X} \tag{2}$$

where $\boldsymbol{X}$ represents MDE or DD or non-DD repositories.

3.3.2 | Metrics and thresholds

Metrics and thresholds are uniform for all subject repositories as listed in Table 2 and 5. The Table 2 describes the definitions of all twelve variables that are used in this study. We develop a custom program[19] that can read all the files and folders from the MDE repositories iteratively using the Java program extension and constructs an array of files and directories by filtering *.JAVA* or *.java* extension. To identify handwritten code files from previous filtered results, we determine which files are GF and which files are MGF. We classify the files that do not belong to Generated or Modified Generated as handwritten code files. The classification process of the files is followed by some search criteria which are shown in Table 4. This file search process is performed within MDE repositories.

3.3.3 | Code quality metrics

This section describes CS and TD that asses the code quality of the subject code repositories.

TABLE 3 Basic information of subject software repositories

					No. of files		Analyzed LoC	
	Repository	**Commits**	**Code size**	**File category**	**Count**	**%**	**Count**	**%**
MDE	WSO2 Tools	2, 609	1, 009, 000	GF	3, 025	35.3	311, 422	30.8
				MGF	615	7.2	149, 801	14.8
				HC	4934	57.5	550, 709	54.6
	aspirerfid	341	145, 000	GF	397	25.7	37, 657	25.9
				MGF	55	2.8	3, 124	2.15
				HC	1105	71.5	99, 147	68.4
	pldoctoolkit	493	182, 000	GF	587	58.2	68, 636	37.7
				MGF	102	10.1	14, 537	7.9
				HC	320	31.7	30, 076	16.5
	UNICASE	8, 506	289, 000	GF	3, 202	54.6	406, 819	59.7
				MGF	464	7.9	111, 792	16.4
				HC	2196	37.5	161, 789	23.7
	Reuseware	104	526, 000	GF	4, 193	80.6	598, 755	85.8
				MGF	107	2.1	25, 414	3.6
				HC	903	17.4	73, 912	10.6
DD	Cdt-tests-runner	19, 589	1, 003, 261	HC	8,122		982, 425	97.9
	Oryx-editor	2, 022	640, 127		2,887		543, 704	84.9
	101repo	2312	183, 083		1421		154, 437	84.4
	Activiti	7741	207, 339		3,078		192, 812	93.0
	Poi	9157	450, 906		3,575		427, 326	94.8
Non-DD	Selenium	21, 788	875, 267	HC	4150		775, 268	88.6
	Fastjson	2673	168, 880		2537		149, 186	88.3
	Mal	2249	178, 870		1,567		166, 296	93.0
	Deeplearning4j	9301	283, 711		2062		221, 711	78.1
	Presto	15, 786	716, 021		5632		716, 021	100

TABLE 4 File search criteria

File category	Search criteria
Generated files (GF)	Search in all files by these strings: "@generated," "@Generated"
Modified generated files (MGF)	Search in all files by these strings: "@generated NOT,", "@generated not," "@Generated not," "@Generated NOT"
Handwritten code (HC)	The files that do not belong to generated or generated and modified are considered as handwritten code files.

Code smell: We use PMD,[20] a source code analysis tool to identify code smells. We select six types of CS as listed in Table 5 which include *God Class*, *Excessive Class Length*, *Excessive Method Length*, *Duplicate Code*, *Cyclomatic Complexity*, and *Excessive Imports*. The details of these CS can be found in PMD tool documentation.[20] These CS are selected because they are frequently used in literature[12,21] as TD indicators. For instance, *God Class*, *Duplicate Code*, and *Cyclomatic Complexity* are related to TD, which influence the maintainability of source code.[6]

TABLE 5 Detected types of code smells

No.	Code smell	Threshold
1	Large class	1000 LOC
2	Large method	100 LOC
3	Excessive imports	30 imports
4	God class	N/A
5	Cyclomatic complexity	10
6	Duplicate code	100 duplicated blocks

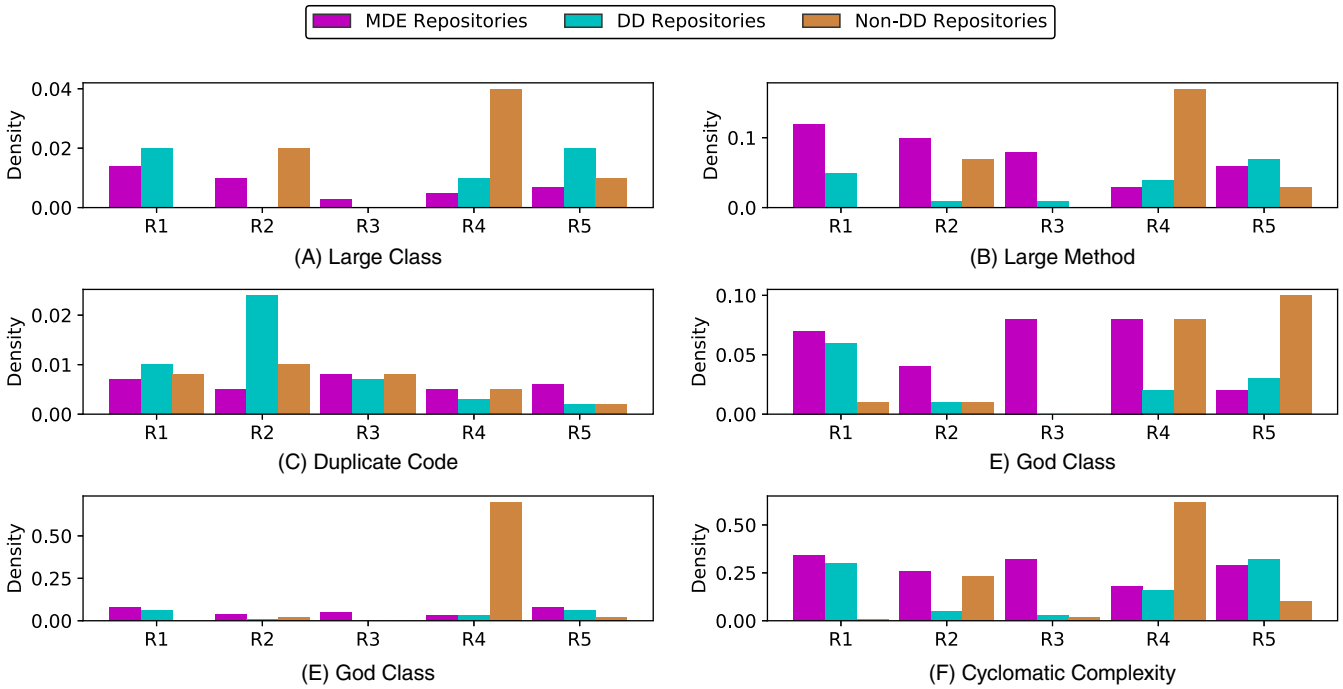

FIGURE 2 Code smells in MDE, DD, and non-DD repositories [Color figure can be viewed at wileyonlinelibrary.com]

In this study, all the CS are measured by the PMD tool except for *Duplicate Code*. Duplicated Code Smell and its density are measured by SonarQube by identifying duplicated block counts of a project divided by physical lines of code. Other CS density are measured by the CS counts divided by analyzed lines of code and multiplied by 100. This density refers to the number of CS per line of code.

Technical debt: TD of subject software repositories are measured using source code analysis tool SonarQube. SonarQube computes TD based on the Software Quality Assessment which is based on Lifecycle Expectations methodology (SQALE).[22] 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. The details of this TD calculation by SonarQube can be found in SonarQube documentation.[23]

We perform similar calculations to measure TD density by dividing the TD counts by analyzed lines of code and multiplying by 100. This density refers to the number of TD per line of code. In other words, the number of TD is the total number of days it will take to fix an issue per line of code.

In Figures 2 and 3, R1, R2 … R5 represents a set of three types of repository that includes MDE, design-driven (DD), and non-design driven (Non-DD), respectively. This repository set selection process for R1, R2 … R5 has been conducted sequentially. For instance, MDE repository *WSO2 Tools* is selected with *Cds-test-Runner* and *Selenium* from DD and Non-DD repository list respectively. These groups (R1, R2..R5) are made because the same group of repositories have similar code size and use objected oriented programming language as a primary language.

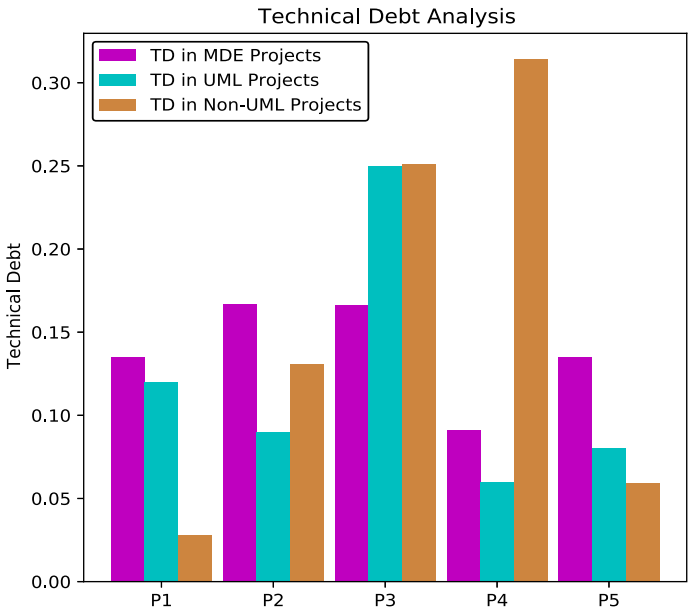

FIGURE 3 Technical debt (TD) result [Color figure can be viewed at wileyonlinelibrary.com]

4 | RESULTS

Our assessment criteria are based on two primary measurements; measurements of code smells and measurements of technical debt. Code smells and technical debt are calculated by the total number of code smells and technical debt (number of days). These measures are normalized by using a density. In the following, we report on these two measurements.

4.1 | Code smell results

Table 6 lists all six types of code smells and technical debt that are measured by code static analysis tool PMD[20] and SonarQube.[23] The total number of code smells in handwritten code in the MDE context are significantly reduced as shown in Table 6. The total number of CS increases with code size metrics in any type of repositories. We found that handwritten code in MDE contexts are associated with reduced CS ($\#CS_{MDE} < \#CS_{DD}$ & $\#CS_{MDE} < \#CS_{NDD}$). Since the number of CS is associated with elevated values when the code size increases, we calculate the frequency of CS for each repository, and we formulate normalized CS metrics as CS density.

Figure 2 illustrates results of six CS density of MDE HC and non-MDE repository code. Figure 2(A–F) illustrates large class, large method, duplicate code, excessive imports, god class, and cyclomatic complexity CS density, respectively. In addition, we report on pairwise comparative analysis of CS in MDE handwritten code and non-MDE repository code.

Figure 2 shows that 60% of the HC from selected MDE repositories have elevated large method, duplicate code, and cyclomatic complexity CS densities. In the case of excessive imports and god class CS densities, 80% of the HC (MDE) have more CS density than non-MDE repository code. However, we observed opposite the results in Large class CS density in 80% of the HC in selected MDE repositories. We also found that all MDE HC associated with elevated CS density on average compared to non-DD repositories (In Figure 4, $\#CSD_{MDE} > \#CSD_{NDD}$). However, MDE CS density is slightly less compare to DD repositories ($\#CSD_{MDE} < \#CSD_{DD}$).

In normalized CS, we found that large methods, excessive imports, cyclomatic complexity are the top three CS that are introduced in MDE HC. However, god class and large class are the least introduced CS in the MDE environment.

4.2 | Technical debt results

Table 6 shows a total number of Technical Debt in MDE handwritten code and non-MDE repository code. We found that total TD in MDE HC is associated with reduced TD. In other words, HC in the MDE environment introduce less TD than

TABLE 6 Code smells and technical debt results

			Code smells							
	Repository	Analyzed LoC	Large class	Large method	Code clone	Excessive imports	God class	Cyclomatic complexity	Total	Technical debt (days)
MDE	WSO2 Tools	550, 709	78	636	5600	371	414	1872	8971	741
	aspirerfid	99, 147	10	103	1000	40	40	257	1450	166
	pldoctoolkit	30, 076	1	24	349	24	15	97	510	50
	UNICASE	161, 789	8	49	1281	122	49	297	1806	148
	Reuseware	73, 912	5	44	794	18	62	217	1140	100
	Total	915, 633	102	856	9024	575	580	2740	13, 877	1205
	Average	183, 127	20	171	1805	115	116	548	2775	241
DD	Cdt-tests-runner	982, 425	150	477	9535	69	319	1619	12, 169	1200
	Oryx-editor	543, 704	16	28	16, 991	30	62	277	17, 404	486
	101repo	154, 437	1	15	2475	0	6	43	2540	386
	Activiti	192, 812	19	77	890	40	66	302	1394	122
	Poi	427, 326	96	311	1628	131	238	1362	3766	322
	Total	2, 300, 704	282	908	31, 519	270	691	3603	37, 273	2516
	Average	460, 141	56	182	6304	54	138	721	7455	503
Non-DD	Selenium	775, 268	3	8	10, 104	71	11	92	10, 289	217
	Fastjson	149, 186	23	103	1955	10	25	341	2457	196
	Mal	166, 296	0	2	3075	0	4	27	3108	415
	Deeplearning4j	221, 711	79	374	2699	184	160	1381	4877	720
	Presto	716, 021	57	181	2536	744	114	693	4325	420
	Total	2, 028, 482	162	668	20, 369	1009	314	2534	25, 056	1968
	Average	405, 696	32	134	4074	202	63	507	5011	394

non-MDE environment code. To normalize the total number of TD in the MDE handwritten code base, we computed TD density.

Figure 3 illustrates TD density results for MDE HC and non-MDE repository code. Overall, 80% of HC from selected MDE repositories have higher TD density than non-MDE repository code.

We also calculate TD elevation between MDE handwritten code bases and non-MDE repositories. There is a 10.2% TD density elevation in all five MDE HC compared to non-DD repository code ($\#TDD_{MDE} > \#TDD_{NDD}$) and TD density increased compared to DD is 6.5% ($\#TDD_{MDE} > \#TDD_{DD}$) in Figure 5.

5 | ANALYSIS

The study results demonstrate that both code smells and technical debt is significantly elevated in handwritten code in MDE repositories. There are smells that were largely unique to this handwritten code, namely, large methods, duplicate code, and excessive imports. Interestingly, this code also had a significantly low number of large class code smells. This suggests that refactoring for large methods would be relatively straightforward. Another key finding is that TD density was incurred in HC code in the MDE context (Figures 3 and 5). Based on our sample, we found evidence that designs by themselves tend to reduce TD compare to MDE HC TD, as evident in the TD density for design-driven repositories. Further, this would suggest that the elevated TD counts in MDE repositories are largely due to the unique constraints that

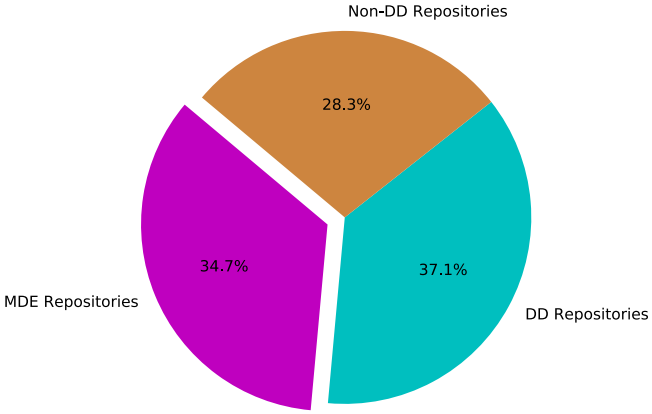

FIGURE 4 Average code smell density results [Color figure can be viewed at wileyonlinelibrary.com]

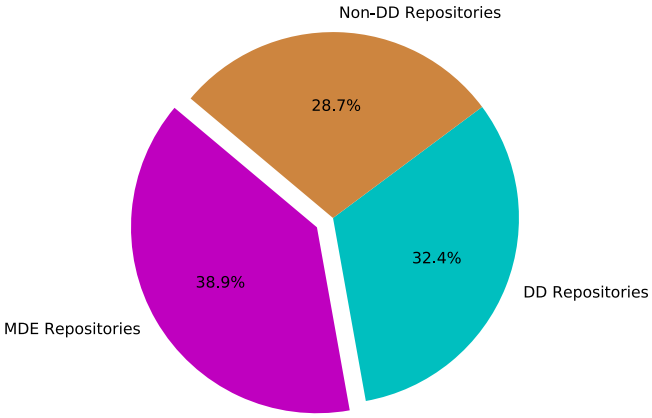

FIGURE 5 Average technical debt density results [Color figure can be viewed at wileyonlinelibrary.com]

software engineer face in developing this code. Overall, this confirms the hypothesis that handwritten code in the MDE context is subject to unique constraints that adversely affect its quality and sustainability. This suggests that handwritten code requires more attention and maintenance. Robust modeling tool can make the task easier integrating handwritten code with model-generated code. For example, code generation rules require to redefining to reduce code smells and TD. Moreover, avoiding edits on tool generated code can reduce TD and code smell as well. In addition, the model sent to the code generator can be defective and the modeling tool generates source code precisely according to input models and it will convert defective models into defective code. To minimize this issue customized modeling tool can be applied to MDE which is also proposed in Reference 24.

5.1 | RQ1: Quality characteristics of handwritten code in MDE context

The first research question investigates the code quality characteristics. For that, we found that code smell density and TD density are elevated in HC in MDE repositories. For example, large method code smell is elevated in three triplets (R1,R2,R3) out of five compared to DD and non-DD repositories in Figure 2(B). In Figure 2(D), excessive imports code smells shows the same results. We also observe that Cyclomatic complexity to be elevated in HC MDE repositories (Figure 2(F)). Moreover, Figure 4 depicts average code smell density in all MDE repositories is maximum compare to DD and non-DD repositories. Thus, we answer our first research question which is related to code characteristics of handwritten code in the MDE context.

5.2 | RQ2: Code deficiencies in handwritten code in MDE projects

The second research question focuses on investigating unique deficiencies in the handwritten code in MDE contexts. This study finds that large method code smell density is the highest overall in HC code, followed by duplicate code and

excessive imports in Figure 2. Large method code smells are often associated with Large class smells, but this was not the case in this study. This suggests that classes in the HC MDE context have few numbers of methods but with a significantly large number of lines of code within each method. This is potentially due to how these methods grow over time, or how these methods extend and/or integrate with generated artifacts. Therefore, there are specific types of code smells that are increased in HC in the MDE context and we answer our second research question RQ2.

5.3 | RQ3: Technical debt accumulated in handwritten code in MDE context

The third research question investigates technical debt measures. Often, TD follows code smells as is the case in this study. In Figure 3, TD count and density are elevated in HC code in MDE contexts in all four subject MDE repositories. Further, Figure 5 shows higher technical debt density in HC (MDE) compare to DD and non-DD repositories. The intuition is that HC in the MDE context are not well maintained and these triggers incurred TD.

6 | RELATED WORKS

There is very limited literature on code quality within the context of MDE environment. Hutchinson et al.[25] conducted an empirical study of MDE projects in the context of industry by questionnaire and interviews. They reported on the understanding of social and organizational factors on MDE usages and investigated factors of failure and success aspects of MDE such as benefits of code generation. Moreover, Fernandez-Saez et al.[26] conducted interviews and questionnaire on UML and software modeling with employees of a software company who works on software maintenance projects. The results of this survey suggest that UML modeling is beneficial; however, there are concerns about integrating modeling into the overall software engineering approach. These investigations are based on interviews and questionnaires on industry MDE software. However, our study investigates characteristics of handwritten code (HC) in MDE and non-MDE software sub-systems from open-source GitHub.

In 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 than what software developers would normally produce.[6] This study by He et al. observed there seems to be limited literature on TD in the context of modeling and MDE.[27] Izurieta et al. discussed in a position paper basic concepts of TD in the context of MDE but did not provide code-level analysis.[28]

Nurgoho and Chaudron[29] analyzed the impacts of UML modeling (class diagram and sequence diagram) in terms of defect density with software modules that are not modeled and found that UML modeling reduces the defect density in code than not modeled modules.

Consequently, Lucredio et al.[5] conducted three case studies and measured the impact of MDE on software reuse. They compared software systems that are developed in MDE and non-MDE environments. The results of this investigation suggest that in some domains such as in the business domain MDE environment improves re-usability. However, the MDE environment software development approach is associated with some maintenance costs.

Mohagheghi and Dehle[30] reported on MDE applications in the industry; their studies aims at investigating the impact of MDE on software quality. They conducted a literature review of 25 papers and found very few empirical data that focuses on the quality of software that is developed in the MDE environment. In their report, they found when MDE is applied in software developed it advocates the productivity and the quality of the software system.

Arisholm et al.[31] carried out controlled experiments to assess the impacts of using UML documentation in software maintenance. The results of this study suggest that UML documentation does help to save effort in terms of time. Moreover, the authors found that UML documentation has positive impacts on the most complex tasks. Dzidek et al.[32] conducted similar controlled experiments and observed UML provides benefits in terms of correctness, time, and software quality.

However, model and code synchronization is reported as the most fragile and time-consuming activity (see Forward and Lethbridge[4] and Thorn and Gustafsson[33]). They found that software models are used as means of communication and collaboration among team members.

7 | THREATS TO VALIDITY

There are some threats to validity in this study that we categorize as construct threats to validity and external threats to validity. Constructs threats refer to threats to which the study measures what it claims to be measuring. However, external threats refer to whether we can generalize this study with different settings. This study does not deal with internal threats to validity and we exclude it.

7.1 | Construct validity

The selected types of CS are a subset of all the CS that are predominantly found in the code-base and indicate maintenance needed in the code-base. We do not claim that the selected types of CS are a complete set for TD. Furthermore, we do not claim that other CS that are not included in this study can not be TD indicators. However, it is an open question to investigate which code smells are more suitable than others as TD indicators. In the future, we plan to repeat this study with other code smells.

The second threat of this study is the precision of measuring CS by PMD and SonarQube tool. We do not claim that PMD and SonarQube are the best tools to measure CS. There are many source code analysis tools out there that can measure CS. We use SonarQube and PMD because these are the most popular and standard source code analysis tools.[34,35] We plan to minimize this threat by using multiple code analysis tools and synthesizing the results.

The third threat is artifacts identification in MDE and non-MDE projects which verifies whether a selected project is MDE or non-MDE. We conducted a semi-automated process to identify MDE elements in the codebase. We do not claim that our identification process is the most appropriate one. This threat can be minimized by regenerating the code from models and compare the current version with the regenerated version. However, this requires a lot of effort that can not be spent in this explanatory study.

The fourth threat is the comparison of TD in handwritten code (MDE environment) with TD in non-MDE code whether reasonable or not. We argue that this is reasonable to compare because we compare non model generated code quality with the non-MDE code. This is comparable since the coding is similar in terms of codebases which are handwritten.

7.2 | External validity

There is the risk that the selected 15 repositories are not the best 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 repositories that are trivial. We defined trivial repositories that have less than 100 commits and code size is less than 145k.

The second external threat of this study is identifying MDE projects considering the GMF/EMF framework. There are some other modeling frameworks such as Xtext that also can generate executable code from the model. We use GMF/EMF because these are the most popular modeling frameworks in the MDE context.[36]

The third type of threat is identifying comparable repositories by their code size, the number of commits in GitHub, and primary programming language. We do not argue that these criteria are the best criteria to select comparable MDE and non-MDE repositories. In the future, we plan to include other criteria such as software domain and software technology to identify comparable repositories.

The fourth external threat is we undermine the importance of assessing some confounding factors such as developers expertise. Assessing these types of confounding factors will not give us a perfect perception of MDE environment, but may provide us the developer's circumstances under which the MDE environment could be beneficial.

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

8 | CONCLUSION

The study analyzes the code quality characteristics in MDE repositories. We investigate the handwritten code in MDE projects and compare their code quality with non-MDE environment codebases. The handwritten code in the MDE

context is unique because it must integrate with and extend code that is automatically generated from models. The results of this study suggest that the handwritten code developed in MDE environments suffers from elevated levels of code smells and technical debt. This HC shows degraded quality in terms of poor design. The study found that there are more code smells, such as god class, excessive imports, large method, and cyclomatic complexity, that are more prevalent in handwritten code in MDE repositories. In addition, measures of technical debt were also elevated in this code.

In this study, we reported key code smells that tend to be more prevalent in this unique handwritten code; namely, large method, excessive imports, and duplicate code smell. We attribute this to the constraints that are unique to the handwritten code in MDE projects. These constraints include integrated and extended generated artifacts. MDE repositories often use code generators that may produce code that is not intuitive or comprehensible. Such factors, among others, contribute to the degraded code quality. And since this handwritten code tends to consume a significant portion of the maintenance effort, its degraded quality may cancel out or overshadow the benefits of automated code generation.

This study highlights the need to optimize code generators for human comprehension, and to prioritize generating modular extensible code.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Khandoker Rahad https://orcid.org/0000-0001-7481-8954
Omar Badreddin https://orcid.org/0000-0002-8851-4131
Sayed Mohsin Reza https://orcid.org/0000-0003-3379-6319

REFERENCES

1. Whittle J, Hutchinson J, Rouncefield M. The state of practice in model-driven engineering. *IEEE Softw*. 2013;31(3):79-85.
2. Schmidt DC. Model-driven engineering. *Comput-IEEE Comput Soc*. 2006;39(2):25.
3. Badreddin O, Khandoker R, Forward A, Masmali O, Lethbridge TC. A Decade of software design and modeling: a survey to uncover trends of the practice. Paper presented at: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems; 2018:245-255; ACM, New York, NY.
4. Forward A, Badreddin O, Lethbridge TC. Perceptions of software modeling: a survey of software practitioners. Paper presented at: Proceedings of the 5th Workshop from Code Centric to Model Centric: Evaluating the Effectiveness of MDD (C2M: EEMDD), San Francisco, CA; 2010.
5. Lucredio D, Almeida DES, Fortes RP. An investigation on the impact of mde on software reuse. Paper presented at: Proceedings of the 2012 6th Brazilian Symposium on Software Components, Architectures and Reuse, Natal, Brazil; 2012:101-110; IEEE.
6. He X, Avgeriou P, Liang P, Li Z. Technical debt in MDE: a case study on GMF/EMF-based projects. Paper presented at: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems; 2016:162-172; ACM, New York, NY.
7. Herrmannsdoerfer M, Ratiu D, Wachsmuth G. Language evolution in practice: the history of GMF. Paper presented at: Proceedings of the International Conference on Software Language Engineering; 2009:3-22; Springer, New York, NY.
8. Eclipse Foundation. Graphical Modeling framework/Eclipse Modeling Framework. 2019. https://www.eclipse.org/articles/article.php?file=Article-Integrating-EMF-GMF-Editors/index.html. [Accessed 4th Oct 2019].
9. Cedrim D, Sousa L, Garcia A, Gheyi R. Does refactoring improve software structural quality? a longitudinal study of 25 projects. Paper presented at: Proceedings of the 30th Brazilian Symposium on Software Engineering; 2016:73-82; ACM, New York, NY.
10. Van Emden E, Moonen L. Java quality assurance by detecting code smells. Paper presented at: Proceedings of the 9th Working Conference on Reverse Engineering, Richmond, VA; 2002:97-106; IEEE.
11. Zazworka N, Izurieta C, Wong S, et al. Comparing four approaches for technical debt identification. *Softw Qual J*. 2014;22(3):403-426.
12. Brown N, Cai Y, Guo Y, et al. Managing technical debt in software-reliant systems. Paper presented at: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research; 2010:47-52; ACM, New York, NY.
13. Kazman R, Cai Y, Mo R, et al. A case study in locating the architectural roots of technical debt. Paper presented at: Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy; 2015:179-188.
14. Basili VR. Software modeling and measurement: the Goal/Question/Metric paradigm. Technical Report; 1992.
15. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. *Empir Softw Eng*. 2009;14(2):131.
16. Ho-Quang T, Hebig R, Robles G, Chaudron MR, Fernandez MA. Practices and perceptions of UML use in open source projects. Paper presented at: Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP); 2017:203-212; IEEE.

17. Gousios G, Spinellis D. GHTorrent: GitHub's data from a firehose. Paper presented at: Proceedings of the 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), Zurich, Switzerland; 2012:12-21; IEEE.
18. Baddreddin O, Rahad K. The impact of design and UML modeling on codebase quality and sustainability. Paper presented at: Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, Toronto, Canada; 2018:236-244; IBM Corp.
19. Eclipse Foundation. File classification GMF/EMF. 2019. https://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_1. [Accessed 1st Oct 2019].
20. PMD Documentation. Static Code Analysis Tool. 2019. https://pmd.github.io/latest/index.html. [Accessed 27th Mar 2019].
21. Marinescu R. Assessing technical debt by identifying design flaws in software systems. *IBM J Res Develop*. 2012;56(5):9:1-9:13.
22. Letouzey JL. The SQALE method for evaluating technical debt. Paper presented at: Proceedings of the 2012 3rd International Workshop on Managing Technical Debt (MTD), Zurich, Switzerland; 2012:31-36; IEEE.
23. SonarQube Documentation. Static Code Analysis Tool. 2019. https://docs.sonarqube.org/latest/. [Accessed 27th Mar 2019].
24. Giraldo FD, Espana S, Pineda MA, Giraldo WJ, Pastor O. Conciliating model-driven engineering with technical debt using a quality framework. Paper presented at: Proceedings of the International Conference on Advanced Information Systems Engineering; 2014:199-214; Springer, New York, NY.
25. Hutchinson J, Whittle J, Rouncefield M, Kristoffersen S. Empirical assessment of MDE in industry. Paper presented at: Proceedings of the 33rd International Conference on Software Engineering; 2011:471-480; ACM, New York, NY.
26. Fernández-Sáez AM, Chaudron MR, Genero M. An industrial case study on the use of UML in software maintenance and its perceived benefits and hurdles. *Empir Softw Eng*. 2018;23(6):3281-3345.
27. Seaman C, Nord RL, Kruchten P, Ozkaya I. Technical debt: beyond definition to understanding report on the sixth international workshop on managing technical debt. *ACM SIGSOFT Softw Eng Notes*. 2015;40(2):32-34.
28. Izurieta C, Rojas G, Griffith I. Preemptive management of model driven technical debt for improving software quality. Paper presented at: Proceedings of the 2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA), Montreal, QC, Canada; 2015:31-36; IEEE.
29. Nugroho A, Chaudron MR. The impact of UML modeling on defect density and defect resolution time in a proprietary system. *Empir Softw Eng*. 2014;19(4):926-954.
30. Mohagheghi P, Dehlen V. Where is the proof?-a review of experiences from applying MDE in industry. Paper presented at: Proceedings of the European Conference on Model Driven Architecture-Foundations and Applications; 2008:432-443; Springer, New York, NY.
31. Arisholm E, Briand LC, Hove SE, Labiche Y. The impact of UML documentation on software maintenance: an experimental evaluation. *IEEE Trans Softw Eng*. 2006;32(6):365-381.
32. Dzidek WJ, Arisholm E, Briand LC. A realistic empirical evaluation of the costs and benefits of UML in software maintenance. *IEEE Trans Softw Eng*. 2008;34(3):407-432.
33. Thörn C, Gustafsson T. Uptake of modeling practices in SMES: initial results from an industrial survey. Paper presented at: Proceedings of the 2008 International Workshop on Models in Software Engineering; 2008:21-26; ACM, New York, NY.
34. Louridas P. Static code analysis. *IEEE Softw*. 2006;23(4):58-61.
35. Lenarduzzi V, Sillitti A, Taibi D. A survey on code analysis tools for software maintenance prediction. Paper presented at: Proceedings of the International Conference in Software Engineering for Defence Applications; 2018:165-175; Springer, New York, NY.
36. Evans A, Fernández MA, Mohagheghi P. Experiences of developing a network modeling tool using the eclipse environment. Paper presented at: Proceedings of the European Conference on Model Driven Architecture-Foundations and Applications; 2009:301-312; Springer, New York, NY.

How to cite this article: Rahad K, Badreddin O, Mohsin Reza S. The human in model-driven engineering loop: A case study on integrating handwritten code in model-driven engineering repositories. *Softw Pract Exper*. 2021;1–14. https://doi.org/10.1002/spe.2957