Assessing Code Smell Interest Probability: A Case Study

Conference Paper · May 2017

CITATIONS
0

READS
4

4 authors:

Sofia Charalampidou
University of Groningen
12 PUBLICATIONS **50** CITATIONS
SEE PROFILE

Apostolos Ampatzoglou
University of Groningen
53 PUBLICATIONS **247** CITATIONS
SEE PROFILE

Alexander Chatzigeorgiou
University of Macedonia
160 PUBLICATIONS **1,498** CITATIONS
SEE PROFILE

Paris Avgeriou
University of Groningen
235 PUBLICATIONS **2,588** CITATIONS
SEE PROFILE

Assessing Code Smell Interest Probability: A Case Study

Sofia Charalampidou
Department of Mathematics and Computer Science
University of Groningen
Groningen, The Netherlands
s.charalampidou @rug.nl

Apostolos Ampatzoglou
Department of Mathematics and Computer Science
University of Groningen
Groningen, The Netherlands
a.ampatzoglou@rug.nl

Alexander Chatzigeorgiou
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
achat@uom.gr

Paris Avgeriou
Department of Mathematics and Computer Science
University of Groningen
Groningen, The Netherlands
paris@cs.rug.nl

ABSTRACT

An important parameter in deciding to eliminate technical debt (TD) is the probability of a module to generate interest along software evolution. In this study, we explore code smells, which according to practitioners are the most commonly occurring type of TD in industry, by assessing the associated interest probability. As a proxy of smell interest probability we use the frequency of smell occurrences and the change proneness of the modules in which they are identified. To achieve this goal we present a case study on 47,751 methods extracted from two well-known open source projects. The results of the case study suggest that: (a) modules in which "code smells" are concentrated are more change-prone than smell-free modules, (b) there are specific types of "code smells" that are concentrated in the most change-prone modules, and (c) interest probability of code clones seems to be higher than the other two examined code smells. These results can be useful for both researchers and practitioners, in the sense that the former can focus their research on resolving "code smells" that produce more interest, and the latter can improve accordingly the prioritization of their repayment strategy and their training.

CCS CONCEPTS

• **Software and its engineering** → Software creation and management → Software development techniques → *Object-oriented development* • **Software and its engineering** → Software creation and management → Software verification and validation → *Empirical software validation* • **Software and its engineering** → Software creation and management → Extra-functional properties

KEYWORDS

Change proneness; interest probability; technical debt; case study

MTD 2017, May 22, 2017, Cologne, Germany
© 2017 ACM. ISBN 978-1-4503-4486-9/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3019612.3019781

1. INTRODUCTION

Technical Debt Items (TDIs) [22] are different types of artifacts, like modules, design decisions, or requirements that suffer from technical debt. According to industrial experience [12], completely eliminating technical debt from all TDIs is unrealistic and sometimes undesirable. Particularly, technical debt that is concentrated on TDIs that are not being maintained, will not produce any interest in future maintenance activities. Therefore, spending effort on repaying technical debt from such TDIs will not be cost-effective. To quantify this varying need for repayment, Seaman et al. [21] have introduced the term ***interest probability***, which represents the probability of a TDI to produce interest. Therefore, interest probability is of great importance in the process of technical debt management as it helps to prioritize which technical debt to repay.

In this study, we focus on code TD, which is the most relevant type of TD in industry [2]. In the case of code TD, when assessing the interest probability of a module, we need to evaluate its change proneness, i.e., the probability for this module to change in the future. This includes all possible types of changes: changes in requirements, changes from bug fixing, and changes due to ripple effect [3], [7]. The most usual proxy of module change proneness is its change frequency in past versions (i.e., system history) [6]. The most common way to identify code TD is to detect the existence of code smell occurrences [1].

The goal of this study is to assess the interest probability incurred by specific code smells. Conceptually, interest probability for a smell X represents the probability that at least one module of the system (***that contains an occurrence of smell X***) will change in the next version of the system. For example interest probability$_{\text{smell X}}$ = 0.5 suggests that there is a 50% chance that at least one module suffering from smell X will change in the next version of the system. This offers awareness of which code smells are more probable to generate interest along maintenance and can thus help to: (a) prioritize the refactoring of the most risky smells; and (b) train staff accordingly, in order to prevent their future introduction into software systems. The interest probability of a code smell in a specific system can be calculated by considering: (a) the occurrence frequency of the investigated code smells, and (b) the change proneness of modules in which code smells reside, and

then we will synthesize these results to calculate interest probability (for more details see Section 3.4). The code smells that we will investigate in this study are detectable at the method level. The selection of the studied smells along with their selection process is thoroughly discussed in Section 3. In order to achieve the aforementioned goal, we have performed a case study on 47,751 methods, extracted from two open source projects, namely *Spring Framework* and *AndEngine*. More details on these projects can be found in Section 4.2.

The rest of the paper is organized as follows: In Section 2, we present related work and in Section 3, we present the studied types of technical debt and the tools that we have used for their identification. In Section 4, we present the case study design, whose results are presented and briefly discussed in Section 5. Further discussion concerning the implications to researchers and practitioners is presented in Section 6. Threats to validity are presented in Section 7, and finally the paper is concluded in Section 8.

2. RELATED WORK

As related work to this study we consider papers that investigate the frequency of "code smells" [13] and the change proneness of the modules in which smells are identified. Specifically we present: (a) studies investigating the frequency of code smell occurrences and studies exploring the frequency of applying refactorings; (b) studies on the impact of code smells on change proneness, and (c) the contribution of this paper to the technical debt field.

Code Smells / Refactoring Application Frequency: In order to investigate the occurrences of bad smells in real projects, Chatzigeorgiou and Manakos conducted a case study to investigate the presence and evolution of four types of code smells (i.e., Long Method, Feature Envy, State Checking and God Class) using two Open Source Systems [9]. According to the results of the study, the existence of long methods, (i.e. methods of large size, which have semantic distance between what the method is supposed to do and how it does it), is the most common smell.

Murphy-Hill et al. [15] conducted an extensive study on the application of refactorings, using four data sets and gathering data from 3,400 version control commits. The findings of the study showed that refactoring activity is often not reported in commit logs as assumed in the past, while refactoring tasks are often blended within other programming changes. Additionally, refactoring identification from version systems has been performed by Ratzinger et al. [17]. Here the authors analyzed five open-source projects to investigate the relation between refactorings and the probability of future software defects. To achieve this goal the authors have analyzed project commit messages and extracted the required information.

Code Smells and Change proneness: Olbrich et al. [16] investigated the impact of two code smells (God Class and Shotgun Surgery) on change-proneness, by analyzing the historical data of two open-source projects. According to their results, there are different phases in the evolution of code smells during the system development affecting the change proneness of the components that suffer by code smells. However, it was observed that the classes infected by the examined smells suffer more changes than the non-infected ones.

Similarly, Khomh et al. [14] studied the impact of classes with code smells on change-proneness, by analyzing two open-source projects, and additionally investigated fault-proneness, as well as particular kinds of changes occurring on classes participating in certain anti-patterns. The results indicated that the likelihood for classes with code smells to change is in general very high, but having some combinations of code smells can result to classes which are more difficult to change and thus, are less change-prone than others.

Contribution: To the best of our knowledge this is the first study that investigates the relationship of change proneness and the existence of code smells in the context of technical debt management. This different perspective (TDM instead of smell occurrence or change frequency) provides a contextual meaning to our findings. This is an important contribution, in the sense that it enables us to discuss the results in a way that they can be directly exploitable by the technical debt community. Finally, to the best of our knowledge this is the first study that focuses on the specific code smells (detectable in the method body), which enables us to perform a more concrete interpretation of the results that we have obtained.

3. TYPES OF TECHNICAL DEBT AND IDENTIFICATION TOOLS

In this section of the paper we discuss the code smells that we investigate in this study. Upon the selection of these smells, we will present the tools that can be used for their occurrence identification, and those selected for this study.

The most popular catalogue of source code smells is the one presented by Fowler and Kent [13] in the seminal book on software refactorings. According to Fowler, each refactoring can be mapped to a "code smell", which represents a symptom of "bad" design or implementation. The book presents 22 code smells, which we categorized based on their scope as follows: (a) smells spreading across classes (e.g., *feature envy*), (b) smells spreading across multiple methods (e.g., *message chain*), (c) smells related to the interplay between method and attributes of the same class (e.g., *god class*), and (d) smells that are focused on the body of specific methods (e.g., *long method*).

In this study, we have selected to investigate code smells that are limited to the body of a single method (i.e. the fourth category). Although, we do not imply that the rest of the smells are less important, or that they are not detectable at the source code level, the fact that they can be detected by design-level artifacts (e.g., class or sequence diagrams) make them more ambiguous to categorize between code or design smells; our scope is clearly on code smells. The special case of the *comments* smell (that belongs to the fourth category) has not been considered since: (a) their categorization as superfluous or useful would require the processing of textual information, and (b) the mapping between comments and the methods that they correspond to could not always be au-

tomated without the manual inspection of the code[1]. Therefore, we have selected to investigate three smells:

- ***Long method***. The long method code smell exists when a method is large in size and holds many responsibilities. This smell can be resolved by extracting smaller methods from the long one, so that each one conforms to the single responsibility principle [13].
- ***Conditional Complexity***. The problem with conditional statements (i.e., if or switch) that perform type checking is essentially that of duplication. The object-oriented notion of polymorphism provides an elegant way to deal with this problem [13].
- ***Code clones*** exist when the same code structure is identified in more than one places of the code base. The existence of this smell hinders maintainability and testability, and it can be resolved by applying the extract method refactoring [13].

To identify methods suffering from the aforementioned code smells of interest, we used three existing tools (using the default configuration), which parse Java code, are available online, and whose accuracy has been evaluated in previous studies. Regarding ***the identification of long methods*** we used the SEMI tool [10][11], which is a standalone tool that calculates the need for a method to be refactored and proposes potential extract method opportunities, ranked based on an estimate of their fitness for extraction. The evaluation of the approach, conducted on both open source and industrial data, suggested that SEMI was more accurate than other existing tools serving similar goals (i.e. JDeodorant (Long Method Detector) [24] and JExtract [23])

Concerning the ***identification of conditional complexity*** occurrences, which corresponds to the lack of polymorphism, we used JDeodorant (Type Checking Detector) [25]. JDeodorant is an Eclipse plugin that provides a recommender on refactoring opportunities that facilitates the use of polymorphism, through type checking. To the best of our knowledge, no other tools exist on identifying the corresponding smell. Thus, we did not have the option to choose among multiple tools. However, in the original introduction of the JDeodorant methodology, the tool has been evaluated in three ways: first, in terms of precision and recall, showing moderate precision and relatively high recall scores; second according to experts' opinion about the importance of the identified refactoring opportunities; and third in terms of scalability for analyzing large projects.

Finally, to ***identify instances of the duplicate code smell***, we used NiCad [19]. NiCad is a command line tool that provides as output sets of lines of code that have been duplicated in the source code. Due to the large number of tools that are able to extract duplicate code statements, we based our selection on the results of an independent study that compared 42 clone detection tools and approaches on 4 different scenarios [18]. By considering the point system proposed in the paper, we came up with NiCad as the most prominent tool for the identification of duplicate code.

[1] Although in some cases comments might reside in a method's body, and thus the mapping is evident, we believe that the accuracy of the dataset would be threatened by the amount of false-negatives, i.e., comments that refer to a specific method, but are located outside of its body.

4. CASE STUDY DESIGN

The case study presented in this paper, has been designed and is reported according to the linear-analytic structure template suggested by Runeson et al. [20]. In particular, in the upcoming sections we present the: (a) research objectives and the corresponding research questions, (b) case and subjects selection process, (c) data collection procedure, and (d) data analysis process.

4.1 Objectives and Research Questions

The of this case study in terms of Goal Question Metric (GQM) [8] is formulated as follows: **"*analyze*** code smells ***with the purpose of*** evaluation, ***with respect to*** their interest probability (based on their frequency of occurrence and the change proneness of modules in which they are identified), ***from the point of view of*** *software engineers*, ***in the context of*** technical debt management**"**. This leads to the following main research question (RQ): ***What is the interest probability incurred by code smells****?* To answer this research question we will first investigate the following sub-questions:

RQ_1: *What is the occurrence frequency for each code smell*?

This research question will aim at identifying the most commonly occurring code smells at the method level. The more occurrences of a code smell exist in the code-base, the more probable it is for the software engineers to face interest, due to the existence of the specific smell, while maintaining the software.

RQ_2: *What is the mean change proneness of the modules in which each type of code smell is identified?*

This research question will explore whether the identified methods suffering from code smells tend to change frequently, increasing the chances of producing interest. To answer the research question we will report on the average change proneness of modules that suffer from each code smell.

We note that the answer to the main question will be provided after the answer to RQ_1 and RQ_2, since the calculation of smell interest probability requires a synthesis of the information gathered when answering the two sub-questions.

4.2 Case Selection and units of analysis

Our study is an embedded multiple case study that has been conducted on Java open source code. In this study as cases we consider the different projects, whereas as units of analysis we consider their methods. The reason for restricting our case selection to Java projects was a limitation of the tools used for identifying code smell occurrences. The two open source projects that we used in our study, and the rationale of their selection are presented below:

- **Spring** is a framework that provides a comprehensive programming and configuration model for modern Java-based enterprise applications on any kind of deployment platform. Spring is a very successful project with more than 100 releases and 14,000 commits and it can be considered as a piece of software of good quality since it adheres to well-known principles and patterns. From Spring Framework, we have extracted 5,284 classes that offer us 44,746 units of analysis (i.e., methods).
- **AndEngine** is a successful engine for developing Android games. AndEngine holds a substantial history with more

than 1,800 commits and offered us 459 classes with 3,005 methods. The rationale of selecting a game engine as our second subject was our motivation not to focus this study only on "good" quality software. Thus, we selected a project from the application domain of computer games, which according to existing literature often lacks in terms of structural quality [4].

4.3 Data Collection

The data collection process can be divided into two main parts: (a) the identification of code smell occurrences, and (b) the assessment of method change proneness. The identification of code smell occurrences has been performed with the tools that have been presented in Section 3, namely: *SEMI*, *JDeodorant*, and *NiCad*. The assessment of change proneness has been performed through a rather simple metric, named *Percentage of Commits in which a Method has Changed (PCMC)*, calculated through a tool that has been developed by Arvanitou et al. [5]. On the completion of data collection the following variables have been recorded for every unit of analysis (i.e. method):

[V1] ***Method name***: The name of the considered method

[V2] ***Class name***: The class in which the method belongs to

[V3] ***Long Method:*** Is the method classified as long by the SEMI tool (yes / no)?

[V4] ***Code Clone:*** Number of clones identified in the method's body by NiCad

[V5] ***Conditional Complexity:*** Number of conditional statements in the method that have been flagged as unnecessary (i.e., they can be replaced with polymorphism) by Deodorant

[V6] ***PCMC***: Percentage of commits in which the method has changed. The complete history of the method is considered.

4.4 Data Analysis

In order to answer the research questions set in Section 4.1, we will statistically analyze the collected data, through descriptive statistics and hypothesis testing.

To answer **RQ_1** we will use frequency tables and a heatmap as a means of visualization. To assess the occurrence frequency of each smell we will use: (a) the actual values for comparison among types of code smells, and (b) the occurrences per mile (‰) to check the reliability of our findings across the two projects. We note that for answering RQ_1, the number of occurrences of the same smell in the same method is irrelevant, because even the existence of one smell type in a method, would generate interest upon the method's change. To avoid confusion, we note that in methods that involve multiple smells, the interest amount would increase, in the sense that the effort required to maintain the code would be higher. However interest probability would remain the same independently of the number of smells. Thus, the existence of any number of smells should be equally counted as a reason for increasing interest probability. Therefore regarding [V4] and [V5] we only count the number of methods for which the values are ≥ 1, rather than summing-up the actual occurrences.

To answer **RQ_2** we will perform both descriptive statistics and hypothesis testing. To provide a fair comparison of the relatively small amount of methods that suffer from code smells, compared to those that do not, we have avoided the use of independent sample t-testing. Thus, we have preferred to calculate the mean change proneness ([V6]) of all methods per system and perform one-sample testing against this value. In this way, we can observe if the change proneness of methods that suffer from one smell is statistically different from the change proneness of the population (regardless of the existence/absence of code smells). In order to perform this analysis, for each type of smell, we filtered out methods that do not suffer from the corresponding smell. An overview of our data analysis plan (test, variables used, and notes) is presented in Table I.

Table I. Data Analysis Overview

Question	Variables	Statistical Analysis
RQ_1	[V3] [V4] [V5]	Frequency Table (actual value) Heatmap (per mille)
RQ_2	[V3] [V4] [V5] [V6]	One-sample Hypothesis Testing of [V6] against the mean [V6] of all project's methods Select cases based on [V3], [V4], or [V5]

Finally, to calculate the smell interest probability based on the results obtained from answering RQ_1 and RQ_2, we will calculate the joint probability of events. Specifically, as an event we consider the action of maintaining a module that suffers from a specific smell. This event holds a specific probability to occur. The probability that at least one of the modules suffering from the same smell will change (i.e., the interest probability of the smell), is calculated as the joint probability of any maintenance event to occur. The calculation of smell interest probability (vertical axis), contrasted with TDI interest probability (horizontal axis) is presented in Fig. 1. For example, for Smell-2, we can observe that its occurrence frequency is `3/n`, since it appears in three modules (i.e., `2`, `3` and `n`) and the mean change proneness of the modules it appears in is: $(cp_2 + cp_3 + cp_n) / 3$.

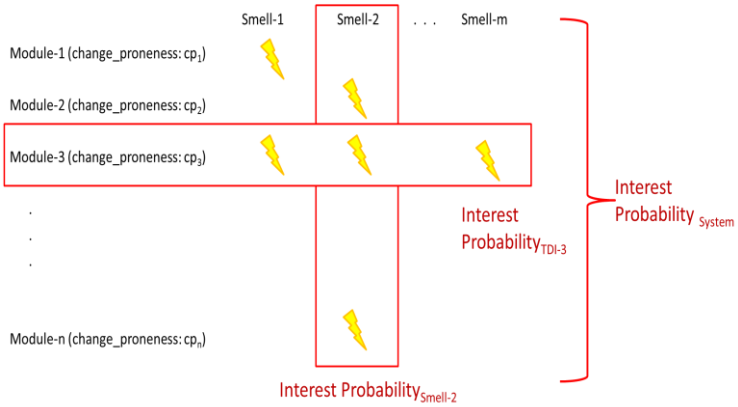

Fig. 1: Smell Interest Probability

To calculate the joint probability, we will use the data obtained from RQ_1 and RQ_2. In particular, the answer from RQ_1 will pro-

vide us with the number of smells in the system. In terms of the calculation, this will correspond to the ***number of events***, in the sense that there can be one maintenance action for resolving each smell occurrence. The answer to RQ_2 will provide us with the ***probability of each maintenance event to occur*** (i.e., the probability of each module that contains the smell to change and produce interest). Based on the mathematical formula, the joint probability of two events is calculated as follows, and accordingly scales to more than two events:

$$P(A|B) = P(A) + P(B) - P(A)*P(B)$$

5. RESULTS

In this section, first we present the answers to RQ_1 and RQ_2 (see Section 5.1) and then a synthesis of these results as an answer to the main research question (see Section 5.2).

5.1 Smell Interest Probability Factors

5.1.1 Occurrence Frequency of Code Smells (RQ_1)

Table II presents the actual count of methods in which we have identified smell occurrences, whereas in Fig. 2, we visualize a different view of the same information through a heatmap, by considering the frequency of smells per thousand methods. Fig. 2 allows to filter out project size, since the Spring Framework is substantially larger than the AndEngine. Based on the above, the results of Table II can be used only for within project interpretation, whereas the results of Fig. 2, for within smell interpretation.

Table II. Number of methods with smell occurrences

Project	Long Method	Conditional Complexity	Code Clones	Total
Spring	166	28	1689	1883
AndEngine	5	20	45	70

The results of Table II suggest that code clones are in both projects the most frequently detected code smell, while the ranking of long methods and conditional complexity smells (in terms of occurrence frequency) differs between the two projects. This outcome suggests that most of the code TD items (i.e. methods) identified into the projects suffer from code duplication.

By observing the findings presented in Fig. 2, and contrasting them to those of Table II, we can claim that the difference in the number of identified long methods across the two projects is not as large as it seems from the actual values. In particular, the level of magnitude for long methods is not substantially different, in the sense that we have identified approximately 4 long methods for every thousand methods for the Spring Framework, and 2 for every thousand methods of the AndEngine. However, the results for the Conditional Complexity are quite different: 0.6‰ for Spring and 6.6‰ for the AndEngine. The same happens for the total number of smells, as well: approx. 44‰ for Spring and 24‰ for AndEngine. A possible interpretation of this result is the (necessarily) higher complexity of the Spring Framework compared to the AndEngine. However, we note that this comparison is out of the scope of this manuscript, which basically aims at the comparison of different types of smells.

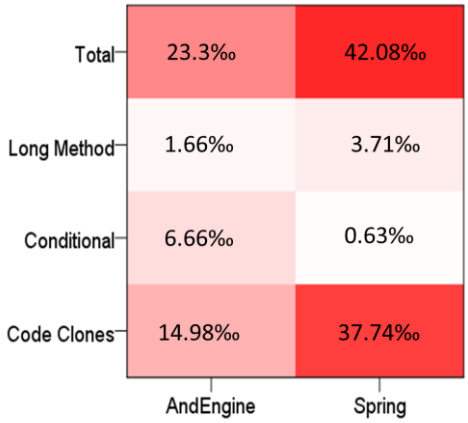

Fig. 2: Smell Frequency per Thousand Methods

The most frequent type of code TD is code clones. However, their frequency-level is project-related. Concerning long methods, approximately 2-4 can be identified in a thousand methods. The frequency of Conditional Complexity is also project related since it varies between less than one to 6 per mile in the two projects.

5.1.2 Change Proneness of Code Smells (RQ_2)

To perform one sample t-tests, we first needed to calculate the mean change proneness of all classes of the Spring Framework and the And Engine. Then, we could compare the change proneness of technical debt items (i.e., methods) suffering from each smell individually, to the specific value, and we check the existence of a statistically significant difference. The results of the hypothesis testing are presented in Table IV and Table V, respectively. In each table, we denote the significant differences with italic fonts in the sig. column.

Table IV. Change proneness of methods of Spring (test value: 0.39) – in 14,000 commits

Smell	Mean	Std. Dev.	Sig.	95% conf. interval	
				Low	Up
Long Method	2.00	4.549	*.000*	0.91	2.31
Conditional Complexity	2.36	3.793	*.011*	0.50	3.44
Code Clones	0.45	1.434	.106	-0.01	0.12

Table V. Change proneness of methods of AndEngine (test value: 0.72) – in 1,800 commits

Smell	Mean	Std. Dev.	sig.	95% conf. interval	
				Low	Up
Long Method	3.60	4.615	.235	-2.85	8.61
Conditional Complexity	3.21	3.735	*.009*	0.69	4.29
Code Clones	1.86	3.921	.060	-0.05	2.34

The results of both tables suggest that methods suffering from the *Conditional Complexity* smell are more change prone than the average method of a system, and this finding has proven to be statistically significant for both projects. On the other hand, *Code Clones* are usually identified into parts of the system whose

change proneness is not statistically different than the rest of the methods of the system. Finally, technical debt items that suffer from the *Long Method* smell, are significantly more change prone in the Spring Framework, but not in the AndEngine. However, even regarding the AndEngine the Long Methods are in average located to the most change prone methods of the system. The fact that this difference is not statistically significant is probably due to the small number of smells identified in the AndEngine (N=5). This observation can be explained by the fact that long methods serve more than one functionality. Thus, they subject to more "*reasons to change*" leading to a higher change proneness.

Methods that suffer from code smells are more change prone than TD-free methods. Among specific types of code smells, long methods and the use of conditionals instead of polymorphism are usually encountered in change prone methods. On the other hand code clones are usually positioned in system parts that do not change frequently.

5.2 Calculation of Smell Interest Probability

To assess interest probability of various types of code TD, we have quantified two parameters: (a) how many items suffer from each code smell (i.e., type of TD), and (b) what is the probability of each item to change in an upcoming commit, based on change history. Based on the outcome of RQ1 and RQ2 the two parameters do not uniformly rank the encountered code smells (e.g., code clones are the most frequently occurring smells, but are identified in the least change prone methods). Therefore there is a need of synthesizing the two pieces of information so as to assess the interest probability for each smell (as explained in Section 3.4). Based on the above information, we will calculate the interest probability for the studied systems. The results are presented in Table VI (Spring Framework) and Table VII (AndEngine).

Table VI. Interest Probability per Code-Smell (Spring)

	Long Method	Conditional Complexity	Code Clones
#TDIs (#events)	166	28	1689
Mean Change Probability (mean probability of event to occur)	$0.14e^{-3}$	$0.16e^{-3}$	$0.03e^{-3}$
Interest Probability	2.07%	0.44%	14.34%

Table VII. Interest Probability per Code-Smell (AndEngine)

	Long Method	Conditional Complexity	Code Clones
#TDIs (#events)	5	20	45
Mean Change Probability (mean probability of event to occur)	$2.00e^{-3}$	$1.78e^{-3}$	$1.03e^{-3}$
Interest Probability	0.99%	3.50%	4.53%

Based on the results of Table VI and VII, we can observe that interest probability can significantly vary for different projects. The interpretation of the results is as follows: in the best case scenario—i.e., AndEngine, there is an almost 4.5% probability that at least one method with a code clone (out of 45) will change in every commit, along system maintenance. The aforementioned results are considered intuitive in the sense that a single code clone is spread into multiple methods. Therefore, the same smell occurrence is affecting more than one method, whereas concerning the rest smells, each occurrence is located in a single TDI. By considering that, based on our observations, each clone is on average spread across 3.5 methods, the code clone occurrences are approximately at the same levels as the other two smells. However, we need to note that interest probability is correctly presented at method level rather than smell-occurrence level, because all clones will need to be updated (interest presence), even if one method of the clone is changed. Additionally, we can observe that the long method smell is the one showing the smallest deviation in terms of smell interest probability, in the examined projects, suggesting that this result is more reliable than the others.

Code clones is the smell that has the higher probability to produce interest in future maintenance activities in the two examined projects. This characteristic is mostly attributed to the smell occurrence frequency rather than its identification in change prone methods. The long method smell is the code TD type that presents the most similar smell interest probability in the examined projects

6. DISCUSSION

In this section we discuss the main findings of the case study and present the implications that this study provides to researchers and practitioners. In parallel we present interesting future work opportunities. The findings of the study suggest that:

- ***Long Methods*** are code smells of which 1.6 – 3.7 occurrences can be identified per mille methods, which however are changing 0.14 – 2.00 times per mile commits. Leading to an interest probability of 1.0%-2.0% per commit.
- ***Conditional Complexity*** is a smell that occurs in approximately 0.6 – 6.7 occurrences per mile methods, which are changing 0.16 – 1.78 times per mile commits. Leading to an interest probability of 0.5%-3.5% per commit.
- ***Code Clones*** is the most frequently occurring smell, since we have identified 14.9 – 37.7 occurrences per thousand methods. These methods were changing 0.03 – 1.03 times per thousand commits. Leading to an interest probability of 4.5%-14.0% per commit.

From the above information it becomes clear that the most frequently occurring bad smells (i.e., code clones) are placed in the least change prone parts of the system, whereas long methods, which are the rarest have been identified in the most frequently changing ones. The synthesis of the results suggests that code clones, despite their identification in less change prone methods, are the smell with the highest interest probability. The findings of this study can be used by practitioners in the following ways:

- ***Existence of smells and method change proneness***. Although this case study was not meant to explore whether heavy maintenance is responsible for introducing smells, or

if the existence of smells is responsible for the change frequency of the methods, we have revealed that a relation between the two exists. More particularly, more smells exist in more change-prone methods. Thus, we advise practitioners to be careful in the development and maintenance of change-prone modules, so as not to introduce code smells to them.

- ***Training in TD repayment***. The findings of this case study suggest that the interest probability for method-level smells is quite high (ranging from 9% to 16%, by summing up the probabilities of all smells). This finding suggests that the maintenance cost indeed increases due to code smells and that technical debt does not only lie in parts of the system that are not maintained. Thus, we advise practitioners to train on: (a) ways to prevent the creation of TD at the source code level, and (b) techniques to repay technical debt (e.g., refactorings).
- ***Alert on types of code TD***. Based on the results of this case study, we advise quality managers to alert developers, especially concerning the frequency of code clones occurrences. The amount of clones and the fact that a single smell occurrence can trigger interest on the modification of various modules, renders this type of code TD as one of paramount importance.

Regarding researchers, the methodology of this study has provided a structured way to assess the interest probability of various types of technical debt. The methodology can be reused / tailored in many ways, as follows:

- ***more smells***. The methodology can be applied to more code smells that are described in the book of Fowler et al. [13]. Applying the method to more smells would: (a) provide a holistic evaluation of code smells, and (b) make the results of such a study more accurate in the sense that in the current study we considered as TD-free the modules that do not involve instances of the three bad smells under investigation.
- ***different levels of granularity***. The methodology can be tailored to fit different levels of granularity, such as requirements, or architecture. Such an analysis would be of great importance in the sense that TD is a multi-perspective notion that spans across all development phases.
- ***more projects***. The application of the method to more projects would increase the reliability of the presented results. Also, it could possibly unveil differences in the interest probability of smell types in projects with different characteristics (e.g., size, maturity, history, levels of quality, etc.). An interesting special case of such an extension would be the application of the proposed approach to industrial projects, checking if there are differences compared to OSS.

7. THREATS TO VALIDITY

In this section, we present and discuss construct, reliability, external, and internal validity threats for this study.

Construct validity reflects to what extent the phenomenon under study really represents what is investigated according to the research questions [20]. Thus, concerning *construct validity*, the potential threats are related to the accuracy of the tools used to assess the change proneness of methods and to detect code smells (TD) in the source code. This is a construct validity threat in the sense that inaccurate results might lead to measuring a different phenomenon than the one that we originally intended to investigate. However, to mitigate this threat we used tools that have been evaluated in previous studies in terms of accuracy of the results they provide. Additionally we should mention that a potential threat is related to our definition of TD-free methods. As mentioned in Section 6, in this study as TD-free methods we consider methods that present none of the three studied smells. Thus, if additional code smells were studied this number would differ. In terms of *external validity*, i.e. possible threats while generalizing the findings derived from the sample to a general population [20], three potential threats have been identified. First, in our study we used systems written in Java and there is a possibility that the results would be different for other object-oriented languages. Second, results cannot be generalized to other code smells, or other types of TD, e.g., design, architecture etc. Finally, since the results have been obtained by studying two open source projects they cannot be generalized to the compete OSS population.

The reliability of the case study concerns the replicability of the collected data and the analysis performed, so that same results to be reproduced [20]. The study has limited *reliability* threats, since all research questions were answered by statistical analysis of automatically generated results, which involves no researcher bias. However, to assure the correct data analysis, two researchers collaborated and the one double-checked the results of the data analysis performed by the other researcher. Finally, all primitive data can be reproduced by using the dataset collected by GitHub (i.e. source code of the two projects and their evolution data), and the tools mentioned in Section 3. Nevertheless, we need to acknowledge that a replication with different tools for identifying code smell occurrences, might lead to different results. However, as mentioned before the accuracy of the employed tools has been successfully validated in empirical ways. Finally, *internal* validity is related to the identification of confounding factors, i.e., factors other than the independent variables that might influence the value of the dependent variable [20]. Internal validity is not relevant for our study since no causal relationships have been explored.

8. CONCLUSIONS

Efficient technical debt management requires the prioritization of repayment activities, since the complete repayment of technical debt is not feasible with limited resources. A rule of thumb for the selection of which technical debt items should be refactored and which should remain intact, suggests that quality assurance teams should first refactor modules that are more prone to produce interest, i.e., be maintained in the future. The quality attribute that assesses this possibility is change proneness.

In this study we performed an exploratory case study to identify the types of code TD that are more commonly placed in spots of the system that tend to change more frequently. In this way, we assess the interest probability of each type of TD, so as to aid quality managers in their decision making process. To achieve this goal we have studied more than 45,000 methods retrieved

from two well-known open source projects. The projects have been statically analyzed with state-of-the-art tools to identify code smell occurrences and assess the change proneness of the corresponding methods. The results of the study have indicated that source code spots, in which code smells are concentrated, present a higher probability to change compared to TD-free parts of the system. Additionally, the obtained results suggested that TDIs suffering from code clones present the highest interest probability (max: approximately 35%) compared to other types of code smells. Based on the findings of this study valuable implications to researchers and practitioners have been reported.

ACKNOWLEDGMENT

This research has been partially funded by the ITEA2 project 11013 PROMES.

REFERENCES

[1] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," Information and Software Technology, vol. 70, pp.100–121, 2016.

[2] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, P. Abrahamsson, A. Martini, U. Zdun, and K. Systa. The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study. In 8th International Workshop on Managing Technical Debt (MTD' 2016). IEEE Computer Society, 2016.

[3] A. Ampatzoglou, A. Chatzigergiou, S. Charalampidou, and P. Avgeriou. The Effect of GoF Design Patterns on Stability: A Case Study. IEEE Transactions on Software Engineering, 2015.

[4] A. Ampatzoglou, A. Gkortzis, S. Charalampidou, and P. Avgeriou. An Embedded Multiple-Case Study on OSS Design Quality Assessment across Domains. In Proceedings of the 2013 International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE Computer Society, pages 255–258, 2013.

[5] E. M. Arvanitou, A. Ampatzoglou, A. Chatziogeorgiou, and P. Avgeriou, "A Method for Assessing Class Change Proneness", Evaluation and Assessment in Software Engineering, ACM, Karlskrona, Sweden, 15-16 June 2017.

[6] E. M Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, M. Galster, and P. Avgeriou. "A Mapping Study on Design-Time Quality Attributes and Metrics". Journal of Systems and Software, 127(5):52–77, 2017.

[7] E. M. Arvanitou, A. Ampatzoglou, A. Chatziogeorgiou, and P. Avgeriou. Introducing a Ripple Effect Measure: A Theoretical and Empirical Validation. In 9th International Symposium on Empirical Software Engineering and Measurement (ESEM' 15). IEEE, 2015.

[8] V. Basili, G. Caldiera, D. Rombach, "The Goal Question Metric Approach", Encyclopedia of Software Engineering, John Wiley & Sons, pp. 528-532. 1994

[9] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of code smells in object-oriented systems", Innov. Syst. Softw. Eng. 10(1) , pp. 3-18. March 2014.

[10] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou. Identifying Extract Method Refactoring Opportunities based on Functional Relevance. IEEE Transactions on Software Engineering, 43, 2017

[11] S. Charalampidou, A. Ampatzoglou, and P. Avgeriou. Size and cohesion metrics as indicators of the long method bad smell: An empirical study. In 11th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2015). ACM, 2015.

[12] R. Eisenberg, "Management of Technical Debt: A Lockheed Martin Experience Report," 3rd International Workshop on Managing Technical Debt (MTD' 13), Baltimore, USA, 2013.

[13] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D.Roberts, "Refactoring: Improving the Design of Existing Code", Addison-Wesley Professional, 1 edition. July 1999.

[14] F. Khomh, M. Di Penta, Y-G Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness", In: Proceedings of the 16th working conference on reverse engineering (WCRE). IEEE Computer Society Press, Piscataway. 2009

[15] E. Murphy-Hill, C. Parnin, A.P.Black. How we refactor, and how we know it. In: Proceedings of 31st IEEE international conference on software engineering (ICSE'09), Vancouver, Canada, pp 287–297, 2009

[16] S. Olbrich, D.S. Cruzes, V. Basili, N. Zazworka. The evolution and impact of code smells: a case study of two open source systems. In: Proceedings of 3rd international symposium on empirical software engineering and measurement (ESEM'09), Florida,USA, pp 390–400, 2009

[17] J. Ratzinger, T. Sigmund, H.C. Gall. On the relation of refactorings and software defect prediction. In: Proceedings of 5thworking conference on mining software repositories (MSR'2008), Leipzig, Germany, pp 35–38, 2008

[18] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of Computer Programming, Volume 74, Issue 7, 2009, Pages 470-495

[19] C. K. Roy and J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," 2008 16th IEEE International Conference on Program Comprehension, Amsterdam, 2008, pp. 172-181.

[20] P. Runeson, M. Höst, A. Rainer, B. Regnell, "Case Study Research in Software Engineering: Guidelines and Examples", John Wiley and Sons, Inc. 2012.

[21] C. Seaman, Y. Guo, N. Zazworka, F. Shull, C. Izurieta, Y. Cai, and A.Vetró. "Using technical debt data in decision making: Potential decision approaches". In: 3rd International Workshop on Managing Technical Debt (MTD' 12), IEEE Computer Society, 2012

[22] C. Seaman, Y.Guo. "Measuring and Monitoring Technical Debt". Advances in Computers, Vol 82, pp. 25-46. Elsevier. 2011

[23] D. Silva, R. Terra, M. T. Valente, "Recommending automated extract method refactorings". In Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014), ACM, New York, NY, USA, pp.146-156. 2014.

[24] N. Tsantalis, A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods", Journal of Systems and Software, 84 (10), pp. 1757-1782. October 2011.

[25] N. Tsantalis, A. Chatzigeorgiou, Identification of refactoring opportunities introducing polymorphism, Journal of Systems and Software, Volume 83, Issue 3, pp. 391-404, March 2010