

Is There Any Correlation between Refactoring and Design Smell Occurrence?

Lerina Aversano¹^a, Mario Luca Bernardi¹^b, Marta Cimitile²^c, Martina Iammarino¹^d
and Debora Montano¹

¹University of Sannio, Department of Engineering, Benevento, Italy

²Unitelma Sapienza University, Rome, Italy

Keywords: Software Evolution, Software Quality, Refactoring, Design Smells.

Abstract: Software systems are constantly evolving making their architecture vulnerable to decay and the emergence of numerous design problems. This paper focuses on the occurrence of design smells in software systems and their elimination through the use of refactoring activities. To do this, the data relating to the presence of Design Smell, the use of refactoring, and the result of this use are analyzed in detail. In particular, the history of five open-source Java software systems and of 17 different types of design smells is studied. Overall, the results show that the removal of Design Smells is not correlated with the use of refactoring techniques. The analysis also provides useful insights about the developers' use of refactoring activities, the likelihood of refactoring on affected commits and clean commits, and removing and/or adding Design Smells both during refactoring and manual code cleaning operations.

1 INTRODUCTION

A problem widely discussed by the software engineering communities is software changes, which if done in a very short time often lead to design errors. Indeed, software systems' continuous evolution makes their architecture vulnerable to decay and introduces numerous design problems. In literature, several research articles deal with design issues and their impact on the evolution of the software system.

Not managing the design smells can lead to technical issues that increase the maintenance effort.

Often design smells are introduced as the result of inefficient decisions in terms of design and architecture (Hochstein and Lindvall, 2005; Garcia et al., 2009) that harm system internal qualities, such as maintainability, understandability, testability, extensibility, and reusability (Le and Medvidovic, 2016a; Aversano et al., 2007).


Of all the design smells defined in the literature (Fowler, 2002), our analysis focuses on 17 different types of design smells classified into four main cat-


egories: abstraction, encapsulation, modularization, and hierarchy. Furthermore, the study investigates the possible impact that refactorings have on source code components for both removal and internal quality improvement. The results confirm that source code elements affected by design smells are more prone to change, but developers seldom perform these changes through refactoring activities. This suggests that the removal of the design smell is not related to the presence of refactoring.


The remaining of the work is structured as follows: Section 2 reports studies related to our study, in Section 3 we detail the data extraction process (subsection 3.2), the features set (subsection 3.3) and the research questions (subsection 3.1). Section 4 describes the results of the empirical study we have conducted, while the threats that could affect the results obtained are reported in Section 5. Finally, Section 6 reports the conclusions and future work.


2 RELATED WORKS

Fowler (Fowler, 2018) defines refactoring as "A *disciplined technique for restructuring an existing body of code, altering its internal structure without changing*

^a <https://orcid.org/0000-0003-2436-6835>

^b <https://orcid.org/0000-0002-3223-7032>

^c <https://orcid.org/0000-0003-2403-8313>

^d <https://orcid.org/0000-0001-8025-733X>

its external behavior". The purpose of this technique is therefore to improve the errors present in the source code by modifying only the lines of code and not its final result.

Some empirical studies show that refactoring can only remove code smells included in the Fowler list (Chaparro et al., 2014; Aversano et al., 2008). For this reason, in recent years more and more researchers have investigated the role that refactoring plays in the removal of code smells and in the changes in quality metrics after its use (Cedrim et al., 2016; Chávez et al., 2017; Mkaouer et al., 2017; AlOmar et al., 2019; Fernandes et al., 2020). In particular, Bibiano et al. (Bibiano et al., 2019) surprisingly show that batch refactoring in 51% of cases introduces new smells and in 38% of cases does not remove all smells from the code.

Like our results, these studies show that refactoring is useful only in very few and rare cases.

Sahraoui et al (Sahraoui et al., 2000) report that the process between design improvement and its automation required developer validation and manual use of the control. So, it is important to understand when a code is smelly, how programmers recognize it, and when they care. A work of Yamashita et al. (Yamashita and Moonen, 2013) studies developers' knowledge of code smells through the use of a survey. Results show that a large number of programmers were unaware of the smells of the wrong code. They also show that, unfortunately, the knowledge of smells is entirely theoretical, and the name and definition are known, but this is so far from practice because most errors do not occur as described by the theory.

Furthermore, another study (Tufano et al., 2015) shows that every time a change is introduced in the source code it is more likely to induce new smells. They testify that the number of smells is greater when developers use the refactoring technique and for this reason, the refactoring technique is used only for source code that has at least one critical attribute (Fernandes et al., 2020). A lot of attention is paid to design smells because they are frequently indicators of architectural problems of the code and often a symptom of the harmful maintainability of the software (Aversano et al., 2020a; Le and Medvidovic, 2016b; Fowler, 2002; Alkharabsheh et al., 2019). For this reason, the study (Tsantalis and Chatzigeorgiou, 2009) emphasizes the need for tools capable of recognizing the presence of design smells in code before their introduction into it. The authors have predicted design smells through a new version of Temporal Convolutional Networks (TCN) that provides better estimates than traditional TCN. Some empirical studies (Ardimento et al., 2021; Aversano et al.,

2020b; Sharma et al., 2020; Aversano et al., 2020a) show that refactoring cannot remove any kind of design smells from the source code. Furthermore, in (Aversano et al., 2020b) it is shown that refactoring most of the time introduces new smells into the code. These studies also found that code smells removal is more likely when refactoring is not done. Driven by these results, our work aims to analyze the impact of refactoring on design smells and try to overturn or confirm what others have found to date.

3 APPROACH

The execution of the empirical study entails the following steps:

- identification of the research questions;
- extraction and gathering of the required data;
- identification of the feature set.

Each step is described in more detail in the following subsections.

3.1 Research Questions

Below are the research questions on which this study is based.

- *RQ1: Design Smells VS Refactoring: Do Programmers Use Refactoring When Design Smells Occur?* We aim to link developers' use of refactoring to design smells occurrences.
- *RQ2: Is refactoring Correlated to the Type of Design Smell?* Our purpose is to analyze the probability, given a design smell presence, of refactoring activities. Furthermore, we want to investigate if there are any relationships between the type of design smell involved and such refactoring probability.
- *RQ3: How Often Are Refactoring and Elimination of Design Smells Co-occur?* Our goal is to test to which extent refactoring activities and design smells removal occur together during the history of commits.

3.2 Data Extraction

The data used for this study comes from 5 open-source Java software systems. Table 1 presents in the first column the names of the system considered, in the second column the tags created to briefly name each of them, in the third column the number of commits analyzed, and in the last column the dates of the

Table 1: Software systems involved into the analysis.

System	Tag	#commit	First - Last Commit Date
Atlas	AT	13667	08/12/2014 - 05/07/2019
Guice	GU	8567	16/11/2006 - 06/06/2019
jUnite4	JU	5442	03/12/2000 - 20/06/2019
Log4j	LO	9117	14/12/2000 - 11/02/2014
Zookeeper	ZO	6216	06/11/2007 - 17/07/2019
Total	TOT	43009	03/12/2000 - 17/07/2019

first and last commit taken into consideration. These systems are selected considering that their repositories are available on Github ¹, have different sizes, and belong to different domains. The data collection follows a process made of several steps. The first step is consisted of extracting the source code elements to analyze all the commits of each system. On the extracted data, we execute the tool for detecting the presence of smells, and for the refactoring detection. Finally, we compare each commit with its previous one to identify a possible addition or removal of the design smell and collect refactoring activities by commit. More specifically, we use two tools: Designite ² and RefactoringMiner³. Respectively, the first for the detection of smells, capable of evaluating the quality of the design and therefore identifying the presence of design smells. Therefore, for the sake of completeness, in Table 2 we report the design smells detected by this tool, indicating in the first column the macro-categories of design smells, and in the second the list of smells. The second tool identifies the possible presence of refactoring activities carried out on the source code. More specifically, this is capable ⁴ of detecting 40 kinds of refactorings, for operations carried out on packages, classes, methods, features, parameters, and attributes. Finally, the dataset includes the history of all software systems composed of 43009 commits and 35 features related to each type of design smell occurrence, the respective removal, and the presence of the refactoring on the commit and the specific class.

3.3 Feature Set

To examine the correlations between refactoring, design smells, and the removal of the latter, we use the following features:

- **Refactoring:** a dichotomous feature that assumes the value "true" when the refactoring techniques are applied to its reference commit, "false" otherwise
- **Design Smells:** we analyze 17 different types of smell. These smells are represented as dichoto-

¹<https://github.com>

²<https://www.designite-tools.com>

³<https://github.com/tsantalis/RefactoringMiner>

⁴At the time of extraction, in its version 2.1

Table 2: Type of design smells analyzed.

Design smell Category	Type of smells
ABSTRACTION	Imperative Abstraction Multifaceted Abstraction Unnecessary Abstraction Unutilized Abstraction
ENCAPSULATION	Deficient Encapsulation Unexploited Encapsulation
MODULARIZATION	Broken Modularization Cyclically Dependent Modularization Insufficient Modularization Hub-Like Modularization
HIERARCHY	Broke Hierarchy Cyclic Hierarchy Deep Hierarchy Missing Hierarchy Multipath Hierarchy Rebellious Hierarchy Wide Hierarchy

mous features with a value of '0' if the particular type of smell does not appear in the commit under consideration, a value of '1' otherwise.

- **Removal of the Design Smells:** 17 nominal features, one for each design smells, which take into account the removal of the smell they refer to. In particular, these features can assume three values: *no change* if the smell is not been eliminated, *smell eliminated* if the smell is been eliminated and *added smell* if the smell is been added.
- **Year:** the year in which the current commit is made.

4 EMPIRICAL STUDY RESULTS

This section reports all the results obtained through the data to approach previously explained.

4.1 Do Programmers Use Refactoring When Design Smells Occur?

To understand if there is a connection between design smells and the use of refactoring on them, we analyze the use of refactoring on commits affected and non by the presence of a design smells. We analyze the contingency tables, double-entry frequency tables in which two qualitative features cross, in our case, the number of the commit where a particular type of design smell occurs, and the Refactoring feature. In Tables 3 and 4, we report respectively the contingency analysis carried out on smelly commits and the one carried out on clean commits, therefore unaffected by the presence of design smells. Tables show in the first column the type of the smell, in the second the number of commits with at least one smell present, which have not been refactored, in the third the number of commits in which there is at least one smell and it has

been carried out a refactoring activity, in the fourth the sum of the two previous columns, and in the last two columns the calculation of the respective ratios, without refactoring and with refactoring. In particular, from the fourth column of Table 3 we can see that some design smells appear more frequently, such as Unutilized Abstraction, Deficient Encapsulation, Cycle-Dependent Modularization, Insufficient Modularization, and Broken Modularization. To better understand if the refactoring technique is used on affected smells, we study the proportion between commits where refactoring is used compared to the total of commits by design type smells, and between the commits where refactoring is not used compared to the total of the affected commits. The results on the smelly commits, reported in Table 3, highlight that in most commits affected by design smells, no refactoring is used on them. Looking at the Table, the number of commits affected by design smells where refactoring is not used varies between 61% in the case of the Hub-like Modularization and 100% for the Deep Hierarchy. Therefore, on average, for all types of smells considered, 78% of smelly commits are not refactored. The design smells types handled by refactoring techniques are in the order: Deep Hierarchy (100%), Unnecessary Abstraction (97%), Imperative Abstraction (90%), Cyclic Hierarchy (86%), and Rebellious Hierarchy (82%). These types of design smells are also the ones that occur less frequently. Therefore, we can say that refactoring is seldom applied to rare smells. On the other hand, on average 22% of commits affected by design smells undergo refactoring operations. In very few cases, in particular, for Unexploited Encapsulation, Cyclic Dependent Modularization, Insufficient Modularization, Hub-like Modularization, and Multipath Hierarchy, the percentage of commits affected by design smells and affected by refactoring is greater than or equal to 30%. The results reported in Table 4 show that the number of refactored commits is much greater than those affected by design smells and refactored. However, if we consider the totality of commits not affected by design smell, from the Table it is clear that most of the clean commits are not refactored, in fact, on average, the refactoring techniques are used on about 25% of the commits without the presence of smells design. Therefore, refactoring is done more on commits with no design smells than on commits with design smells.

Summary: The use of refactoring is not as common on commits affected by design smells and, in particular, it is almost not used in the case of rare smells.

Table 3: Contingency Table & Table of Ratio - Refactoring VS Commits affected by Design smells.

Type of design smell - ALL DATA	Ref.false	Ref.true	Ref.false	Ref.true
Imperative Abstraction	62	7	69	0.90
Multifaced Abstraction	218	68	286	0.76
Unnecessary Abstraction	153	4	157	0.97
Unutilized Abstraction	10517	2765	13282	0.79
Deficient Encapsulation	5661	2173	7834	0.72
Unexploited Encapsulation	154	66	220	0.70
Broken Modularization	52	15	67	0.78
Cyclic Dependent Modularization	4499	2136	6635	0.68
Insufficient Modularization	5757	2525	8282	0.70
Hub like Modularization	359	234	593	0.61
Broken Hierarchy	1930	634	2564	0.75
Cyclic Hierarchy	6	1	7	0.86
Deep Hierarchy	5	0	5	1.00
Missing Hierarchy	309	135	444	0.70
Multipath Hierarchy	9	4	13	0.69
Rebellious Hierarchy	58	13	71	0.82
Wide Hierarchy	147	47	194	0.76

Table 4: Contingency Table & Table of Ratio - Refactoring VS Commits not affected by Design smells.

Type of design smell - ALL DATA	Ref.false	Ref.true	Ref.false	Ref.true
Imperative Abstraction	29097	9563	38660	0.75
Multifaced Abstraction	29041	9502	38543	0.75
Unnecessary Abstraction	29006	9566	38572	0.75
Unutilized Abstraction	18642	6805	25447	0.73
Deficient Encapsulation	23498	7397	30895	0.76
Unexploited Encapsulation	29005	9504	38509	0.75
Broken Modularization	29107	9555	38662	0.75
Cyclic Dependent Modularization	24660	7434	32094	0.77
Insufficient Modularization	23402	7045	30447	0.77
Hub like Modularization	28800	9336	38136	0.76
Broken Hierarchy	27229	8637	35866	0.76
Cyclic Hierarchy	29153	9569	38722	0.75
Deep Hierarchy	29154	9570	38724	0.75
Missing Hierarchy	28850	9435	38285	0.75
Multipath Hierarchy	29147	9566	38713	0.75
Rebellious Hierarchy	29098	9557	38655	0.75
Wide Hierarchy	29009	9523	38532	0.75

4.2 Is Refactoring Correlated to the Type of Design Smell?

To understand the relationships between the design smell and the use of refactoring on it, we use the logistic regression model.

In Table 5 we report the results of the logistic model on the entire dataset. In particular, the first column shows the type of predictor, the second the values of the *BETA coefficients*, a measure of the relationships between the dependent feature and the predictive features, the third column the values of the *Standard Error* associated with BETA coefficients, fourth and fifth column the values of *z-value* and of the relative *p-value*, which respectively indicate the number of standard deviations of each data value with respect to the mean, and how much the BETA coefficient is significantly correlated to the dependent feature (The BETA coefficient is significant when the p-value is less than 0.05) and finally in the last column the values of the *Odds Ratio* (OR) which represents the probability ratios between design smells and Refactoring.

The results highlight that when certain types of smells are present such as Unnecessary Abstraction, Unutilized Abstraction, Cycle Dependent Modularization, and Rebellious Hierarchy the probability of refactoring decreases because the OR is < 1. Otherwise, when the design smell is Deficient Encapsu-

Table 5: Logistic Model Estimates (all data).

Coefficients, Estimates and Odds Ratio - all data					
	Beta	St. Err	z value	Pr(> z)	OR
(Intercept)	-0.83	0.02	-51.48	<2e-16	0.44
Imperative Abstraction	-0.43	0.29	-1.48	0.14	
Unnecessary Abstraction	-2.10	0.39	-5.43	0.00	0.12
Unutilized Abstraction	-0.23	0.02	-9.84	<2e-16	0.79
Deficient Encapsulation	0.08	0.03	2.97	0.00	1.08
Unexploited Encapsulation	-0.31	0.18	-1.77	0.08	
Broken Modularization	-0.41	0.29	-1.42	0.15	
Cyclic Dependent Modularization	0.14	0.03	4.66	0.00	1.15
Insufficient Modularization	0.47	0.03	18.36	<2e-16	1.60
Broken Hierarchy	0.11	0.04	2.62	0.01	1.12
Cyclic Hierarchy	3.14	1.05	2.98	0.00	23.10
Deep Hierarchy	-12.87	53.44	-0.24	0.81	
Missing Hierarchy	0.20	0.11	1.78	0.07	
Rebellious Hierarchy	-0.72	0.29	-2.46	0.01	0.49
Wide Hierarchy	0.40	0.13	3.22	0.00	1.50

Table 6: Table of Added smells when Refactoring is used.

ADDED SMELL vs REFACTORING						
Type of design smells	ALL	ATLAS	GUICE	JUNITE	LOG4J	ZOOKEEPER
Imperative Abstraction	0%	0%	0%	0%	0%	0%
Multifaced Abstraction	2%	3%	3%	3%	0%	0%
Unnecessary Abstraction	0%	0%	0%	0%	0%	0%
Unutilized Abstraction	13%	13%	11%	24%	16%	5%
Deficient Encapsulation	15%	11%	10%	23%	25%	18%
Unexploited Encapsulation	0%	0%	2%	0%	0%	2%
Broken Modularization	0%	0%	0%	0%	1%	0%
Cyclic Dependent Modularization	23%	26%	39%	15%	17%	13%
Insufficient Modularization	37%	38%	28%	26%	39%	37%
Hub-like Modularization	4%	5%	2%	0%	0%	6%
Broken Hierarchy	4%	4%	3%	9%	2%	2%
Cyclic Hierarchy	0%	0%	0%	0%	0%	0%
Deep Hierarchy	0%	0%	0%	0%	0%	0%
Missing Hierarchy	0%	0%	2%	0%	0%	2%
Multipath Hierarchy	0%	0%	0%	0%	0%	0%
Rebellious Hierarchy	1%	0%	0%	0%	0%	15%
Wide Hierarchy	0%	0%	0%	0%	0%	0%
	100%	100%	100%	100%	100%	100%

lation, Cycle-Dependent Modularization, Insufficient Modularization, Broken Hierarchy, Cyclic hierarchy, and Wide Hierarchy, the probability of refactoring increases because the OR is > 1 . More specifically, when the smell Cyclic Hierarchy is present, the probability of refactoring increases 23 times more.

Summary: The probability to use refactoring is connected with the type of the design smells.

4.3 How Often Are Refactoring and Elimination of Design Smells Co-occur?

To address this research question, we try to understand the effects of refactoring operations on source code in terms of management and removal of design smells.

We first analyze the distribution frequencies of the added or removed design smells in those commits in which the presence of refactoring activities is also detected. To conduct this analysis we only consider commits where refactoring activity is detected. In Table 6 we report the distribution of the added design smells, while in Table 7 the distribution of the removed design smells in concomitance with the refac-

Table 7: Table of Removed smells when Refactoring is used.

REMOVED SMELL vs REFACTORING						
Type of smells	ALL	ATLAS	GUICE	JUNITE	LOG4J	ZOOKEEPER
Imperative Abstraction	0%	0%	0%	0%	0%	1%
Multifaced Abstraction	3%	6%	2%	0%	0%	0%
Unnecessary Abstraction	0%	0%	0%	0%	0%	1%
Unutilized Abstraction	22%	18%	31%	16%	20%	53%
Deficient Encapsulation	13%	9%	6%	18%	25%	15%
Unexploited Encapsulation	0%	0%	1%	0%	0%	0%
Broken Modularization	0%	0%	0%	0%	0%	0%
Cyclic-Dependent Modularization	23%	28%	39%	21%	11%	1%
Insufficient Modularization	23%	35%	16%	13%	2%	19%
Hub-like Modularization	1%	0%	1%	0%	0%	8%
Broken Hierarchy	10%	4%	3%	32%	26%	3%
Cyclic Hierarchy	1%	1%	0%	0%	0%	0%
Deep Hierarchy	0%	0%	0%	0%	0%	0%
Missing Hierarchy	0%	0%	2%	0%	0%	0%
Multipath Hierarchy	0%	0%	0%	0%	0%	0%
Rebellious Hierarchy	0%	0%	0%	0%	0%	0%
Wide Hierarchy	4%	0%	0%	0%	17%	0%
	100%	100%	100%	100%	100%	100%

toring operations. More specifically, tables show in the first column the type of smell, and in the following columns the percentages of their addition in all systems and then in the single systems considered for the study (Table 6), and the percentages of their removal (Table 7) in the commits in which refactoring is present. The results of the distribution of the added design smell show that not all types of design smells are added after refactoring but the most added design smells are Insufficient Modularization (37%), Cyclic Dependent Modularization (23%), Deficient Encapsulation (15%), and Unutilized Abstraction (13%). Considering the individual systems, for all systems, except Guice, the most added design smell is Insufficient Modularization. Similarly, the results of the distribution of removed smells show that not all types of design smells are removed with the same frequency where refactoring is also been done. The most removed design smells are Insufficient Modularization (23%), Cycle Dependent Modularization (23%), Unutilized Abstraction (22%), Deficient Encapsulation (13%), and Broken Hierarchy (10%). The results are almost the same for the single system analysis.

Moreover, to understand if the number of design smells introduced is greater or less than those eliminated and vice versa, we analyze the relationship between design smells added and design smells removed (i) and the relationship between design smells removed and smells added (ii). The ratio is calculated by dividing the number of design smells for a particular type over the total of all smells counted in the reference dataset. In Table 8 we report in the first columns the type of design smell, from the second to the seventh column the ratios of added/removed and from the eighth column at the end of the Table, the ratios of removed/added are reported. Focusing on the relationships between added design smells and removed design smells we have found that in most cases smells are added rather than removed where refactoring has also been done. In particular, Unexploited

Table 8: Table of ratios 'Added vs Removed' & Table of ratios 'Removed vs Added'.

Type of smells	RATIO ADDED vs REMOVED						RATIO REMOVED VS ADDED					
	ALL	ATLAS	GUICE	JUNITE	LOG4J	ZOOKEEPER	ALL	ATLAS	GUICE	JUNITE	LOG4J	ZOOKEEPER
Imperative Abstraction	-	-	-	-	-	-	-	-	-	-	-	-
Multifaced Abstraction	0.82	0.72	2.00	-	-	-	1.22	1.39	0.50	-	-	-
Unnecessary Abstraction	-	-	-	-	-	-	-	-	-	-	-	-
Unused Abstraction	0.96	1.19	0.49	1.80	1.07	0.27	1.04	0.84	2.04	0.56	0.94	3.77
Deficient Encapsulation	1.83	2.24	2.40	1.55	1.31	3.43	0.55	0.45	0.42	0.65	0.76	0.29
Unexploited Encapsulation	12.00	-	5.00	-	-	-	0.08	-	0.20	-	-	-
Broken Modularization	-	-	-	-	-	-	-	-	-	-	-	-
Cyclic-Dependent Modularization	1.66	1.57	1.41	0.85	2.08	33.00	0.60	0.64	0.71	1.18	0.48	0.03
Insufficient Modularization	2.61	1.84	2.46	2.38	25.89	5.39	0.38	0.54	0.41	0.42	0.04	0.19
Hub-like Modularization	9.00	39.00	2.00	-	-	2.43	0.11	0.03	0.50	-	-	0.41
Broken Hierarchy	0.55	2.12	1.20	0.35	0.08	1.33	1.83	0.47	0.83	2.86	13.00	0.75
Cyclic Hierarchy	-	-	-	-	-	-	-	-	-	-	-	-
Deep Hierarchy	-	-	-	-	-	-	-	-	-	-	-	-
Missing Hierarchy	4.00	-	1.67	-	-	-	0.25	-	0.6	-	-	-
Multipath Hierarchy	-	-	-	-	-	-	-	-	-	-	-	-
Rebellious Hierarchy	-	-	-	-	-	-	-	-	-	-	-	-
Wide Hierarchy	0.03	-	-	-	-	-	38.00	-	-	-	-	-

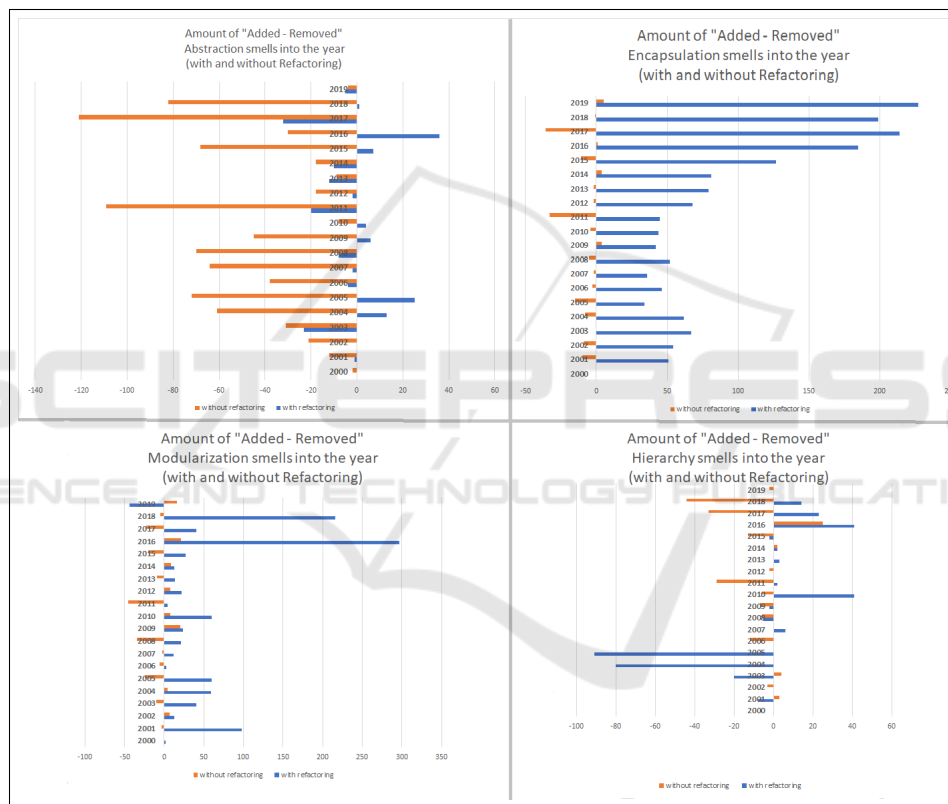


Figure 1: Differences between 'Added' and 'Removed' design smells - temporal analysis.

Encapsulation, Hub-like modularization, and Missing Hierarchy are the design smells with the highest ratio as well as the smells with the highest percentage of introductions in the code. The analysis by system shows the same results.

These results are also confirmed by the analysis of the relationship between design smells removed and design smells added. We find that all ratios are less than 1, so the removed design smells are inferior to the design smells introduced into the code where refactoring is done. The only type of design smell on

with the practice of refactoring is useful is Wide Hierarchy.

We have also analyzed the distribution of added and removed design smells as a function of the 'Year' feature, to eliminate the redundancy of information due to time. The data collection goes 2000 and 2019.

To understand if more design smells are added or removed, we calculate the differences between the design smells added and removed year by year. So when the difference is negative it means that the design smells added in a year are more than the removed

ones, otherwise when the difference is positive. In Figure 1 we show the results for the four categories of smell, Abstraction, Encapsulation, Modularization, and Hierarchy, reporting with the blue bars the differences on which the refactoring is performed, and with the orange bars the differences on which is not refactored. It can be seen that the only category of design smells that has more negative differences between added and removed design smells is the Abstraction Category, but this happens, most of the time, when there is no refactoring. In all other categories, the number of design smells added is greater than the number of design smells removed. This is especially true when also refactoring tasks are present (most of the blue bars are on the positive side of the graphs).

Summary: Refactoring is very often more related to the introduction of new design smells rather than to the removal. When refactoring is used, the added design smells are 1.60 times greater than the number of removed design smells.

5 THREATS TO VALIDITY

The threats to the validity concern the relationship between the theoretical world and what we observe.

Concerning the removal of Design Smells, the dataset is built and validated through a few studies and could be considered reliable. To mitigate the same threat, in the case of the Refactoring detection, we use the Refactoring Miner tool, for which the (Tsan-talis et al., 2018) study showed that this tool has a high precision (98%) and recall (87%).

Threats to internal validity relate to factors internal to our research that could affect the results obtained. In general, we cannot pretend that the cause-effect relationship between the refactoring and removal actions of Design Smells is always the same as that obtained from our analyses. To mitigate this threat, we investigate refactoring on both smells-affected commits and clean commits where possible.

The threats to the validity of the conclusions are due to the lack of generalization of the results obtained. To mitigate this threat, we consider five different systems, carrying out the analyses both on the individual systems, taking into account their particularities, and on the union of these, to obtain results for both single sample and overall population.

The generalizability of our results is also an external threat, but being this a preliminary work, we analyze only five software systems. The results can

be consolidated or denied during the study phases of future work.

6 CONCLUSIONS AND FUTURE WORK

This study proposes a novel approach to analyzing the relationship between design smells and refactoring activities.

We show that refactoring techniques are still very little used in the world of software development. We found that the likelihood of using refactoring in commits where smells are present is closely related to the type of design smell present in the code. In particular, refactoring is more linked to the presence of Encapsulation and Modularization design smells, and less to the presence of Abstraction and Hierarchy design smells. This is an interesting aspect worth investigating on a larger number of systems in a dedicated study. We also found that the refactoring activity is not correlated with the removal of some particular design smells. More specifically, for the Wide Hierarchy smell, and in most commits where there is refactoring there is the addition of other design smells, rather than the removal of those already existing.

Future work could be a predictive model that can forecast the behavior of refactoring in terms of adding or removing design smells. It would be interesting to understand how the use of refactoring changes as the number of types of design smells increases and to examine multiple software systems even with different programming languages to validate the techniques and results obtained in this study.

REFERENCES

- Alkharabsheh, K., Crespo, Y., Manso, E., and Taboada, J. A. (2019). Software design smell detection: a systematic mapping study. *Software Quality Journal*, 27(3):1069–1148.
- AlOmar, E. A., Mkaouer, M. W., Ouni, A., and Kessentini, M. (2019). On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE.
- Ardimento, P., Aversano, L., Bernardi, M. L., Cimitile, M., and Iammarino, M. (2021). Temporal convolutional networks for just-in-time design smells prediction using fine-grained software metrics. *Neurocomputing*, 463:454–471.
- Aversano, L., Bernardi, M. L., Cimitile, M., Iammarino, M., and Romanyuk, K. (2020a). Investigating on the rela-

- tionships between design smells removals and refactorings. In *ICSOFT*, pages 212–219.
- Aversano, L., Carpenito, U., and Iammarino, M. (2020b). An empirical study on the evolution of design smells. *Information*, 11(7):348.
- Aversano, L., Cerulo, L., and Del Grosso, C. (2007). Learning from bug-introducing changes to prevent fault prone code. pages 19–26.
- Aversano, L., Cerulo, L., and Palumbo, C. (2008). Mining candidate web services from legacy code. pages 37–40.
- Bibiano, A. C., Fernandes, E., Oliveira, D., Garcia, A., Kalinowski, M., Fonseca, B., Oliveira, R., Oliveira, A., and Cedrim, D. (2019). A quantitative study on characteristics and effect of batch refactoring on code smells. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE.
- Cedrim, D., Sousa, L., Garcia, A., and Gheyi, R. (2016). Does refactoring improve software structural quality? a longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*, pages 73–82.
- Chaparro, O., Bavota, G., Marcus, A., and Di Penta, M. (2014). On the impact of refactoring operations on code quality metrics. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 456–460. IEEE.
- Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., and Garcia, A. (2017). How does refactoring affect internal quality attributes? a multi-project study. In *Proceedings of the 31st Brazilian symposium on software engineering*, pages 74–83.
- Fernandes, E., Chávez, A., Garcia, A., Ferreira, I., Cedrim, D., Sousa, L., and Oizumi, W. (2020). Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology*, 126:106347.
- Fowler, M. (2002). Tutorials-refactoring: Improving the design of existing code. *Lecture Notes in Computer Science*, 2418:256–256.
- Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
- Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009). Toward a catalogue of architectural bad smells. In *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems, QoSA '09*, page 146–162, Berlin, Heidelberg. Springer-Verlag.
- Hochstein, L. and Lindvall, M. (2005). Combating architectural degeneration: A survey. *Inf. Softw. Technol.*, 47(10):643–656.
- Le, D. and Medvidovic, N. (2016a). Architectural-based speculative analysis to predict bugs in a software system. In *Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16*, page 807–810, New York, NY, USA. Association for Computing Machinery.
- Le, D. and Medvidovic, N. (2016b). Architectural-based speculative analysis to predict bugs in a software system. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 807–810.
- Mkaouer, M. W., Kessentini, M., Cinnéide, M. Ó., Hayashi, S., and Deb, K. (2017). A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, 22(2):894–927.
- Sahraoui, H. A., Godin, R., and Miceli, T. (2000). Can metrics help bridging the gap between the improvement of oo design quality and its automation. In *Proc. of the International Conference on Software Maintenance (ICSM'00)*, pages 154–162.
- Sharma, T., Singh, P., and Spinellis, D. (2020). An empirical investigation on the relationship between design and architecture smells. *Empirical Software Engineering*, 25(5):4020–4068.
- Tsantalis, N. and Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367.
- Tsantalis, N., Mansouri, M., Eshkevari, L., Mazinianian, D., and Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 483–494. IEEE.
- Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414. IEEE.
- Yamashita, A. and Moonen, L. (2013). Do developers care about code smells? an exploratory survey. In *2013 20th working conference on reverse engineering (WCRE)*, pages 242–251. IEEE.