Investigating on the Relationships between Design Smells Removals and Refactorings

Lerina Aversano[1], Mario Luca Bernardi[1], Marta Cimitile[2], Martina Iammarino[1] and Kateryna Romanyuk[3]
[1]Department of Engineering, University of Sannio, Via Traiano, Benevento, Italy
[2]UnitelmaSapienza, University of Rome, Italy
[3]CERICT, Benevento, Italy

Keywords: Design Smells, Software Maintenance, Software Evolution.

Abstract: Software systems continually evolve and this conducts to its architectural degradation due to the existence of numerous design problems. The presence of Design Smells is the main indicator of such problems, it points out the use of constructs that generally hurt system evolution. In this work, an investigation on Design Smells removals has been performed, focusing specifically on the co-occurrence of refactoring and related changes performed on a software system. An empirical study has been conducted considering the evolution history of 5 software systems. The detection of instances of multiple Design Smell types has been performed, along with all the history of the systems, along with, the detection of refactoring activities. The empirical study shows that Design Smells removals are not correlated to the presence of refactoring. The analysis provides useful indications about the percentage of activities conducted on smelly classes, including refactoring (even if these activities in few cases lead to effective smell removals).

1 INTRODUCTION

Software system evolution is usually affected by challenging trade-off between short-term and long-term goals. Usually, stringent deadlines lead to design problems inducing rework costs. In particular, design violations taken to satisfy fast delivery might compromise software maintainability and introduce technical debt.

In the literature, there are numerous studies investigating design problems introduced along with the evolution of a software system. Specifically, at the design level the presence of construct design problems, the so-called Design Smells, contribute to system erosion. Design smells can be defined as indicators of poor design quality (Hochstein et al., 2005, De Silva et al., 2012, Garcia et al., 2009, Le et al., 2016, Sharma et al., 2016), and are originated from implementation constructs (e.g., classes).

Design Smells are closely related to system evolution, as often an organization necessity deal with accumulated design problems when adding new features to a software system. Unmanaged, Design Smells can lead to significant technical problems and increased maintenance and evolution efforts.
More in detail, during the software evolution, the amount and complexity of the interactions among the software elements increase, with a consequent effect on the design structure.

In this work, an investigation on the Design Smells removals has been performed, focusing on the relationships with refactoring performed on the software system (Bernardi et Al., 2016).

An empirical study has been conducted considering the evolution history of 5 software systems. The detection of instances of multiple Design Smell types has been performed, as well as the detection of the refactoring activities. Then a co-occurrence analysis has been conducted to investigate their relationships.

The empirical study confirmed that classes affected by Design Smells are more subject to refactoring, evidencing that especially when multiple smells are detected in the same classes these are more frequently subject to changes. Moreover, it emerged that in some cases Design Smells are removed, and

Aversano, L., Bernardi, M., Cimitile, M., Iammarino, M. and Romanyuk, K.
Investigating on the Relationships between Design Smells Removals and Refactorings.
DOI: 10.5220/0009887102120219
In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT 2020)*, pages 212-219
ISBN: 978-989-758-443-5

smells removals are not correlated to the presence of refactoring.

The rest of the paper is organized as follow: Section 2 discusses the related work, Section 3 describes the empirical study design, while the results of the study are reported in Section 4. Section 5 discusses the threats that could affect the obtained results, finally, conclusions are given in the last section.

2 RELATED WORK

The impact of code level smells defined by Fowler (Fowler, 1999) has been widely investigated in the research literature. In particular, several studies analysed their effects on maintainability (Sjoberg et al., 2013, Tufano et al., 2015, Sjoberg et al., 2013), program comprehension (Abbes et al., 2011), change, and fault-proneness (Khomh et al, 2012, Palomba et at., 2017).

Currently, there are also several tools used to automatically detect (Moha et al., 2010, Palomba et al., 2015), code smells, exploiting different sources of information. Several papers discuss code smells fix through the application of refactoring operations (Tsantalis et al., 2009, Suryanarayana et al., 2014).

At architectural level smells are ultimately instances of poor design decisions (Garcia et al., 2009). The technical debt is due to the presence of construct design problems, the so-called architectural smells (Hochstein et al., 2005, de Silva et al., 2012, Garcia et al. 2009), that contribute to system erosion. They have a negative impact on system life cycle properties, such as understandability, testability, extensibility, and re-usability (Le et al., 2016).

Several research papers in the literature deal with the detection of architectural smells, and part also with the influence of architectural smell on issue related activities. Brunet et al. (Brunet et al., 2012) studied the evolution of architectural violations in 76 versions selected from four subject systems showing how the number of architectural violations is constantly growing over time. Moreover, some previously identified violations reappear, and in all the studied systems a critical core is identified and this core does not change over time.

Arcan (Arcelli Fontana et al., 2017) is a static analysis tool targeted at the detection of three architectural smells, including cycles and hubs. Arcan creates a graph database containing the structural dependencies of a Java system and then runs several detection algorithms (one per smell) on this graph. Finally, there are some commercial tools for detecting architectural smells, such as Designite (Sharma et al., 2016), which identifies seven architecture smells, including cycles and other dependency-based smell. As far as we are aware, all the previous tools have no predictive capabilities.

Le et al. (Le et al., 2016) presented an empirical study to date of architectural decay and its impact on software systems. For each version of the system, different architectural recovery techniques have been applied considering different types of smells. They examined the relationships between the collected smells and the issues extracted from the repositories of the various systems in question. This has shown how architectural decay can cause significant problems for each software system.

Mo et al. (Mo et al., 2015) presented an empirical study of hotspot patterns that cause high maintenance costs. The aim of the study shows like these patterns not only identify the most error-prone and change-prone files, but also the root causes of bug-proneness and change-proneness in specific architecture problems. Tufano et al. (Tufano et al., 2015) presented an empirical investigation into when and why code smells are introduced in software projects. The study conducted over the commit history of open source projects demonstrated that most of the times the smells identified since their creation.

All of these studies were conducted at a significantly high-level scope than our work. Indeed, none of these studies considered the different types of Design Smells removals and refactoring at commits level.

3 STUDY DEFINITION AND PLANNING

The study aims to investigate the recurrence between refactoring actions and code smell in the source code, and also to understand if refactoring really contributes to the removal of Design Smells, or if, instead, both activities are used to improve the quality of the systems.

To conduct the empirical study, the complete historical evolution of five open-source Java projects has been considered: Atlas, Guice, Junit4, Log4j, and Zookeeper. Table I reports the number of commits, the relative number of files analysed, and the number of detected Design Smells.

To achieve the aforementioned objective, it has been necessary to identify to what extent the refactoring actions take place in the same files in which at least one code smell has been detected and if this happens with a greater or lesser percentage than other files. Therefore, the first research question is:

RQ1: *To what extent are smelly file more subject to refactoring actions?*

The occurrences of refactoring on smelly files and clean files, commits per commits, have been analysed, to understand if refactoring activities are more used in the commits changing smelly files.

The second objective was to understand if the refactoring actually contributes to the removal of smells code. The simultaneous occurrence of these two activities could be due to chance, for this reason, a more in-depth study was conducted to understand if it could be a true cause-effect relationship between them. Therefore, the second research question is:

RQ2: *To what extent do refactoring actions and Design Smells removals co-occur?*

Table 1: Characteristics of the studied Projects.

Project	**#Commits**	**#Files**	**#Design Smells**
Atlas	1580	13822	15383
Guice	1177	8590	7400
Junit4	1406	5444	5529
Log4j	2042	9235	9049
Zookeeper	1084	6216	8952

3.1 Data Extraction

To carry out this work, two datasets were built for each project.

The first dataset collects for each commit, at the class level, the presence or absence of each one of the 17 Design Smells considered in the study. Moreover, for each commit, a comparison with the previous commit has been performed to understand if someone of the detected smells has been removed or added, or if nothing has changed.

The Designite tool was used to detect the presence of the Design Smells. Designite is a software design quality assessment tool. It offers various features to support the identification of design problems that contribute to the decay of the software architecture. Table 2 lists the Design Smells that are detectable by Designite (Sharma et al., 2016).

The second dataset collects, for each commit, all the data related to the presence or absence of refactoring activities. For the detection of refactoring activities, version 2.0 of Refactoring Miner (Tsantalis et al., 2018) was used, which allows the identification of 40 different types of refactoring.

For each commit, trough Refactoring Miner, it has been extracted the list of refactoring actions performed in the commit, with the specification of the refactoring type and the source/target classes and methods involved. The complete dataset used in the empirical study was obtained from the merge of the two abovementioned datasets and provides information on the refactoring actions that occurred together with the removal of the code smells.

Table 2: Detected Design Smells.

Abstraction Design Smells	Imperative Abstraction Unnecessary Abstraction Multifaceted Abstraction Unutilized Abstraction
Encapsulation Design Smells	Deficient Encapsulation Unexploited Encapsulation
Modularization Design Smells	Broken Modularization Insufficient Modularization Hub-like Modularization Cyclically dependent Modularization
Hierarchy Design Smells	Wide Hierarchy Deep Hierarchy Multipath Hierarchy Cyclic Hierarchy Rebellious Hierarchy Missing Hierarchy

Figure 1 shows in detail the adopted process, highlighting the role of the adopted tools.

To address RQ1, the occurrences of refactoring that concurred with the removal of one or more code smells were counted.

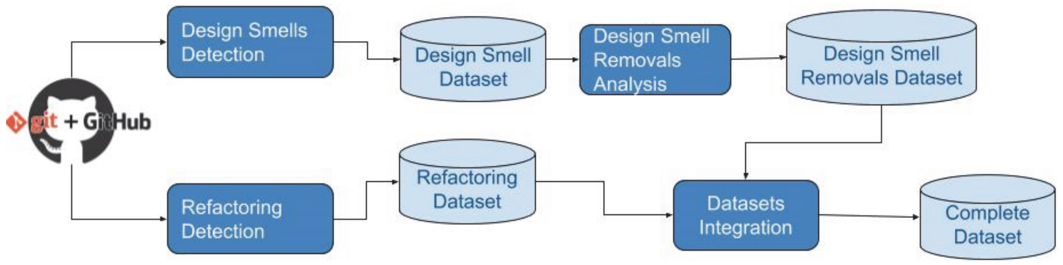


Figure 1: Toolchain of the Analysis Process.

To address RQ2, Fisher's exact test (Fisher, 1962) and Odds Ratio were used to compare the proportion of refactoring that occur together with the removal of code smells. More in detail, for each system, this analysis was carried out by considering the individual smells separately, then considering the categories of smells, and finally all the smells together.

4 EXPERIMENTAL RESULTS

This section reports the results achieved in our study and aims at answering our two research questions.

RQ1: *To what extent are smelly file more subject to refactoring actions?*

It has been evaluated how many times a refactoring activity has been applied on a file containing at least one smell. Figure 2 depicts the co-occurrence between Refactoring actions and Design Smells, comparing with the blue bars, the set of files, where both were contained, and with light blue bars the set of files with just refactoring inside.

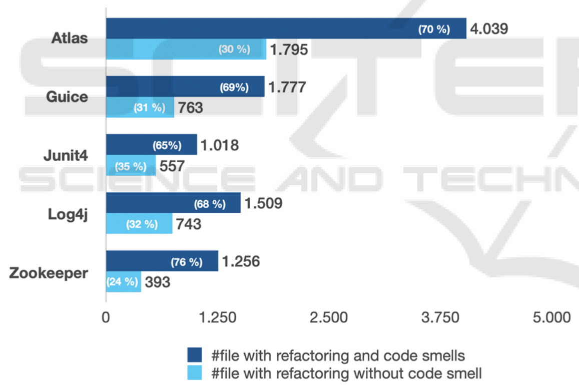


Figure 2: Number of commits with refactoring actions: smelly file and clean files.

For each analysed system, the occurrences of refactoring in the files have been measured together with the presence of Design Smells. As shown in Figure 2, the number of files containing both refactoring and at least one Design Smell is substantially higher in all the analysed systems.

More specifically, for Atlas, files with the presence of refactoring and Design Smell represent 70% of the cases, while files containing only refactoring represent only 30% of the cases, therefore less than half.

In the case of Guice, the percentages are very closed to those of Atlas.

The analysis of Junit confirms the trend, indeed also, in this case, the number of files containing both the refactoring and the Design Smell is greater, with a percentage of 65%, compared to the number of files containing only the refactoring which represents 35%. Results are similar also for the Log4j system, where the percentage of smells and refactoring co-occurrences is 68%, while the percentage of commits on files with just refactoring detected is 32%.

Finally, in the case of Zookeeper, the highest percentage is recorded. Here the number of files containing both activities is more than three times higher (76%) compared to the number of files with only the refactoring inside them (24%).

Finally, the results indicate that, in general, refactoring actions have much more chance of occurring inside files that contain at least one code smell.

RQ2: *To what extent do refactoring actions and Design Smells removals co-occur?*

This analysis aims to highlight a possible correlation between two main maintenance activities, that are refactoring and the removal of the Design Smells. In particular, the research objective is to understand if refactoring actions contribute in some way to the removal of Design Smells, or if both are generically used in the context of the source code improvement. Therefore, first of all, a comparison was made of each file related to a commit, containing a Design Smell, with the same file in the successive commit, to understand if that Design Smell has been removed or if it remains unchanged. Then, some statistical analysis has been conducted to perform a deep analysis of the occurrences. Fisher's exact test was used to analyse the correlation between refactoring and smells removal, to compare the proportion of refactoring activities that occurred together with the Design Smells removals.

Fisher's test was applied to 3 different couples of variables:

a) individual smells and refactoring;

b) smells by categories (as shown in Table 2) and refactoring;

c) all the smells on the whole project and refactoring.

From a statistical point of view, Fisher's exact test was used to understand if refactoring is most used in the files in which the removal of a code smell was detected, or not. In this specific case, every single smell was considered concerning refactoring.

Table 3: Proportion of refactoring involving Design Smell removal separately: Fisher's exact test *p*-value and Odds Ratio.

Code smell	**Atlas**		**Guice**		**Junit4**		**Log4j**		**Zookeeper**	
	p-value	Odds Ratio	*p*-value	Odds Ratio	*p*- value	Odds Ratio	*p*-value	Odds Ratio	*p*-value	Odds Ratio
Imperative Abstraction	1	0	1	0	1	0	1	0	0,29	7,41
Multifaced Abstraction	**0,036**	**4,43**	1	1,01	0,57	0	1	0	1	0
Unnecessary Abstraction	1	0	1	0	1	0	1	0	1	0
Unutilized Abstraction	**0,0098**	**0,66**	**0,001**	**0,47**	**0,000002**	**0,2**	0,47	0,87	1	0,96
Deficient Encapsulation	0,24	0,75	0,23	0,44	1	0,94	0,21	0,66	0,68	0,73
Unexploited Encapsulation	1	0	1	1,51	1	0	1	0	1	0
Broken Modularization	1	0	1	0	1	0	1	0	1	0
Cyclic-Dependent Modularization	0,78	1,07	0,28	0,76	1	1,02	0,62	0,78	**0,006**	**0,11**
Insufficient Modularization	0,17	1,3	**0,01**	**0,46**	0,24	1,9	0,06	0,46	0,63	0,7
Hub-like Modularization	1	0,86	0,11	0,19	1	0	1	0	**0,007**	**inf**
Broken Hierarchy	**0,02**	**0,42**	0,77	0,69	1	0,95	0,14	1,63	**0,01**	**0,12**
Cyclic Hierarchy	1	0	1	0	1	0	1	0	1	0
Deep Hierarchy	1	0	1	0	1	0	1	0	1	0
Missing Hierarchy	1	0	1	0,94	1	0	1	0	1	0
Multipath Hierarchy	0,44	0	1	0	1	0	1	0	1	0
Rebellious Hierarchy	1	0	1	0	1	0	1	0	1	0
Wide Hierarchy	1	0	1	0	1	0	0,16	15,19	1	0

Table 4: Proportion of refactoring involving smell code removal by category.

Code smell	**Atlas**		**Guice**		**Junit4**		**Log4j**		**Zookeeper**	
	p-value	Odds Ratio	*p*-value	Odds Ratio	*p*-value	Odds Ratio	*p*-value	Odds Ratio	*p*-value	Odds Ratio
Abstraction	**0,02**	**0,7**	**0,001**	**0,49**	**0,0000004**	**0,19**	0,43	0,87	1	0,995
Encapsulation	0,24	0,74	0,37	0,54	1	0,89	0,22	0,66	0,558	0,73
Modularization	0,25	1,18	**0,003**	**0,6**	0,24	1,54	0,065	0,61	0,101	0,52
Hieerarchy	**0,007**	**0,39**	0,389	0,59	1	0,88	0,22	1,53	**0,006**	**0,11**

Table 5: Proportion of refactoring involving smell code removal.

	p-value	Odds Ratio
Atlas	**0,001**	**0,74**
Guice	**0,0000003**	**0,52**
Junit4	**0,001**	**0,53**
Log4j	0,09	0,82
Zookeeper	**0,0007**	**0,54**

Fisher's exact test results are reported in Table 3, where it is shown that a few number of times the difference was statistically significant.

More in detail, in Atlas the test is significant in the case of Multifaced Abstraction, Unutilized Abstraction, and Broken Hierarchy. Only for the smell Multifaced Abstraction the Odds Ratio is greater than 1.

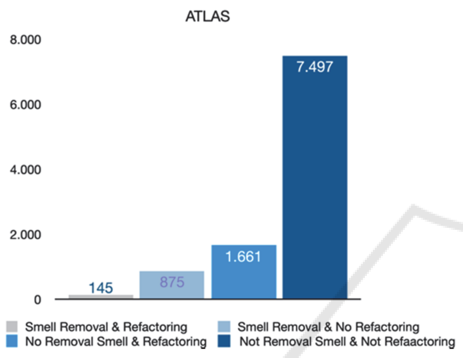
In Guice, it is statistically significant only in the case of Unutilized Abstraction, as previously, and Insufficient Modularization.

For JUnit4 only one significance is found, in the case of the smell Unutilized Abstraction.

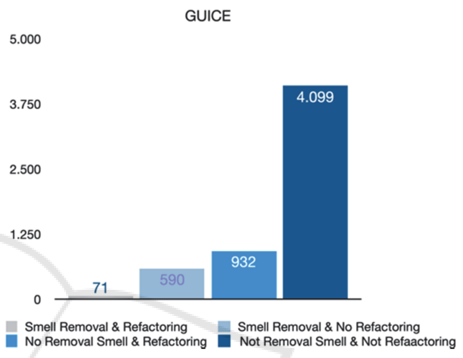
In the case of Log4j, no smell is statistically significant.

Finally, in Zookeeper, the following smells are statistically significant: Cyclic-Dependent Modularization, Hub-like Modularization, and Broken Hierarchy.

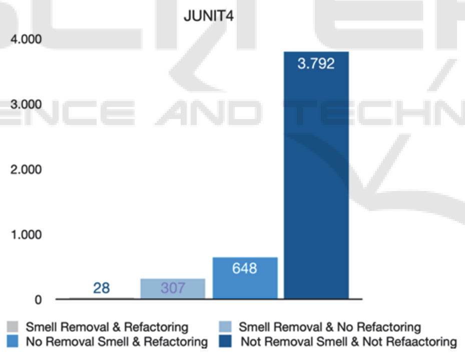
The analysis of the dependence between the presence of refactoring and the effective removal of smell shows that the two phenomena are not related. Only for some types of smells, there was a dependence between refactoring and elimination of smells, so it is possible to say that the elimination of smell is more likely in the absence of refactoring. Only in two particular cases, there was the opposite trend, a high probability of removing smell in the presence of refactoring (in Atlas for Multifaced Abstraction, and in Zookeeper for Hub-like Modularization). However, this evidence was only observed in two different projects and for two different types of smell designs.



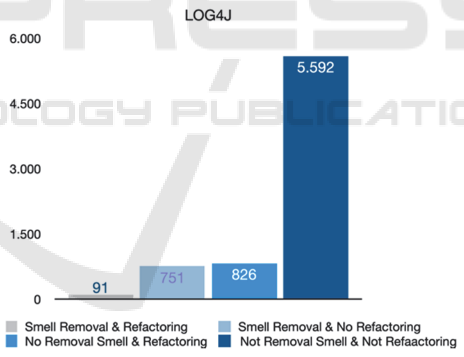
a)



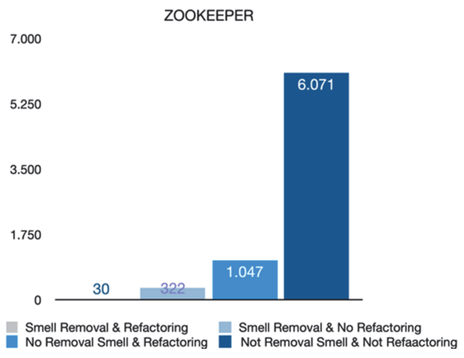
b)



c)



d)



e)

Figure 3: Number of files with and without refactoring actions: smelly file and clean files.

Another analysis considers smells by category.

As shown in Table 4, in the case of smells belonging to the Abstraction category, the difference is statistically significant for 3 projects (Atlas, Guice, and Junit4) out of 5.

For Encapsulation, no project shows a significant result, in line with the results obtained by considering the smells separately.

For the Modularization category, only Guice reports a statistical significance.

Finally, for the Hierarchy category, the significance is found only in the case of Atlas and Zookeeper.

In any case, we cannot speak of correlation between refactoring activities and removal of smells, because although in some cases the p-value is significant, the Odds Ratio is always less than 1.

The latest analysis considered all smells with refactoring activities on each project.

To better understand the data that was used in this analysis, Figure 3 shows a graph for each system.

The graphs use: i) grey bars to describe the number of files in which the removal of the smell was identified and there is presence of the refactoring, ii) the dark grey bars for the number of files in which the removal of the smell was identified and there is absence of the refactoring, iii) the blue bars for the number of files where the smell has not been removed but the presence of the refactoring has been identified, and finally iv) the dark blue bars to describe the number of files in which neither the smell has been removed and there is not the presence of refactoring.

Table 5 shows the results of the exact Fisher test.

As you can see, the difference was statistically significant for all projects, except for Log4j. However, even in this case, as in the previous one, the Odds Ratio is always less than 1.

5 THREATS TO VALIDITY

Threats to construct validity concern the relationship between theory and observation. Such threats mostly regard possible imprecision in our measurements. Indeed, the Design Smells removals belong to a dataset constructed and validated in this study, and we considered refactoring actions as detected by the Refactoring Miner tool. However, Tsantalis et al. show, for it, high precision ('98%) and recall ('87%) values (Tsantalis et al., 2018).

Threats to internal validity concerns factors internal to our study that can influence the results. In particular, we cannot, in general, claim a cause-effect relationship between refactoring actions and Design Smells removal. We mitigate this threat by analysing, in RQ2, refactorings occurred on smelly files.

Threats to conclusion validity concern the relationship between experimentation and outcome. We used an appropriate statistical test (Fisher's exact test) and effect size measure (Odds Ratio) to support our findings.

Threats to external validity concern the generalizability of our findings. This is a small, preliminary study, conducted on only 5 projects. Results might be confirmed or contradicted when analysing other projects.

6 CONCLUSIONS

The data collected for this study highlighted that in the software project analysed, almost 70% of refactoring is applied to classes affected by Design Smell. Despite this, the analysis of the dependence between the presence of refactoring activities and the effective removal of these smell shows that the two phenomena are not related.

Only for some types of smells, a co-occurrence has been observed between refactoring and removal of Design Smells, and surprisingly it refers that the elimination of Design Smell is high also in case of absence of refactoring.

In two particular cases, the opposite trend was observed, namely a high probability of removing smells in the presence of refactoring. However, this evidence was only observed in two different projects and for two different types of smell designs.

Future work aims at extending this work towards several directions. In particular, we may perform an in-depth quantitative and qualitative analysis of commits to study general quality improvement activities.

REFERENCES

L. Hochstein and M. Lindvall. *Combating architectural degeneration: A survey*. Information and Software Technology, 47(10):643–656, July 2005.

L. de Silva and D. Balasubramaniam. *Controlling software architecture erosion: A survey*. Journal of Systems and Software, 85(1):132–151, 2012.

J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. *Toward a catalogue of architectural bad smells.* In QoSA, volume 5581 of LNCS, pages 146–162. Springer, 2009.

F. Arcelli Fontana, I. Pigazzini, R. Roveda, D.A. Tamburri, M. Zanoni, and E. Di Nitto. *Arcan: A tool for architectural smells detection.* In 2017 IEEE ICSA

Workshops 2017, Gothenburg, Sweden, April 5-7, 2017, pages 282–285, 2017.

R. Mo, Y. Cai, R. Kazman, and L. Xiao. *Hotspot patterns: The formal definition and automatic detection of architecture smells*. 2015, 12th Working IEEE/IFIP Conf. on Software Architecture, pages 51–60, 2015.

M. Abbes, F. Khomh, Y.-G. Guéhéneuc, G. Antoniol, *An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension, 15th European Conference on Software Maintenance and Reengineering*, CSMR 2011, 1–4 March 2011, Oldenburg, Germany, IEEE Computer Society, 2011, pp. 181–190.

F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, G. Antoniol, *An exploratory study of the impact of antipatterns on class change- and fault-proneness*, Empir. Softw. Eng. 17 (3) (2012) 243–275.

F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, *On the diffuseness and the impact on maintainability of code smells: a large-scale empirical investigation*, Empir. Softw. Eng. (2017) 1–34.

D. I. K. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå. 2013. *Quantifying the Effect of Code Smells on Maintenance Effort*. IEEE Transactions on Software Engineering 39, 8 (Aug 2013), 1144–1156.

M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, *When and why your code starts to smell bad*, Proceedings of the *37th International Conference on Software Engineering* - Volume 1, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 403–414. [Online].

N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F.L. Meur, *Decor: a method for the specification and detection of code and design smells*, IEEE Transaction on Software Engineering, 36 (2010) 20–36.

D. Le and N. Medvidovic. *Architectural-based speculative analysis to predict bugs in a software system*. In Proceedings of *the 38th International Conference on Software Engineering Companion* (ICSE '16), pages 807–810, Austin, TX, USA, 2016.

F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, A. De Lucia, *Mining version histories for detecting code smells*, Softw. Eng. IEEE Trans. 41 (5) (2015) 462–489.

N. Tsantalis, A. Chatzigeorgiou, *Identification of move method refactoring opportunities*, IEEE Trans. Softw. Eng. 35 (3) (2009) 347–367.

M. Fowler, Refactoring: *Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

J. Brunet, R. A. Bittencourt, D. Serey, and J. Figueiredo. *On the evolutionary nature of architectural violations*. In *Reverse Engineering* (WCRE), 2012 19th Working Conference on. IEEE, 2012.

D. I. K. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå. 2013. *Quantifying the Effect of Code Smells on Maintenance Effort*. IEEE Transactions on Software Engineering 39, 8 (Aug 2013), 1144–1156.

M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, Denys Poshyvanyk. *When and Why Your Code Starts to Smell Bad*.

G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann, 1 edition, 2014.

Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. 2016. *Designite: a software design quality assessment tool. In Proceedings of BRIDGE '16, New York, NY, USA, 1–4.*

N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "*Accurate and efficient refactoring detection in commit history,*" in Proceedings of the *40th International Conference on Software Engineering*, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, 2018, pp. 483–494.

R. A. Fisher, "*Confidence limits for a cross-product ratio*," Australian Journal of Statistics, 1962.

M. Bernardi, M. Cimitile, *Reducing Static Dependences Exploiting a Declarative Design Patterns Framework*, in Proceedings of the 11th International Joint Conference on Software Technologies - Volume 2: ICSOFT-PT, ISBN 978-989-758-194-6, pages 154-160.