House of Cards: Code Smells in Open-source C# Repositories

Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis
Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{tushar,mfg,dds}@aueb.gr

***Abstract*—*Background:* Code smells are indicators of quality problems that make a software hard to maintain and evolve. Given the importance of smells in the source code's maintainability, many studies have explored the characteristics of smells and analyzed their effects on the software's quality.**

***Aim:* We aim to investigate fundamental characteristics of code smells through an empirical study on frequently occurring smells that examines inter-category and intra-category correlation between design and implementation smells.**

***Method:* The study mines 19 design smells and 11 implementation smells in 1988 C# repositories containing more than 49 million lines of code. The mined data are statistically analyzed using methods such as Spearman's correlation and presented through hexbin and scatter plots.**

***Results:* We find that *unutilized abstraction* and *magic number* smells are the most frequently occurring smells in C# code. Our results also show that implementation and design smells exhibit strong inter-category correlation. The results of co-occurrence analysis imply that whenever *unutilized abstraction* or *magic number* smells are found, it is very likely to find other smells from the same smell category in the project.**

***Conclusions:* Our experiment shows high average smell density (14.7 and 55.8 for design and implementation smells respectively) for open source C# programs. Such high smell densities turn a software system into a *house of cards* reflecting the fragility introduced in the system. Our study advocates greater awareness of smells and the adoption of regular refactoring within the developer community to avoid turning software into a *house of cards*.**

***Keywords*-Code smells, Implementation smells, Design smells, Code quality, Maintainability, C#.**

I. Introduction

Code smells indicate the presence of quality problems impacting mainly the maintainability of a software system [1], [2]. The presence of an excessive number of smells in a software system turns it into a *house of cards* which is hard to maintain and evolve.

Code smells and their impact on various aspects of software development have been explored and discussed extensively [3]–[6]. Many source code mining studies have examined code smells [7]–[9]. Further, the impact of smells on dimensions such as change-proneness [10], [11], defect-proneness [12], [13], and systems' quality [6] have been explored.

However, we observe the following gaps in the existing mining studies on smells:

- Existing mining studies on smells lack *scale*; the majority of the studies analyze a few subject systems (<10). To the best of our knowledge, Fontana et al. [6] analyzed 68 subject systems which are the maximum number of subject systems analyzed in this context. Generalizing a theory based on a few subject systems presents a considerable threat to validity.
- Existing mining studies also do poorly with respect to *breadth* of the experiment *i.e.,* number of smells analyzed in the study. Most of the existing mining studies consider a small sub-set of known smells. This under-analysis makes a mining study incomplete or even incorrect.
- Most of the mining studies on code smells are performed on the Java programming language. The under-representation of other programming languages makes us wonder whether the results of the existing mining studies are applicable in other similar languages.

In this paper, we aim to fill these gaps and reveal fundamental yet interesting characteristics of code smells in C# projects. These characteristics include — frequently occurring smells, inter-category and intra-category correlation between design and implementation smells, and the relationship of smell density with lines of code in each repository.

By exploring relationships between smell types we seek to find and comprehend models of smell occurrence and expansion. For instance, smell types with high probability may lead us to explore causal relationship among smells. Knowing the commonly occurring smells together may also be used to improve accuracy of smell detection mechanisms. This study can benefit software developers to help them understand the characteristics of relatively a wide range of smells and their potential implications. Raising the awareness is the first step towards cleaner code and less technical debt.

Based on the granularity and scope, smells can be classified as implementation smells [1], design smells [2], and architecture smells [14]. We limit our discussion of smells to implementation and design smells in this paper.

II. Overview of the study

We formulated the following research questions towards the quality analysis goal of C# projects.

RQ1. *What is the distribution of design and implementation smells in C# code?* We investigate the distribution of smells to find out whether there exists a set of implementation and design smells that occur more frequently *w.r.t.* another set of smells.

RQ2. *What is the relationship between the occurrence of design smells and implementation smells?* We study the degree of inter-category co-occurrence between design smells and implementation smells.

RQ3. *Is the principle of coexistence applicable to smells in C# projects?* It is commonly believed that patterns (and smells) co-exist [2], [15] *i.e.,* if we find one smell, it is very likely that we will find many more smells around it. We investigate the intra-category co-occurrences of a smell with other smells to find out whether and to what degree the folklore is true.

RQ4. *Does smell density depend on the size of the C# repository?* We investigate the relationship between the size of a C# project repository and associated smell density to find out how the smell density changes as the size of a C# project increases for both the smell categories. Smell density is a normalized metric that represents the average number of smells identified per thousand lines of code.

III. Study Design

In this section, we present the method that we employed to download and analyze source code repositories.

A. Mining GitHub repositories

We follow the procedure given below to select and download our subject systems.

1) We employ RepoReaper [16] to select curated GitHub repositories to download. RepoReaper assesses 10 dimensions (architecture quality, community, continuous integration, documentation, history, license, management, state (active or dormant), unit tests, and number of stars) of each repository and assigns a score corresponding to each dimension. We select all the repositories containing C# code where at least 8 out of 10 RepoReaper's dimensions have favourable score. We consider a score favourable if it has *true*, *active* (for the state dimension), assigned stars greater than 10, or value greater than zero for rest of the dimensions.
2) With this criterion, we download more than 2400 repositories. We are able to analyze 1988 repositories using Designite [17]. Some of the repositories couldn't be analyzed due to either missing external dependencies or custom build mechanism (*i.e.,* missing standard C# project files).
3) We exclude test code from the analysis since test code contains a different type of smells (*i.e.,* test smells [18]) which is not in the scope of this paper.

Table I summarizes the characteristics of the analyzed repositories. In the table, LOC, WMC, NC, and DIT refers to *Lines Of Code*, *Weighted Method per Class*, *Number of Children*, and *Depth of Inheritance Tree* respectively [19].

TABLE I
Characteristics of the Analyzed Repositories — Averages are computed per type; median LOC is computed over repository

Attributes	Values	Attributes	Values
Repositories	1,988	Median LOC	4391
Type declarations	436,832	Average methods	5.86
Method declarations	2,265,971	Average fields	2.42
Lines of code (C# only)	49,303,314	Average properties	2.12
		Average WMC	12.43
		Average NC	0.28
		Average DIT	0.31

B. Analyzing C# Repositories

We require a tool that detects a wide variety of design and implementation smells in C# code and at the same time allows us to perform smell mining on a large number of repositories automatically. We employ *Designite*[1] [17] (version 1.47.3) that supports detection of 19 design and 11 implementation smells. Further, it offers a customizable mechanism (*i.e.,* console application and capability to specify projects to be analyzed in a batch file with other required parameters) to automatically analyze C# code in each repository.

Tables II and III provide a brief description of detected design smells [2] and implementation smells [1], [20]–[22] respectively. We assign an acronym to each smell — design smell acronyms start with 'D' and implementation smell acronyms start with 'I' to make referencing easier.

C. Manual Verification

We chose two repositories randomly from the selected repositories and analyzed all the implementation and design smells detected by the tool manually. The selected repositories were *RestSharp*[2] (191KLOC) and *rtable*[3] (12KLOC). We analyzed 261 design smells and 1863 implementation smells detected by Designite in these repositories manually. We found 11 instances of false positives (10 instances belonging to *complex method* and one instance belonging to *long method* smell) in implementation smells category and 2 instances of false positives (both belonging to *unutilized abstraction*) in design smells category.

The tool generates false positive instances of *unutilized abstraction* majorly due to the project which is using the abstraction is not included in the analysis (for example a test project). The reason behind the generated false positive instances belonging to *complex method* smell can be traced back to slightly different algorithm used to compute cyclomatic complexity by the tool. In summary, the tool exhibits very low false positive rate and is suitable for a large scale mining study.

IV. Results and discussion

This section presents the results gathered from the analysis and our observations *w.r.t.* each research question addressed.

[1]http://www.designite-tools.com
[2]https://github.com/restsharp
[3]https://github.com/Azure/rtable

TABLE II
DESCRIPTION OF DETECTED DESIGN SMELLS AND THEIR DISTRIBUTION

Acronym	Design smell	Brief description	#Instances	Percentage
DBH	Broken Hierarchy	a supertype and its subtype conceptually do not share an "IS-A" relationship	20,332	4.8%
DBM	Broken Modularization	data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions	15,624	3.7%
DCM	Cyclically-dependent Modularization	two or more abstractions depend on each other directly or indirectly	52,436	12.5%
DCH	Cyclic Hierarchy	a supertype in a hierarchy depends on any of its subtypes	4,342	1.0%
DDH	Deep Hierarchy	an inheritance hierarchy is "excessively" deep	179	0.04%
DDE	Deficient Encapsulation	the declared accessibility of one or more members of an abstraction is more permissive than actually required	30,214	7.2%
DDA	Duplicate Abstraction	two or more abstractions have identical names or identical implementation	73,992	17.6%
DHM	Hub-like Modularization	an abstraction has high incoming and outgoing dependencies	676	0.2%
DIA	Imperative Abstraction	an operation is turned into a class	11,790	2.8%
DIM	Insufficient Modularization	an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, or implementation complexity	26,429	6.3%
DMH	Missing Hierarchy	conditional logic to explicitly manage variation in behaviour	2,598	0.6%
DMA	Multifaceted Abstraction	an abstraction has more than one responsibility assigned to it	1,236	0.3%
DMH	Multipath Hierarchy	a subtype inherits both directly as well as indirectly from a supertype	1,454	0.3%
DRH	Rebellious Hierarchy	a subtype rejects the methods provided by its supertype(s)	11,794	2.8%
DUE	Unexploited Encapsulation	client code uses explicit type checks	6,964	1.6%
DUH	Unfactored Hierarchy	there is unnecessary duplication among types in a hierarchy	20,962	5.0%
DUA	Unnecessary Abstraction	an abstraction that is actually not needed	44,583	10.6%
DTA	Unutilized Abstraction	an abstraction is left unused	90,786	21.6%
DWH	Wide Hierarchy	an inheritance hierarchy is "too" wide	3,140	0.7%

TABLE III
DESCRIPTION OF DETECTED IMPLEMENTATION SMELLS AND THEIR DISTRIBUTION

Acronym	Implementation smell	Brief description	#Instances	Percentage
ICC	Complex Conditional	a complex conditional statement	21,643	0.6%
ICM	Complex Method	a method with high cyclomatic complexity	95,244	2.5%
IDC	Duplicate Code	a code clone within a method	17,921	0.5%
IEC	Empty Catch Block	a catch block of an exception is empty	14,560	0.4%
ILI	Long Identifier	an identifier with excessive length	7,741	0.2%
ILM	Long Method	a method is excessively long	17,521	0.5%
ILP	Long Parameter List	a method has long parameter list	79,899	2.1%
ILS	Long Statement	an excessive long statement	462,491	12.4%
IMN	Magic Number	an unexplained number is used in an expression	2,993,353	80.0%
IMD	Missing Default	a switch statement does not contain a default case	23,497	0.6%
IVC	Virtual Method Call from Constructor	a constructor calls a virtual method	4,545	0.1%

RQ1. *What is the distribution of design and implementation smells in C# code?*

Approach: We compute the total number of detected smells for all the smells belonging to both implementation and design smell categories.

Results: Table II and III list the total number of instances detected for each smell. DTA (*unutilized abstraction*) and DCM (*cyclic-dependency modularization*) are the most frequently occurring design smells. On the other hand, DDH (*deep hierarchy*) is the least occurring design smell. To analyze it deeper, we computed the average number of children per class; a mere 0.28 indicates poor application of the principle of hierarchy in practice.

From the implementation smells side, IMN (*magic number*) and ILS (*long statement*) are the most frequently occurring smells. On the other hand, IVC (*virtual method call from constructor*) is the least occurring implementation smell. **We observe that analyzed C# code on an average contains one magic number per 16 lines of code.** It is surprizing to see a large number of *magic number* smells despite the fact that Designite excludes literals 0 and 1 while detecting the smell.

Interestingly, DDA **(*duplicate abstraction*) is one of the most frequently occurring design smells but** IDC **(*duplicate code*) is one of the least frequently occurring implementation smells.** It is because the scope of both the smells differs significantly; clones belonging to DDA occur anywhere in a project (but not in the same method) and clones belonging to IDC only occur within a method.

Implications: A high number of *unutilized abstraction* indicates that developers don't delete the obsolete code either knowingly or unknowingly. Such practices unnecessarily increase the cognitive load on the developers. Further, a large number of duplication also indicates poor application of the principle of abstraction and lack of refactoring.

RQ2. *What is the relationship between the occurrence of design smells and implementation smells?*

Approach: We compute the total instances of implementation and design smells in each repository. We then compute Spearman's correlation coefficient between the detected instances of implementation and design smells for each repository. We also compute total types of smells detected in each repository belonging to both the categories and compute

Spearman's correlation coefficient.

Results: Figure 1 presents a scatter graph showing the co-occurrence between total instances of detected implementation and design smells. The Spearman's correlation coefficient between number of implementation and design smells detected is 0.78059 (with p-value $< 2.2e-16$). **It shows that high volume of design (or implementation) smells is a very strong indication of the presence of high volume of implementation (or design) smells in a C# project.**

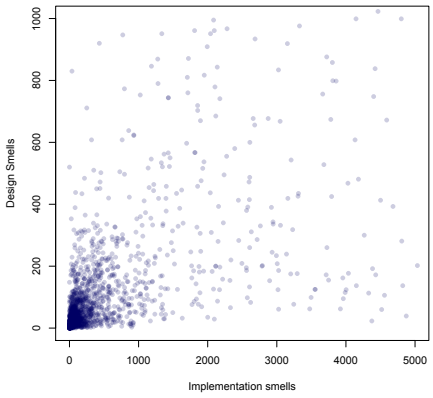

Fig. 1. Co-occurrence between detected implementation and design smell instances

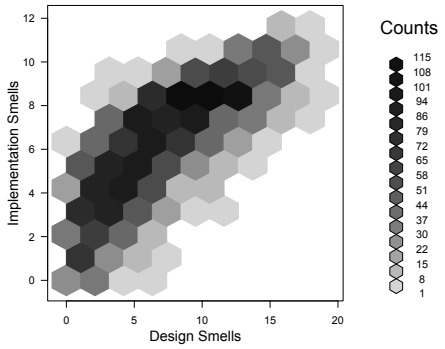

Fig. 2. Co-occurrence between detected implementation and design smell types

Further, we compute total types of smells, apart from total smell instances, detected in each repository belonging to both smells categories. Figure 2 shows a hexbin plot showing the co-occurrence between detected types of implementation and design smells. We get 0.80659 (with p-value $< 2.2e-16$) as the Spearman's correlation coefficient in this case. **It strongly suggests that as the types of detected implementation (or design) smells increases, the types of detected design (or implementation) smells also increases.**

Implications: Let us understand an implication of such a strong co-occurrence. Existing tools (such as NDepend[4] and SonarQube[5] majorly detect implementation issues. Due to this limitation, a software development team using these tools perceive only a limited set of quality issues and thus issues at higher granularities go unnoticed. Our results shows that the presence of implementation smells is a strong indication of design smells and thus the results emphasize the need to pay attention to smells at all granularities.

RQ3. *Is the principle of coexistence applicable to smells in C# projects?*

Approach: We compute the average intra-category co-occurrence for each smell. Co-occurrence is commonly used in the biogeography; we use the co-occurrence index used by Connor et al. [23]. The following equation computes co-occurrence coefficient C between smells $s1$ and $s2$.

$$C(s1, s2) = \frac{n1 \times n2}{N} \quad (1)$$

Here, $n1$ and $n2$ are the number of detected instances of smells $s1$ and $s2$ respectively. N is the total number of detected smells in the repository.

Results: Figures 3 and 4 present the average co-occurrence for each smell for both the smell categories respectively. DTA (unutilized abstraction) and DDH (deep hierarchy) show the highest and lowest co-occurrence respectively in the design smells category. Similarly, figure 4 shows that IMN (*magic number*) and IVC (*virtual method call from constructor*) exhibit the highest and lowest co-occurrence respectively in the implementation smells category.

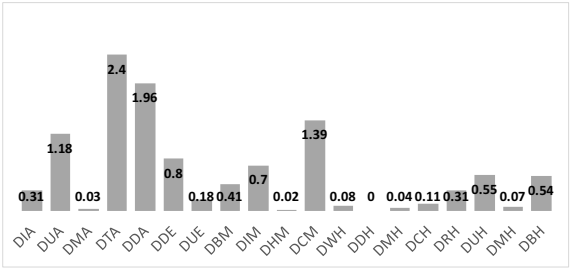

Fig. 3. Average co-occurrence (intra-category) for design smells

It implies that whenever *unutilized abstraction* or *magic number* smells are found in C# code, it is very likely to find other smells from the same smell category in the project. On the other hand, the smells *deep hierarchy* and *virtual method call from constructor* occur more independently. Co-occurrence of implementation smells (figure 4) show a large variation due to the huge difference in number of detected instances for each smell in the category.

Implications: These results reveal different smell expansion models where some smell types arise independently and others often occur as a group. This information is useful to developers when applying code refactoring with the purpose of removing

[4]http://www.ndepend.com/
[5]https://www.sonarqube.org/

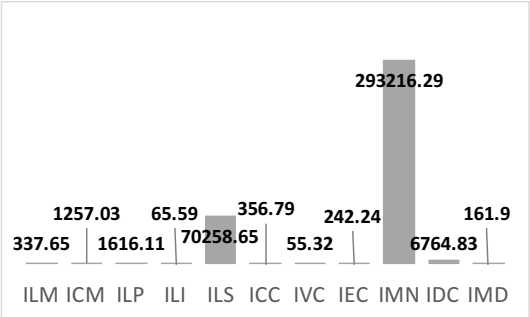

Fig. 4. Average co-occurrence (intra-category) for implementation smells

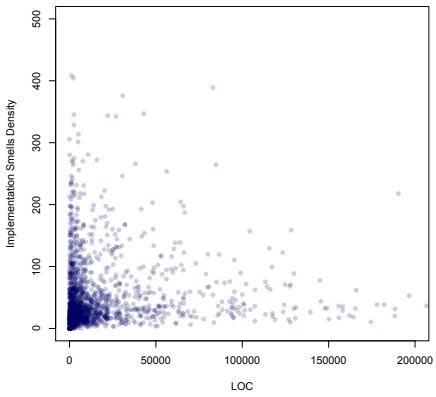

Fig. 5. Smell density for implementation smells against lines of code

smells because depending on the smell type they might have to expect other types of smells around it.

RQ4. *Does smell density depend on the size of the C# repository?*

Approach: We draw scatter plots between lines of code in a repository and the corresponding smell density for both the smell categories. We also computed Spearman's correlation coefficient for both the categories.

Results:

Figure 5 and figure 6 show the distribution of smell density for implementation and design smells against lines of code. A visual inspection of the above graphs shows that distribution in figure 5 is more scattered and random than the distribution shown in figure 6. We compute Spearman's correlation coefficient between implementation as well as design smell density and LOC. The analysis reports 0.27800 and -0.25426 as correlation coefficient (p-value $< 2.2e-16$) *w.r.t.* implementation and design smell density respectively. The results show a weak positive correlation for implementation smell density and weak negative correlation for design smell density with size of the project. Given the low values for both the coefficients, it is undecidable whether smell density depends on the size of the project.

Implications: According to the current study no strong correlation is evident between the number of smell occurrences and project size. If we assume that our sample is representative

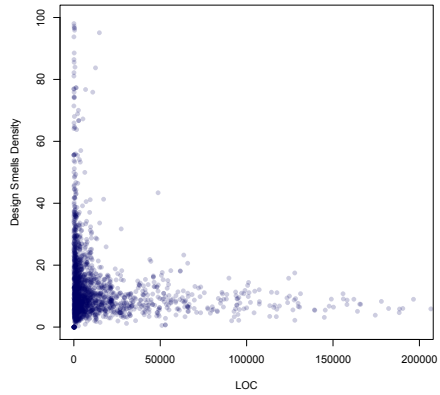

Fig. 6. Smell density for design smells against lines of code

of the population of C# projects then this means that big projects do not necessarily suffer from higher smell density.

V. Related Work

Code smells [1], [2] are symptoms of software quality issues that affect a project's maintainability and other quality attributes. Many aspects of smells have been investigated through empirical studies, such as the causes behind smells [3], the correlations between them [6], and the effect of refactoring actions on smells [9], [24]. The impact of smells on software quality has received considerable attention — some of the examined software quality dimensions are software maintenance [4], [5], change-proneness [10], [11], defect-proneness [12], [13], and systems' quality [6].

Code smells can manifest in different scopes, that is, architecture [14], design [2], and implementation [1]. Usually smells are studied in categories according to the scope they manifest in. In this paper we mine design and implementation code smells from approximately two thousand C# repositories using static analysis.

Yamashita et al. [4] explored the co-existence of smells within the same as well as across categories and revealed their impact on maintainability. Similarly, the co-existence of the *God class* and *God method* smells and their effect has been investigated by Abbes et al. [25]. The relationship between smell density and project size has been previously studied in the context of configuration management code [22].

Our study examines the intra-category and inter-category co-occurrence of smells and smell density over project size for 30 smells in 49 million lines of C# code. The posed research questions combined with the scale of the study and the breadth of the analyzed smells differentiate it from the related work.

Finally, there have been a few attempts to study code smells for the C# language [26], [27]. However, they lack both scale in terms of the number of projects they process and breadth of analysis in terms of the number of smells they examine.

VI. Threats to Validity

Construct validity concerns the appropriateness of the observations made on the basis of measurements. False positives are always associated with static code analysis and so are applicable to the tool that we employed in this study. However, our manual verification (Section III-C) shows that the tool exhibits very low false positive cases.

External validity concerns generalizability and repeatability of the produced results. Our study analyzes only open-source C# projects. Given that majority of existing research target Java [7], [9], our study complements the current literature.

VII. Conclusions

In this paper, we analyzed 1988 repositories containing more than 49 millions lines of C# code and detected 30 types of smells (19 design and 11 implementation smells). The goal of this study is to reveal basic characteristics concerning code smells in C# projects when the scale (*i.e.,* number of repositories) and the breadth (*i.e.,* number of detectable smells) of the analysis are large. We find that *unutilized abstraction* and *magic number* are the most frequently occurring design and implementation smells respectively. We observe a high degree of correlation between the number of detected instances of implementation and design smells. We find that smells *unutilized abstraction* and *magic number* show the highest co-occurrence among the other smells in their category. Finally, our analysis observes that smell density and lines of code in a C# project do not show a strong correlation.

A house of cards is analogous to a fragile system which is very difficult to change and extend. Our experiment shows average density of design smells 14.7 and implementation smells 55.8. Further, the highest recorded density for projects larger than 1000 LOC is 95 and 1893 for design and implementation smells respectively. A software system is turned into a house of cards with such high smell densities.

Acknowledgments

This work is funded by the SENECA project, which is part of the Marie Skłodowska-Curie Innovative Training Networks (ITN-EID) under grant agreement number 642954.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Programs*, 1st ed. Addison-Wesley Professional, 1999.

[2] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, 1st ed. Morgan Kaufmann, 2014.

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

[4] A. Yamashita and L. Moonen, "To what extent can maintenance problems be predicted by code smell detection? – An empirical study," *Information and Software Technology*, vol. 55, no. 12, pp. 2223–2242, 2013.

[5] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *The Journal of System and Software*, vol. 86, no. 10, pp. 2639–2653, Oct. 2013.

[6] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka, "Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains," in *2013 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, Sep. 2013, pp. 260–269.

[7] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, Jan. 2015.

[8] S. Fu and B. Shen, "Code Bad Smell Detection through Evolutionary Data Mining," in *International Symposium on Empirical Software Engineering and Measurement*. IEEE, Nov. 2015, pp. 41–49.

[9] T. Saika, E. Choi, N. Yoshida, S. Haruna, and K. Inoue, "Do Developers Focus on Severe Code Smells?" in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016, pp. 1–3.

[10] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, Aug. 2009, pp. 390–400.

[11] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness," in *2009 16th Working Conference on Reverse Engineering*, Ecole Polytechnique de Montreal, Montreal, Canada. IEEE, Dec. 2009, pp. 75–84.

[12] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, vol. 80, no. 7, pp. 1120–1128, Jul. 2007.

[13] L. Guerrouj, Z. Kermansaravi, V. Arnaoudova, B. C. M. Fung, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Investigating the relation between lexical smells and change- and fault-proneness: an empirical study," *Software Quality Journal*, pp. 1–30, May 2016.

[14] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying Architectural Bad Smells," in *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Mar. 2009, pp. 255–258.

[15] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, 1st ed. Wiley, 1996.

[16] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," preprint available at PeerJ Preprints 4:e2617v1 https://doi.org/10.7287/peerj.preprints.2617v1.

[17] T. Sharma, "Designite: A Customizable Tool for Smell Mining in C# Repositories," *10th Seminar on Advanced Techniques and Tools for Software Evolution, Madrid, Spain*, 2017.

[18] A. v. Deursen, L. Moonen, A. v. d. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, M. Marchesi, Ed. University of Cagliari, 2001, pp. 92–95.

[19] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, Jun 1994.

[20] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *Proceedings - 11th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2011*, Fondazione Bruno Kessler, Trento, Italy. IEEE, Nov. 2011, pp. 125–134.

[21] J. Bloch, *Effective Java (2Nd Edition) (The Java Series)*, 2nd ed. Prentice Hall PTR, 2008.

[22] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Workshop on Mining Software Repositories*, ser. MSR'16, 2016, pp. 189–200.

[23] E. Connor and D. Simberloff, "Species number and compositional similarity of the galapagos flora and avifauna," *Ecological Monographs*, no. 48, pp. 219–248, 1978.

[24] G. Czibula, Z. Marian, and I. G. Czibula, "Detecting software design defects using relational association rule mining," *Knowledge and Information Systems*, vol. 42, no. 3, pp. 545–577, Mar. 2015.

[25] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *15th European Conference on Software Maintenance and Reengineering*, ser. CSMR '11. IEEE Computer Society, 2011, pp. 181–190.

[26] S. Counsell, R. M. Hierons, H. Hamza, S. Black, and M. Durrand, "Exploring the eradication of code smells: An empirical and theoretical perspective," *Adv. Software Engineering*, vol. 2010, pp. 820 103:1–820 103:12, 2010.

[27] M. Gatrell, S. Counsell, and T. Hall, "Empirical Support for Two Refactoring Studies Using Commercial C# Software," in *Proceedings of the 13th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE'09. British Computer Society, 2009, pp. 1–10.