Removal of design problems through refactorings: are we looking at the right symptoms?

Andre Eposhi, Willian Oizumi
Campus Paranavai - IFPR
Paranavai, Brazil
{aheposhi,oizumi.willian}@gmail.com

Alessandro Garcia, Leonardo Sousa
Informatics Department – PUC-Rio
Rio de Janeiro, Brazil
{afgarcia,lsousa}@inf.puc-rio.br

Roberto Oliveira, Anderson Oliveira
Informatics Department – PUC-Rio
Rio de Janeiro, Brazil
{rfelicio,aoliveira}@inf.puc-rio.br

***Abstract*—A design problem is the result of design decisions that negatively impact quality attributes. For example, a stakeholder introduces a design problem when he decides to addresses multiple unrelated responsibilities in a single class, impacting the modifiability and reusability of the system. Given their negative consequences, design problems should be identified and refactored. The literature still lacks evidence on which symptoms' characteristics can be used as strong indicators of design problems. For example, it is unknown if the density and diversity of certain symptoms (e.g., violations of object-oriented principles) are correlated with the occurrence of design problems. Thus, in this paper, we report a case study involving two C# systems. We evaluated the impact of refactoring, focused on removing design problems, on the density and diversity of symptoms. Results indicate that refactored classes usually present higher density and diversity of symptoms. However, the density and diversity of some symptoms, such as the violation of object-oriented principles, was not predominantly higher in refactored classes. Moreover, contrary to our expectations, refactorings caused almost no positive impact on the density and diversity of symptoms.**

***Index Terms*—design problem; design smell; technical debt; refactoring; code smell**

I. Introduction

A software system must provide value through its functionalities and should satisfy a set of quality attributes, such as maintainability, reliability, and efficiency [1], [2]. Neglecting quality attributes can lead to the introduction of design problems [3]–[5]. A design problem is the result of stakeholders' decisions that negatively impact the quality attributes [3]–[5]. The degradation of quality attributes, in the form of design problem, causes negative consequences such as massive refactoring, or even the software discontinuation [6]. Hence, developers have to identify and remove them as early as possible.

The identification of design problems is far from trivial [7]–[11]. It usually occurs based on the localization of symptoms such as code smells [12], [13] and violation of object-oriented principles [14]. In fact, developers tend to combine multiple symptoms to identify design problems in practice [7], [15]–[17], in which they use the density and diversity of symptoms during the identification [7], [18]. Density indicates the number of symptom instances affecting a code element (e.g., class), and diversity indicates how many different types of symptoms a code element contains.

Even though previous studies found that developers rely on and combine multiple symptoms [7], [19], they did not investigate to what extent these multiple symptoms manifest in elements affected by design problems. For instance, some type of symptom may appear homogeneously in all classes, whether they are affected by design problems or not. Therefore, these studies may be misguiding other studies that want to investigate how to support developers in identifying design problems.

To address this matter, we investigated 1,468 classes from two C# systems. First, we identified the classes with design problems, in which we searched for refactoring tasks that were aimed at removing design problems. Second, we collected different symptoms in the refactored classes. Third, we analyzed whether refactored classes presented more or less symptoms and symptom types when compared to other classes in the system. Finally, we evaluated the impact of refactoring tasks on the density and diversity of symptoms. Our results indicate that code smells and internal attributes, such as coupling and complexity, can be strong indicators of design problems. However, the density and diversity of some symptoms, such as the violation of object-oriented principles, were not predominantly higher in refactored classes. In addition, unexpectedly, refactorings caused almost no positive impact on the density and diversity of symptoms.

II. Background

Design problems negatively impact quality attributes [3]–[5], [20], [21]. For example, a design problem related to reusability may cause consequences such as code duplication and rework. According to the ISO/IEC 25010 standard [22], there are eight main quality attributes that should be analyzed when evaluating software quality. They should be addressed to systems meet the stakeholders' expectations. In this study, we restricted our analyses to quality attributes that appeared consistently in our target systems during the refactoring tasks. They are Efficiency, Compatibility, Reliability, and Maintain-

This work was supported by CNPq (grants 153363/2018-5, 434969/2018-4, 140919/2017-1, and 312149/2016-6), CAPES (grant 175956) and FAPERJ (grant 22520-7/2016).

ability. Detailed description about them is available in our replication package[1].

Identifying a design problem in a system is difficult, especially when the source code is the only available artifact. Given the typical lack of documentation [23], developers have to rely on certain indicators in the program, the so-called **symptoms** [19], to identify and remove design problems. In this study, we used three symptoms that developers have been using frequently in the practice. **Code smell** is the first symptom, which is a surface indicator of possible design problem [12]. An example of code smell type is the *God Class*, which indicates classes that are long and excessively complex.

The second symptom that we used is the violation of object-oriented principles [14]. **Principle violation** indicates the violation of object-oriented design characteristics, such as abstraction, encapsulation, modularity, and hierarchy. An example of object-oriented principle is the *Single Responsibility Principle*, which determines that each class should have a single responsibility in the system [14]. **Internal attributes' violation** is the third symptom, which indicates a violation of characteristics that are considered fundamental for software design, such as coupling, cohesion, and complexity. In this study, we opted for using coupling and complexity as internal attributes. *Coupling* indicates the number of classes that a single class uses [24], and cyclomatic complexity (*complexity* for short) measures the structural complexity of the code [25]. We opted for not collecting cohesion because how to appropriate measure it is still challenging.

Refactoring is a popular technique to remove design problems from a system. *Refactoring* consists in transforming the source code structure without changing its functional behaviour [12], [26]. We consider that refactoring is any sequence of source code changes to improve quality attributes. For instance, to remove a design problem that impacts the reusability usually requires refactorings that improve abstractions based on object-oriented principles such as the *Dependency Inversion Principle* and the *Single Responsibility Principle* [14].

III. Study Design

Goal and Research Questions. Several studies (e.g., [27]–[29]) have proposed and evaluated techniques for the identification of design problems. Nevertheless, in practice, most of them are not applied by developers. One of the issues of existing techniques is the high amount of false positives, which leads developers to have little confidence in the presented symptoms. Another problem is that most techniques are based on a single type of symptom. Nevertheless, according to the literature, developers may combine multiple and diverse symptoms for confirming the existence of a design problem. However, unlike what was observed in the study of Sousa et al. [7], existing techniques only combine symptoms of the same type (e.g., code smells). In addition, their efficiency for revealing classes impacted by design problems has not been exhaustively validated. Finally, there is little evidence on the impact of refactoring on design problem symptoms. Thus, in this paper, *we aim at evaluating the relation of design problems with the occurrence of multiple and diverse symptoms*. To achieve our goal, we defined two research questions:

RQ1. Are the density and diversity of symptoms in refactored classes different from the density and diversity in other classes?

With RQ1, we aim at understanding if the design problem symptoms are denser and more diverse in refactored classes when compared to other classes. As developers usually refactor the classes in which they perceive the presence of design problems, we need to know if such classes, before being refactored, present higher density and diversity of symptoms than most regular classes. Answering this question will be helpful for evaluating whether combining multiple and diverse symptoms is indeed an effective strategy for identifying and confirming the existence of design problems.

RQ2. What is the impact of refactoring on symptoms of design problems?

With RQ2, we want to observe if removing design problems, through refactoring, impacts the density and diversity of symptoms. This question will help us to understand if symptoms disappear or decrease after developers try to remove design problems. Based on the answer to this question, it will be possible to improve existing detection techniques by focusing on the (types of) symptoms that often decrease or disappear after refactoring a design problem.

To answer our research questions, we conducted a case study involving two C# systems. We collected and analyzed refactoring tasks that were exclusively intended to remove design problems. Based on manual analyses, we categorized the tasks according to quality attributes that should be improved by removing the design problem. After that, we collected multiple types of design problem symptoms and conducted our data analysis. Below we provide details about the target systems and about our procedures for data collection and analysis.

Target Systems. We selected two C# software systems for conducting our case study: OpenPOS and UniNFe. OpenPOS is a desktop system that provides sales features, such as sales registration and cashier closing. OpenPOS has 97 Kilo Lines of Code (KLOC) and 3,318 commits in the control version system. UniNFe is a background service that sends and receives Brazilian electronic invoices. UniNFe has 492 KLOC and 2,373 commits. These projects are suitable for this study because they present more than two years of source code history – which is registered in tasks and commits, and we have full access to their developers for questions and clarifications.

Data Collection and Analysis. We followed five main steps for data collection and analysis: (1) finding tasks aimed at removing design problems, (2) analyzing tasks for discarding

[1]http://wnoizumi.github.io/ICPC2019/

those that are unrelated to the removal of design problems, (3) classifying tasks according to the improved quality attribute, (4) collecting information about design problem symptoms, and (5) running data analysis. Next we present details about each step.

Task Search and Filtering. In the first and in the second steps, we selected the tasks of each target project that were intended to remove design problems. To achieve this goal, we asked two developers of each project to provide us with a list of tasks aimed at removing design problems through refactoring. After that, we conducted an automated search in the issue tracking system to complement the lists of tasks provided by stakeholders. Our automated search was based on a set of keywords that are often associated with design problems (e.g., structure, interface, and duplicate). The full list of keywords used in this search are presented in our replication package. We have defined those keywords based on the analysis of task descriptions from 50 open source projects. These keywords often occur in the description of tasks that aim at removing design problems. After the automated search with the keywords, we analyzed the resulting list of tasks and discarded those unrelated to the removal of design problems.

Classifying Tasks According to Quality Attributes. In the third step, we analyzed and classified each design problem removal task according to the improved quality attribute. For this classification, we considered the intention of stakeholders as manifested in each task description. After identifying the quality attribute the task was intended to improve, we checked whether there was any obvious discrepancy between the task description and the actual changes made. In such cases, the tasks were not included in our analysis. Based on these procedures, we selected a total of 33 refactoring tasks.

Collecting Information about Design Problem Symptoms. We collected three types of design problem symptoms, which are: code smells, principle violations, and internal attributes. As explained in Section II, we selected coupling and complexity as representatives of internal attributes. Such symptoms were collected for all classes in the systems, before and after each refactoring task. For collecting those symptoms, we used two tools: the Visual Studio Community 2017 [30] and the Designite tool [31]. Detailed descriptions, sub-types, detection strategies, and thresholds for all types of symptoms are available in our replication package. We did not collect the symptoms in isolation for each refactoring task. Instead, we collected the symptoms in the last stable revision of the system before the refactorings and in the first stable revision after the refactorings. We are aware that this approach causes refactoring changes to blend with changes from other tasks. Nevertheless, we have chosen this approach intentionally, since the changes performed in our target systems are not commited to the repository individually. Their developers usually make changes to the repository available only after performing multiple tasks.

Running Data Analysis. After collecting data about tasks, source code changes and symptoms, we conducted data analysis for answering our research questions. For answering RQ1, we classified the classes of each analyzed version into two groups: refactored classes and other classes (or simply, others). The former group contains all classes that were refactored for removing one or more design problems. The latter contains all other classes in the system.

For both groups, we calculated the mean density of symptoms. For code smells and principle violations, the density was considered as the number of individual instances of symptoms occurring in a class. For coupling and complexity, since they are individual numeric values, we considered the density as being the value itself. Thus, the higher the value of coupling or complexity, higher will be the density of such symptoms. We compared the density of both groups by computing the mean number of individual instances for code smells and principle violations, and the mean values for coupling and complexity. Based on this information, we checked whether there was a significant difference in the means presented by both groups. To compare the diversity between the two groups, we considered only code smells and principle violations. We considered a set of 11 sub-types of code smells and 18 sub-types of principle violations, which are the ones detected by the Designite tool. We compared the diversity of symptoms by calculating the mean number of different sub-types of symptoms per class in each group. Diversity analysis was not performed for coupling and cohesion because they are only represented by numeric values, instead of symptom instances.

Finally, for answering RQ2, we collected and compared the same data used for answering RQ1. The difference here is that we compared the density and diversity of classes before and after the execution of refactoring tasks. Thus, for this question we only considered classes that were changed by refactoring tasks. For both research questions, we applied statistical tests using the R Project tool [32].

IV. Results

A. No Difference in the Symptoms of Refactored Classes?

Since principle violations are strongly linked with software design quality, we expected this type of symptom to present significant differences in refactored classes. **However, contrary to our expectations, principle violation was the type of symptom for which the difference was smaller in both systems.** Table I shows the mean density of symptoms in the classes of both groups (refactored and others). Each line shows, for a symptom type, the mean density of symptoms in refactored and in other classes. This information is provided for each system (columns two to five) before and after refactorings. For the OpenPOS system, the mean number of principle violations for refactored classes is 0.80 while for other classes is 0.75. The difference is a little bit higher in the UniNfe system, being 2.23 for refactored classes and 1.13 for other classes, which may be still considered a small difference. This was not expected because our anecdotal knowledge suggests that classes with design problems tend to have more design-related symptoms than other classes.

For code smells, coupling, and complexity, we observed a notable difference when comparing refactored classes with

TABLE I
MEAN DENSITY OF SYMPTOMS IN REFACTORED CLASSES AND IN OTHERS

Symptom Type	Mean Density			
	OpenPOS		UniNFe	
	Refactored	Others	Refactored	Others
Before Refactoring Tasks				
Principle Violations	0.80	0.75	2.23	1.13
Code Smells	**35.84**	4.10	**41.64**	1.41
Coupling	46.54	18.62	37.61	11.32
Complexity	46.39	13.60	**124.74**	16.97
After Refactoring Tasks				
Principle Violations	0.72	0.76	2.40	1.49
Code Smells	**35.46**	3.76	**41.35**	1.71
Coupling	49.05	17.60	39.07	11.75
Complexity	47.83	12.42	**133.81**	16.85

TABLE II
MEAN DIVERSITY OF SYMPTOMS IN REFACTORED CLASSES AND IN OTHERS

Symptom Type	Mean Diversity			
	OpenPOS		UniNFe	
	Refactored	Others	Refactored	Others
Before Refactoring Tasks				
Principle Violations	0.65	0.70	1.16	0.81
Code Smells	**1.58**	0.44	**3.28**	0.19
After Refactoring Tasks				
Principle Violations	0.60	0.72	1.14	1.07
Code Smells	**1.38**	0.42	**3.52**	0.24

others. This difference indicates the density of such symptoms may be used as a strong indicator of design problems. The type that stood out most in both systems was the code smell. For both systems, the density of code smells was more than 8 times higher in refactored classes.

We applied the *Mann-Whitney-Wilcoxon test* to check whether there is a true difference in the density distribution. This statistical test indicated, with a confidence level of 95% and p-value smaller than 0.0001, that the distribution of density of code smells in refactored classes is different from the distribution of density of code smells in other classes. The raw data and the detailed results of this statistical test are available in our replication package. We observed that code smell density for refactored classes is often higher than the code smell density for other classes in the system. We also observed several outliers in the distribution of code smells for others. Many of these outliers may be classes affected by design problems that were not refactored by developers.

Diversity of symptoms is also more significant for code smells. Table II shows the diversity of symptoms for code smells and for principle violations. This table follows an organization similar to Table I, providing the mean diversity of each symptom type. In both systems, the diversity of code smells was significantly higher in refactored classes when compared to other classes in the systems. For refactored classes, the diversity of code smells was more than three times higher in OpenPOS and more than seventeen times higher in UniNFe. The *Mann-Whitney-Wilcoxon test* indicates, with a confidence level of 95% and p-value smaller than 0.0001, that the diversity mean of code smells in refactored classes is different from the diversity mean of code smells in other classes. On the other hand, when we run the same test for the diversity of principle violations, with a p-value of 0.09313, we can not reject the null hypothesis for the OpenPOS system. Therefore, the diversity mean of principle violations in refactored classes may be equal to the diversity mean of principle violations in other classes.

The diversity of principle violations is irrelevant for individual classes. In this case study, diversity of principle violations was shown to be unrelated to refactored classes. When compared to other classes, the difference in diversity was of 0.05 for OpenPOS and of 0.35 for UniNFe. In addition, in no case was the diversity of principle violations higher than 1.2. However, this does not mean that principle violations are useless for identifying design problems. Although density and diversity of principle violations are not determining factors for the existence of design problems, principle violations may be combined with other symptoms for filtering key-classes of the system that need improvements.

Therefore, based on our analyzes, we conclude that **the density and diversity of symptoms in refactored classes are, indeed, different from the density and diversity of symptoms in other classes.** However, principle violations, the type of symptom we thought would be the most relevant to design problems, did not show significant differences for both density and diversity of symptoms. Thus, for practitioners, the density and diversity of code smells may be considered a more reliable indicator of design problems. In addition, if considered together with other symptoms they tend to be even more relevant. For example, we observed that, besides presenting high density of code smells, some refactored classes also presented high values for coupling and complexity. Combining such symptoms may be helpful for finding design problems that tend to grow with the system evolution.

B. Does Refactoring Reduce Symptoms?

To answer our second research question, we compared the density and diversity of symptoms in refactored classes, before and after being refactored. For density, Table I shows the mean density of symptoms before and after the application of refactorings. It is possible to observe that refactorings caused little impact in all types of symptom. In some cases the mean value increased while in others it decreased after the refactorings. However, for none of them there was a significant difference. To confirm whether there is a significant difference between the two groups – before refactoring and after refactoring –, we applied the Mann-Whitney Wilcoxon test. This statistical test allowed us to check if there is a difference between the mean of number symptoms before refactoring and the mean number of symptoms after refactoring. A p-value higher than 0.05 indicates the null hypothesis can not be rejected. In this case, the null hypothesis is: *"true location shift is equal to 0"*.

This statistical test revealed p-values of 0.5950 for Principle Violation, 0.6717 for Code Smells, 0.6043 for Coupling, and 0.7383 for complexity. Thus, it is possible to observe that the p-value was much higher than 0.05 for all symptom types, indicating that we can not reject the null hypothesis. Based on such result, we conclude that refactorings applied in the systems of this case study did not reduce the density of any of the four symptom types.

Regarding diversity, we carried out a similar analysis. As presented in Table II, we computed the mean number of different categories of code smells and of principle violations. In this case, again, the difference was marginal for both symptom types. The mean diversity for principle violations was reduced in both systems. Nevertheless, it is not possible to affirm diversity always reduces after refactorings because the difference was too small. With respect to code smells, we did not observe similar trends for both systems. In OpenPOS, there was a reduction in the diversity of code smells. On the other hand, in UniNFe, we observed increased diversity of code smells. Thus, for both types of symptom, we were unable to observe a clear relation between refactoring and diversity of symptoms. As we could not observe a significant and consistent reduction in the density or in the diversity of symptoms, the answer of our second research question RQ2 is that **refactorings cause little to no impact in symptoms of design problems**.

V. Threats to Validity

The first threat to validity is regarding our sample size. We are aware that analyzing data from two systems is not enough for finding generalizable results. We tried to mitigate this threat by selecting systems with different characteristics. Another threat is related to the method we used for finding the refactoring tasks. We may have missed tasks that were not remembered by the stakeholders or did not contain the searched keywords. To mitigate this threat, we asked for, at least, two developers of each system to provide a list of design problem removal tasks. All participating developers have knowledge about design problems and have worked in the systems since their inception.

Still related to the refactoring tasks, it is possible that a task description demonstrates the intention to remove a design problem but this does not occur in practice. We partially mitigate this threat checking whether there was any obvious discrepancy between the task description and the actual changes made. Moreover, we collected and analyzed multiple types of symptoms before and after refactorings.

There is another threat related to the tools used for detecting symptoms. Aspects such as precision and recall may have influenced the results of this study. We tried mitigate this threat by selecting a consolidated state-of-the-practice tool (Visual Studio) and the only tool we know that is capable of detecting principle violations in C# systems (Designite). Designite has been used successfully in other studies [31], [33].

Finally, there is a threat regarding reproducibility. Despite being open source, the maintenance of our target systems is controlled by a company. Therefore, we are not allowed to publish detailed information about the tasks and about the issue tracking system. In order to mitigate this threat, we created a replication package containing, among other information, the identification and the category of design problems removed by each task. In addition, the source code of both systems is fully available at the SourceForge repository[2,3].

VI. Related Work

Some studies investigated the use of symptoms for identifying design problems [13], [17], [28], [34]–[36]. Sousa's et al. [7], for example, found that developers use multiple symptoms to diagnose design problems in practice. Unfortunately, they did not investigate whether refactored classes had more symptoms than other classes. Mamdouh and Mohammad [33] investigated if design problem symptoms are removed as the system evolves. They found that the density of symptoms undergo few changes throughout the systems evolution. We conducted a similar analysis in our research question RQ2. Nevertheless, our study was focused on refactored classes, while they analyzed all classes indistinctly.

The impact of refactoring in symptoms was also previously studied [37]–[39]. For example, Bavota et al. [37] investigated whether refactorings occur in classes with symptoms such as the code smells. They found that none of the investigated symptoms are strong indicators of refactoring. Unlike them, we found that symptoms such as code smells, cohesion and complexity are strong indicators of the need for refactoring. Similar to us, they also observed that smells are not usually removed by means of refactorings. However, we are the first to investigate the impact of refactorings on principle violations. Cedrim et al. [39] also investigated the impact of refactorings on code smells. Nevertheless, they did not verify whether density and diversity of symptoms in refactored classes are different from non-refactored classes.

VII. Conclusion

In this paper, we investigated whether design problem symptoms appear with higher density and diversity in classes refactored by developers. We also investigated if refactoring tasks have a positive impact on the density and diversity of symptoms. To achieve our goal, we conducted a case study involving two C# open source systems. Our results indicate that refactorings caused almost no positive impact on the density and diversity of any type of symptom. Nevertheless, we also observed that refactored classes have higher density of code smells, coupling and complexity, when compared to other classes in the systems. This result indicates that, despite not being removed by refactorings, some types of symptom may be indeed strong indicators of design problems. Based on our observations, as future works, we intend to (1) replicate this study with more systems and considering more symptom types, (2) conduct a qualitative evaluation to better understand our results, and (3) propose a semi-automated technique to help developers in avoiding design problems during software development.

[2]https://sourceforge.net/projects/openposbr/

[3]hhttps://sourceforge.net/projects/uninfe/

References

[1] J. Offutt, "Quality attributes of web software applications," *IEEE Softw.*, vol. 19, no. 2, pp. 25–32, Mar. 2002. [Online]. Available: https://doi.org/10.1109/52.991329

[2] I. Gorton, *Essential software architecture*. Springer Science & Business Media, 2006.

[3] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, no. C, pp. 193–220, Mar. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2014.12.027

[4] E. Lim, N. Taksande, and C. Seaman, "A balancing act: What software practitioners have to say about technical debt," *IEEE Software*, vol. 29, no. 6, pp. 22–27, Nov 2012.

[5] T. Besker, A. Martini, and J. Bosch, "Time to pay up: Technical debt from a software quality perspective," in *Proceedings of the XX Iberoamerican Conference on Software Engineering, Buenos Aires, Argentina, May 22-23, 2017.*, 2017, pp. 235–248.

[6] A. MacCormack, J. Rusnak, and C. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Manage. Sci.*, vol. 52, no. 7, pp. 1015–1030, 2006.

[7] L. Sousa, A. Oliveira, W. Oizumi, S. Barbosa, A. Garcia, J. Lee, M. Kalinowski, R. de Mello, B. Fonseca, R. Oliveira, C. Lucena, and R. Paes, "Identifying design problems in the source code: A grounded theory," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 921–931. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180239

[8] W. Oizumi, L. Sousa, A. Oliveira, A. Garcia, A. B. Agbachi, R. Oliveira, and C. Lucena, "On the identification of design problems in stinky code: experiences and tool support," *Journal of the Brazilian Computer Society*, vol. 24, no. 1, p. 13, Oct 2018. [Online]. Available: https://doi.org/10.1186/s13173-018-0078-y

[9] R. Oliveira, L. Sousa, R. de Mello, N. Valentim, A. Lopes, T. Conte, A. Garcia, E. Oliveira, and C. Lucena, "Collaborative identification of code smells: A multi-case study," in *39th ICSE, SEIP Track*, 2017, pp. 33–42.

[10] R. Oliveira, B. Estacio, A. Garcia, S. Marczak, R. Prikladnicki, M. Kalinowski, and C. Lucena, "Identifying code smells with collaborative practices: A controlled experiment," in *10th SBCARS*, 2016, pp. 61–70.

[11] R. M. d. Mello, R. F. Oliveira, and A. F. Garcia, "On the influence of human factors for identifying code smells: A multi-trial empirical study," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Nov 2017, pp. 68–77.

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

[13] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proceedings of the 5th international symposium on Software visualization; Salt Lake City, USA*. ACM, 2010, pp. 5–14.

[14] R. C. Martin and M. Martin, *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.

[15] M. Abbes, F. Khomh, Y. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Proceedings of the 15th European Software Engineering Conference; Oldenburg, Germany*, 2011, pp. 181–190.

[16] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, "Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems," in *AOSD '12*. New York, NY, USA: ACM, 2012, pp. 167–178.

[17] W. Oizumi, A. Garcia, L. Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *The 38th International Conference on Software Engineering; USA*, 2016.

[18] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos, "Identifying architectural problems through prioritization of code smells," in *SBCARS16*, Sept 2016, pp. 41–50.

[19] L. Sousa, R. Oliveira, A. Garcia, J. Lee, T. Conte, W. Oizumi, R. de Mello, A. Lopes, N. Valentim, E. Oliveira, and C. Lucena, "How do software developers identify design problems?: A qualitative analysis," in *Proceedings of 31st Brazilian Symposium on Software Engineering*, ser. SBES'17, 2017.

[20] W. Oizumi, A. Garcia, T. Colanzi, M. Ferreira, and A. Staa, "When code-anomaly agglomerations represent architectural problems? An exploratory study," in *Proceedings of the 2014 Brazilian Symposium on Software Engineering (SBES); Maceio, Brazil*, 2014, pp. 91–100.

[21] W. Oizumi, A. Garcia, T. Colanzi, A. Staa, and M. Ferreira, "On the relationship of code-anomaly agglomerations and architectural problems," *Journal of Software Engineering Research and Development*, vol. 3, no. 1, pp. 1–22, 2015.

[22] I. O. for Standardization, *ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models*. ISO, 2011.

[23] P. Kaminski, "Reforming software design documentation," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, Oct 2007, pp. 277–280.

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

[25] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[26] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, Jan 1990.

[27] E. Murphy-Hill and A. P. Black, "Seven habits of a highly effective smell detector," in *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '08. New York, NY, USA: ACM, 2008, pp. 36–40.

[28] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *2018 IEEE International Conference on Software Architecture (ICSA)*, April 2018, pp. 176–17 609.

[29] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 488–498. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884822

[30] Microsoft. (2019, February) Visual studio 2017 version 15.9 release notes. Available at https://docs.microsoft.com/pt-br/visualstudio/releasenotes/vs2017-relnotes.

[31] T. Sharma, P. Mishra, and R. Tiwari, "Designite: A software design quality assessment tool," in *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, ser. BRIDGE '16. New York, NY, USA: ACM, 2016, pp. 1–4. [Online]. Available: http://doi.acm.org/10.1145/2896935.2896938

[32] T. R. Foundation. (2019, February) The r project for statistical computing. Available at https://www.r-project.org/.

[33] M. Alenezi and M. Zarour, "An empirical study of bad smells during software evolution using designite tool," *i-Manager's Journal on Software Engineering*, vol. 12, no. 4, pp. 12–27, Apr 2018.

[34] N. Moha, Y. Gueheneuc, L. Duchien, and A. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transaction on Software Engineering*, vol. 36, pp. 20–36, 2010.

[35] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, May 2015, pp. 51–60.

[36] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *ICSM12*, Sept 2012, pp. 662–665.

[37] G. Bavota, A. D. Lucia, M. D. 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, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215001053

[38] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes?: A multi-project study," in *Proceedings of the 31st Brazilian Symposium on Software Engineering*, ser. SBES'17. New York, NY, USA: ACM, 2017, pp. 74–83. [Online]. Available: http://doi.acm.org/10.1145/3131151.3131171

[39] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 465–475. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106259