# Architecture Smells and Pareto Principle: A Preliminary Empirical Exploration

Alexandra-Maria Chaniotaki
*Athens University of Economics & Business*
Athens, Greece
alchaniotakis@gmail.com

Tushar Sharma
*Siemens Technology*
Charlotte, USA
tusharsharma@ieee.org

***Abstract*—Architecture smells represent violations of best practices recommended for software architecture that adversely impact various quality attributes of a software system. Though architecture quality is considered very important by the software engineering community, architecture refactoring, given involved high risk and effort, is often avoided by software development teams. In this paper, we empirically explore the properties of architecture smells in the context of the Pareto principle. We investigate the degree of adherence of architecture smell occurrences to the Pareto principle and explore the influence of other related factors *i.e.,* programming language and size of the repositories. To this end, we analyzed 750 Java and 361 C# repositories containing more than 50 million lines of code to detect seven kinds of architecture smells. We found that approximately 45% of the Java repositories follow the Pareto principle. Moreover, C# repositories show significantly higher adherence (66%) to the principle than the repositories written in Java. Our results indicate that the size of the repositories shows a low negative correlation with the Pareto categories. The results imply that software development teams can figure out a few vital components suffering from architecture smells by carrying out the Pareto analysis. It will allow them to optimize their efforts towards making their software architecture quality better.**

***Index Terms*—Architecture smells, Pareto principle**

## I. Introduction

The metaphor of code smells refers to the software quality issues impairing the maintainability of a software system [1], [2]. Architecture smells are quality issues at architecture granularity affecting software quality adversely [3], [4]. Software architecture is a blueprint of a software system representing the critical design decisions with system-wide impact [5]. Many studies [6]–[9] in this domain have established that smells negatively impact software architecture. However, given their wide scope and side-effects, architecture smells require considerable effort to refactor; that, in turn, discourage software developers from carrying out architecture refactorings [8], [10].

Software engineering researchers have proposed mechanisms to prioritize code smells [11], [12] based on factors such as severity and change patterns. Though these code smells prioritization techniques could be extended to architecture smells to some extent, these techniques do not fulfill the unique requirements posed by architecture smells. One such requirement is identifying the smallest possible number of architectural components that can be changed for a refactoring exercise in such a way that it fixes as many as possible architecture smells. This is desired because architecture smell refactoring, by definition, changes one or more components and impacts, by ripple changes, a set of components. By keeping the number of components requiring changes due to architecture refactoring limited, ripple effect of the refactoring exercise and the associated refactoring effort is optimized. This optimization is a result of the fact that some refactorings are not required to be carried out as they are not applicable or redundant after applying a subset of refactorings [13].

The *Pareto principle* or the 80:20 rule is attributed to the Italian economist Vilfredo Pareto who observed that 80% of the land in Italy is owned by 20% of the population [14]. The principle exhibits that a task on a set of artifacts can be optimized by performing the task on a subset of artifacts in an attempt to reduce the effort to 20% of its original value and yet achieve as high as 80% of the output. The principle carries a significant influence on understanding the cause and attaining the efficacy in fixing the effects. Hence, the principle has been studied in various domains, including economics [15], law [16], and social sciences [17]. Software engineering researchers have also explored the applicability of the principle on topics including defect analysis [18], [19] and investigations of contributions to open-source projects [20], [21]. In the context of code smells, a study by Malhotra et al. [22] shows that refactoring 10% of the total classes may lead to code smells reduction up to 47%.

In this study, we empirically explore the properties of architecture smells in the context of the Pareto principle. We not only investigate the degree of adherence of architecture smell occurrences to the Pareto principle, but we also explore the influence of other related factors, *i.e.,* programming language and the size of the repositories. Though researchers have studied various aspects of architecture smells recently [8], [23]–[26], to the best of our knowledge, the Pareto characteristic of architecture smells has not been studied yet. Studying this phenomenon in architecture smells could lead us to many insights. A high degree of adherence to the principle would mean that a software development team could optimize their refactoring efforts focusing on *a vital few* components and attain maximum benefits for the effort. Researchers may utilize the insights from the study to better focus on architecture smells of a few critical components resulting in an optimized order of refactorings.

TABLE I
DESCRIPTION OF CONSIDERED ARCHITECTURE SMELLS

| Architecture smell | Description |
|---|---|
| Cyclic dependency | Two or more architecture-level components depend on each other directly or indirectly [4], [27]. |
| Unstable dependency | A component depends on other less stable components [28]. |
| Ambiguous interface | A component offers only a single, general entry-point into the component [29]. |
| God component | A component is excessively large either in terms of Lines Of Code (LOC) or number of classes [4]. |
| Feature concentration | A component realizes more than one architectural concerns or features [3]. |
| Scattered functionality | An architectural concern is realized by multiple components [29]. |
| Dense structure | Excessive and dense dependencies among components lead a project to suffer from this smell [30]. |

## II. METHOD

We explore three research questions in the study. We downloaded a large number of C# and Java repositories from GitHub. We analyzed all the downloaded repositories using Designite and DesigniteJava to detect architecture smells. We provide the answers to the research questions by analyzing the identified smells from all the considered repositories.

### A. Research Questions

**RQ1.** ***Do architecture smells follow the Pareto principle?***
This question investigates the degree of adherence of architecture smells to the Pareto principle by analyzing a large set of open-source repositories. This exploration aims to explore the interesting attribute associated with architecture smells and could lead to optimizing architecture refactoring efforts.

**RQ2.** ***Do different programming languages exhibit the phenomenon differently?***
In this research question, we compare the occurrence of the phenomenon in programs written in Java and C# programming languages. This experiment will allow us to understand the role of programming languages in the adherence of architecture smells to the Pareto principle.

**RQ3.** ***Does the size of the repositories influence the adherence to the Pareto principle?***
We investigate whether the size of the repositories influences the adherence to the Pareto characteristic. It will enable us to better understand the phenomenon and the architecture smells.

### B. Subject Systems

We used RepoReapers [31] to select the subject systems. RepoReapers analyzes GitHub repositories and provides their quality characteristics based on eight dimensions (architecture, continuous integration, unit testing, community, documentation, history, issues, and license). RepoReapers assigns a score corresponding to each dimension. We selected all the repositories containing either Java or C# code where at least seven out of eight RepoReapers' dimensions have a greater than zero score. We selected repositories tagged with more than 10 stars and with at least $1,000$ lines of code. We used the same criteria for both the programming languages and downloaded $1,721$ Java and 468 C# repositories. We used DesigniteJava [32] and Designite [33] for analyzing Java and C# repositories respectively. DesigniteJava requires compiled `.class` files along with source code files to identify smells correctly and hence we first compiled the subject systems. We checked for the usage of one of the three commonly used build systems for Java, *i.e., Maven*, *Gradle*, and *Ant* and triggered the corresponding command to compile the project automatically. We could not analyze some of the repositories due to either missing external dependencies, custom build mechanisms (*i.e.,* missing standard C# project files), or build mechanism other than the ones considered in this study. We discarded *test* and *sample* components to keep the focus of the analysis on the production code. Furthermore, we removed projects for which the total number of lines of code is less than 1,000 after removing test and sample components. At the end, we successfully analyzed and retained 750 Java and 361 C# repositories.

### C. Architecture Smells and Tool Support

Table I provides a brief description of architecture smells considered in this study. Sharma et al. [24] provide a detailed description of the smells and their corresponding detection mechanisms. We used DesigniteJava [32] and Designite [33] to identify architecture smells in the subject systems. Both the software design quality assessment tools support detection of the architecture smells apart from supporting detection of a wide range of design and implementation smells. A detailed validation of the tools can be found in Sharma et al. [24]. Both the tools implement the same mechanism to detect the smells[1]. The tools have been used in various related studies [24], [34], [35]. An architecture component is referred to a namespace (C#) or a package (Java); in case of nesting, we consider only the terminal (most deeply nested) components. We used *Academic* licenses of the tools for this study.

### D. Reproduction package

We have made available all the data and scripts online [36] to facilitate reproducibility. We encourage the reader to explore, utilize, and extend the provided data, scripts, and experiments.

## III. RESULTS

### A. RQ1. Do architecture smells follow the Pareto principle?

*1) Approach:* We consider all the subject systems written in Java among the selected repositories and detect architecture smells using DesigniteJava. We prepare a list of all the components (*i.e., packages* in Java) for all the repositories and note all the architecture smells that were reported in each component. Subsequently, we carry out the Pareto analysis, *i.e.,* we find whether $20\%$ of the components with the highest numbers of architecture smells account for $80\%$ of the total architecture smells.

[1]http://www.designite-tools.com/faq/#11

It is relevant to mention that segregating architecture smells based on the components where they occur is a non-trivial task because the scope of each of the architecture smell is not the same. For instance, the scope of a *god component* smell instance is limited to the component where it gets reported, and hence changes in only that component are required to refactor the smell. However, the scope of a *scattered functionality* smell instance spans multiple components and it requires refactoring in two or more components to remove the smell. Similarly, the scope of *cyclic dependency* and *dense structure* smells is also not limited to a component.

To factor-in the varying scope, we carry out a deeper analysis. We analyze the *cause*—reported by the smell detection tool—of three architecture smells where the scope is not limited to a single component and extract other components that participate in the smell to assign them the identified smell. For example, if an instance of *scattered functionality* is reported in component `A` where two components `B` and `C` implement the same architectural concern then we assign this smell not only to component `A` but also to component `B` and `C`. Similarly, when an instance of *dense structure* is encountered, we assign this smell to all causing components. However, *cyclic dependency* smell has a unique characteristic. Though there could be multiple components participating in an instance but refactoring one component makes the smell disappear. Thus, despite we assign a *cyclic dependency* instance to all the components participating in the cycle but upon the first inclusion of the component in the Pareto analysis for a smell instance, we remove the smell from other components belonging to the same cycle. This mechanism helps us carry out the Pareto analysis respecting the semantics of the smells.

*2) Results:* Figure 1 shows the result of the Pareto analysis. We not only analyze the detected architecture smells data for the traditional 20:80 Pareto category but also for other categories to observe the smell occurring pattern. In the figure, a Pareto category `X-Y` refers to `X%` components having at least `Y%` of architecture smells.

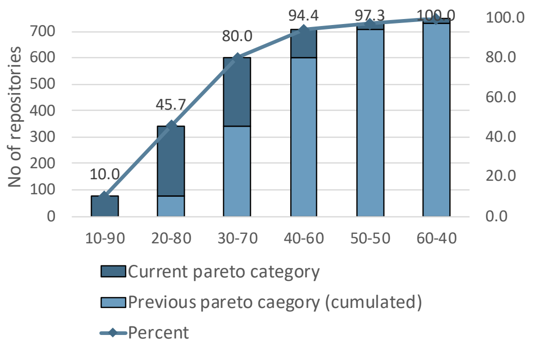


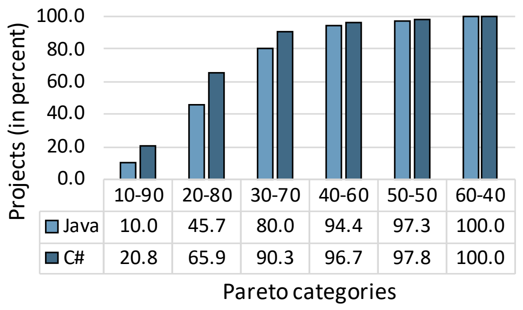
Fig. 1. Results of Pareto analysis for Java repositories

**Our results show that** $45.7\%$ **of the considered Java repositories follow the Pareto principle** *i.e.,* the top $20\%$ of the components have $80\%$ of the architecture smells. We observe that at one extreme, along the expected lines, approximately $10\%$ repositories belong to the $10-90$ category *i.e.,* $10\%$ of the components have $90\%$ of the architecture smells in these repositories. On the other hand, category 60-40 covers all the considered repositories.

*3) Implications:* This result implies that a large number of Java repositories can be refactored from the architecture perspective by modifying a relatively small number of packages to eliminate a relatively large portion of architecture smells. A development team can explore the Pareto category of their project and figure out the vital few components suffering from architecture smells. It will allow them to optimize their efforts towards making their software architecture quality better.

## *B. RQ2. Do different programming languages exhibit the phenomenon differently?*

*1) Approach:* We extend the analysis step of RQ1 to a new set of repositories written in C#. Similar to Java repositories, we analyze all the considered C# repositories using Designite and compile a list of all the components (*i.e., namespaces* in C#) along with their corresponding individual types of architecture smells. We perform the Pareto analysis considering data from both the programming languages and identify different Pareto categories. We use the same method to segregate architecture smells based on the C# components as illustrated in RQ1 for Java repositories. We carry out Mann-Whitney U test to investigate the degree of similarity between categories obtained from both kinds of repositories. For the test, the Pareto categories are the groups where the dependent variable is the Pareto category (ordinal) and the independent variable is the programming language (with two groups C# and Java).



| | 10-90 | 20-80 | 30-70 | 40-60 | 50-50 | 60-40 |
|---|---|---|---|---|---|---|
| ▨ Java | 10.0 | 45.7 | 80.0 | 94.4 | 97.3 | 100.0 |
| ■ C# | 20.8 | 65.9 | 90.3 | 96.7 | 97.8 | 100.0 |

Fig. 2. Comparison of the Pareto analysis between Java and C# repositories

*2) Results:* Figure 2 presents the comparison of the Pareto property between programs written in Java and C#. We observe a significant difference between the sets of programs. Approximately, every two out of three C# repositories ($65.9\%$) adhere to the Pareto principle compared to $45.7\%$ Java repositories. We obtained $W = 102,210$ (p-value $= 3.646e - 12$) from the Mann-Whitney U test that indicates that the Pareto categories of the repositories representing both the programming languages are significantly different. **This result leads to the insight that architecture smells tend to congregate around a few components in C# repositories compared to the Java repositories.**

*3) Implications:* The results imply that refactoring efforts in C# repositories, in general, may tend to concentrate on fewer components than Java repositories for a similar number of architecture smells.

### *C. RQ3: Does the size of the repositories influence the adherence to the Pareto principle?*

*1) Approach:* We carry out a Spearman correlation analysis between LOC of the analyzed subject systems and the assigned Pareto category. We assign an ordinal value to each Pareto category to carry out Spearman correlation analysis. We assign the closest Pareto category that the repository satisfy. We also examine the influence of the size on the Pareto categories by visually analyzing the results.

*2) Results:* Figure 3 shows a boxplot to observe the Pareto characteristic against the size of the repositories. We observe that the Pareto categories $10 - 90$ and $20 - 80$ tend to have larger repositories than the rest of the categories. Specifically, the median LOC of the first two categories combined is $17,241$ and $11,689$ for Java and C# repositories respectively compared to $11,126$ and $9,225$ for all the categories combined.

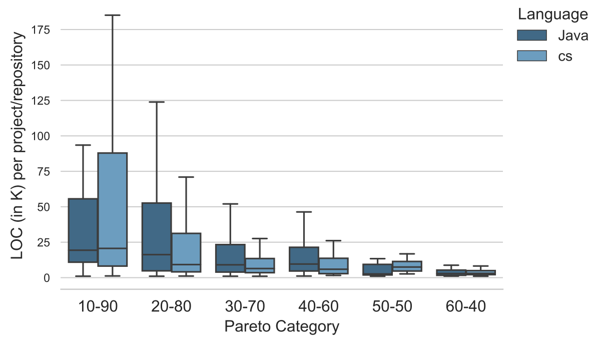


Fig. 3. Comparison of the Pareto categories with respect to size between Java and C# repositories

We obtain Spearman correlation coefficient $\rho = -0.26$ (p-value $< 2.2e{-}16$) between Pareto category and LOC belonging to all the repositories. For individual datasets, C# repositories show $\rho = -0.29$ (p-value $= 8.09e-09$) and Java repositories exhibit $\rho = -0.26$ (p-value $= 1.78e-13$) as the value of the coefficient. The value of the coefficients indicate that LOC **has low negative correlation with Pareto categories.**

*3) Implications:* We observe a low negative correlation between Pareto categories and LOC of the repositories. The observation opens the opportunities for the research community to extend this exploration to better understand other factors that may show higher influence on the phenomenon.

## IV. Related Work

Diverse efforts have been made to offer consolidated lists of definitions [3], [29], [37] as well as to detect architecture smells [26]. Some authors such as Martini *et al.* [8] and Le *et al.* [9] have studied the impact of architecture anomalies on software maintainability. Furthermore, the software engineering community has explored architectural technical debt *i.e.,* non-optimal technical decisions concerning software architecture [38], [39] and its effect on various aspects belonging to software artifacts. Researchers have proposed a mechanism to prioritize code smells [11], [12] using various aspects and strategies. Specifically for architecture smells, there have been some attempts to propose strategies for prioritizing architecture smells and architecture technical debt [8], [40].

The software engineering community has observed the applicability of the Pareto Principle to various software engineering aspects. Iqbal and Rizwan [41] observed that $20\%$ of the initially listed tasks in the waterfall model yield $80\%$ of the results during the software development process. Yamashita *et al.* [21] and Mockus *et al.* [20] discuss the power-law characteristics of the contributions by software developers in open-source software systems. Furthermore, researchers [18], [19] have studied the distribution of defects in software systems and the applicability of the Pareto principle. To the best of our knowledge, this is the first empirical study that investigates the degree of adherence of architecture smells to the Pareto principle by analyzing a large number of programs written in two major programming languages.

## V. Threats to Validity

*Construct validity* measures the degree to which tools and metrics measure the properties they are supposed to measure. Using two separate tools to detect smells and compare the results may pose a threat to validity. To mitigate it, we ensured from the developer of the tools that both the tools use exactly the same mechanism, metrics, and thresholds to detect architecture smells. While processing cyclic dependency smell instances, the order of inclusion of a component may influence the distribution of smells; however, the overall results will not change. *External validity* concerns generalizability and repeatability of the produced results. To encourage the replication and building over this work, we have made all the scripts, and data available online [36]. *Internal validity* refers to the validity of the research findings. In the context of our investigation, we assume that both programming languages are similar by paradigm and language structure and hence comparable. It would be interesting to investigate the observed phenomenon on programs written in programming languages belonging to different paradigms.

## VI. Conclusions and Future Work

In this empirical investigation, we explored the degree of applicability of the Pareto principle on architecture smells. We found that $45.7\%$ of the Java repositories follow the Pareto principle. Our results also reveal that C# programs show a significantly higher degree of adherence to the Pareto principle compared to Java programs. Our exploration shows that LOC of the repositories has a low negative correlation with Pareto categories. In the future, we would like to extend the scope of the study to other programming languages. Also, we are interested in exploring the degree of influence other factors (such as program properties and metrics) on the applicability of the principle.

## References

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

[2] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018.

[3] H. S. de Andrade, E. Almeida, and I. Crnkovic, "Architectural bad smells in software product lines: An exploratory study," in *Proceedings of the WICSA 2014 Companion Volume*, ser. WICSA '14 Companion. ACM, 2014, pp. 12:1–12:6.

[4] M. Lippert and S. Roock, *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons, 2006.

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[6] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, "Are automatically-detected code anomalies relevant to architectural modularity?" in *the 11th annual international conference*. ACM Press, 2012, pp. 167–178.

[7] W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. V. Staa, "On the relationship of code-anomaly agglomerations and architectural problems," *Journal of Software Engineering Research and Development*, vol. 3, no. 1, p. 11, 2015.

[8] A. Martini, F. A. Fontana, A. Biaggi, and R. Roveda, "Identifying and prioritizing architectural debt through architectural smells: A case study in a large software company," in *Software Architecture*, C. E. Cuesta, D. Garlan, and J. Pérez, Eds. Cham: Springer International Publishing, 2018, pp. 320–335.

[9] 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.

[10] G. Samarthyam, G. Suryanarayana, and T. Sharma, "Refactoring for software architecture smells," in *Proceedings of the 1st International Workshop on Software Refactoring*. ACM, 2016, pp. 1–4.

[11] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt: A code smell intensity index," in *IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015, pp. 16–24.

[12] R. Arcoverde, E. Guimarães, I. Macía, A. Garcia, and Y. Cai, "Prioritization of code anomalies based on architecture sensitiveness," in *2013 27th Brazilian Symposium on Software Engineering*, 2013, pp. 69–78.

[13] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, 2014, p. 331–336.

[14] V. Pareto, *Cours d'économie politique*. Librairie Droz, 1964, vol. 1.

[15] R. Sanders, "The pareto principle: its use and abuse," *Journal of Services Marketing*, vol. 1, no. 2, pp. 37–40, 1987.

[16] L. Kaplow and S. Shavell, "The conflict between notions of fairness and the pareto principle," *American Law and Economics Review*, vol. 1, no. 1, pp. 63–77, 1999.

[17] H. F. Chang, "A liberal theory of social welfare: fairness, utility, and the pareto principle," *Yale LJ*, vol. 110, p. 173, 2000.

[18] N. Walkinshaw and L. Minku, "Are 20% of Files Responsible for 80% of Defects?" in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '18, 2018.

[19] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Foundations of empirical software engineering: the legacy of Victor R. Basili*, vol. 426, no. 37, pp. 426–431, 2005.

[20] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, p. 309–346, Jul. 2002.

[21] K. Yamashita, S. McIntosh, Y. Kamei, A. E. Hassan, and N. Ubayashi, "Revisiting the applicability of the pareto principle to core development

[22] R. Malhotra, A. Chug, and P. Khosla, "Prioritization of classes for refactoring: A step towards improvement in software quality," in *Proceedings of the Third International Symposium on Women in Computing and Informatics*, ser. WCI '15, 2015, p. 228–234.

teams in open source software projects," in *Proceedings of the 14th International Workshop on Principles of Software Evolution*. ACM, 2015, pp. 46–55.

[23] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[24] T. Sharma, P. Singh, and D. Spinellis, "An empirical investigation on the relationship between design and architecture smells," *Empirical Software Engineering (EMSE)*, vol. 25, pp. 4020–4068, 2020.

[25] F. A. Fontana, V. Lenarduzzi, R. Roveda, and D. Taibi, "Are architectural smells independent from code smells? an empirical study," *Journal of Systems and Software*, vol. 154, pp. 139 – 156, 2019.

[26] H. Mumtaz, P. Singh, and K. Blincoe, "A systematic mapping study on architectural smells detection," *Journal of Systems and Software*, vol. 173, p. 110885, 2021.

[27] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells," in *WICSA*. IEEE Computer Society, 2015, pp. 51–60.

[28] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 433–437.

[29] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, ser. QoSA '09. Springer-Verlag, 2009, pp. 146–162.

[30] 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.

[31] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, Dec 2017.

[32] T. Sharma, "Designitejava (enterprise)," Sep. 2019, available online at http://www.designite-tools.com/designitejava. [Online]. Available: https://doi.org/10.5281/zenodo.3401802

[33] ——, " Designite - A Software Design Quality Assessment Tool," May 2016, available online at http://www.designite-tools.com. [Online]. Available: https://doi.org/10.5281/zenodo.2566832

[34] W. Oizumi, L. Sousa, A. Oliveira, L. Carvalho, A. Garcia, T. Colanzi, and R. Oliveira, "On the density and diversity of degradation symptoms in refactored classes: A multi-case study," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 346–357.

[35] 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, pp. 12–27, 2018.

[36] A.-M. Chaniotaki and T. Sharma, "Data, scripts, and other relevant information for the pareto principle analysis," Feb. 2021. [Online]. Available: https://github.com/tushartushar/ParetoPrincipleArchSmells

[37] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic, "Relating architectural decay and sustainability of software systems," in *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, April 2016, pp. 178–181.

[38] T. Besker, A. Martini, and J. Bosch, "Managing architectural technical debt: A unified model and systematic literature review," *Journal of Systems and Software*, vol. 135, pp. 1 – 16, 2018.

[39] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, Nov 2012.

[40] E. Guimaraes, A. Garcia, and Y. Cai, "Exploring blueprints on the prioritization of architecturally relevant code anomalies – a controlled experiment," in *2014 IEEE 38th Annual Computer Software and Applications Conference*, July 2014, pp. 344–353.

[41] M. Iqbal and M. Rizwan, "Application of 80/20 rule in software engineering waterfall model," in *International Conference on Information and Communication Technologies*. IEEE, 2009, pp. 223–228.