Unraveling the Impact of Code Smell Agglomerations on Code Stability

***Abstract*—Code smells are symptoms in the source code that indicate code quality degradation and, consequently, may affect code comprehension and maintenance. Moreover, when two or more code smells occur on the same piece of code, forming an *agglomeration*, they may be more harmful to the code quality. Although the impact of smells in isolation is well known, the impact of their agglomeration is still underexplored. Our goal with this study is to provide evidence of how agglomerations impact code stability, *i.e.,* we investigate if agglomeration suffer more modifications along the system evolution, and in which intensity. For this purpose, we mined two years of commit history from 30 open-source Java systems from GitHub. To analyze code stability, we considered four measurements: number of commits, lines of modified code, rate of modified classes, and the proportion of changes. We examined these measurements from two perspectives: by system and by aggregating all system data. Additionally, we further considered how a class created/deleted in this time-span impacts our results. Our main findings are: (i) classes with two or more code smells of different types change more frequently and in more intensity than classes with a single smell or no smell; (ii) the stability of the class varies greatly with the system under analysis; (iii) when a smelly class was deleted in our time-range, they usually had several lines of code added until it became unsustainable. We can conclude that agglomerations change with more frequency and intensity, raising maintenance and evolution costs. Consequently, this information can be used to prioritize code refactoring.**

***Index Terms*—code smell agglomerations, code stability, code maintenance and evolution**

I. Introduction

Code smells are symptoms in the source code that may indicate a degradation in its quality [1]. For instance, code smells can indicate that a class or method has many responsibilities, is complex, presents low cohesion, and inheritance problems. Consequently, they have a direct impact on development activities, such as maintenance and evolution [2, 3]. They also affect understandability and reusability [2, 4, 5], and are related to faults and changes [3, 6–10]. Studies found that they are usually introduced in the creation of new classes and close to deadlines [11–13]. To improve code quality and avoid future costs, developers can refactor the smells. *Refactoring* is an activity in which the code is modified to improve its quality, without changing its external behavior [1]. Such modifications include splitting a class/method, moving a method to another class, renaming, and others.

Studies in the literature identified that when smells occur together in the same snippet of code (*e.g.,* class or method), forming a *code smell agglomeration*, they are even harder for developers to evolve, to maintain, and to refactor the system [14–17]. Moreover, to the best of our knowledge, few studies assess how such agglomerations impact code quality [18–21]. Our goal is to evaluate and compare if code smell agglomerations are more unstable, *i.e.,* they change more frequently and in more intensity when compared to other classes, such as classes with a single smell or no smell. If they are unstable, changing them is more time-consuming and costly than changing other classes. For this purpose, we mine two years of commit history for 30 open-source Java systems from the GitHub platform, one year before and one year after a selected release. We selected these systems from the top-starred ones that received a major release on the data collection (2021-2022). For each system, we identify nine smells using four different detection tools: three smells at the class level and six at the method level. We then investigate the history of changes by analyzing the differences in the distribution for the number of commits, the number of lines of code modified (added/deleted/churn), and the rate of the classes on the major release that got changed in this two year time span. Additionally, we investigate the intensity of these changes, *i.e.,* the proportion of lines added/deleted/churn in respect to the class size. To analyse our data, we have used non-parametric hypothesis testing, the Mann-Whittney U, combined with Cliff's Delta effect size. Our contributions are as follows:

- We provide empirical evidence that classes with code smell agglomerations change more frequently and with more intensity compared to other classes. This finding highlights the importance of refactoring such classes to mitigate future maintenance and evolution costs.
- The design and implementation of a large study about code smell agglomeration analysis with 30 Java systems. We identify 9 code smells over a spanning time of 2 years. All artifacts and data collected during the study are publicly available in a replication package [22].

Audience. Researchers and practitioners shall benefit from our insights in understanding the impact of code smell agglomerations on code stability. Contributors and maintainers of industrial and open-source projects can gain insights into agglomeration patterns and their impact on project evolution. Consequently, developers can make informed decisions about refactoring strategies and prioritize efforts to improve code quality. The findings may stimulate discussions among developers about best practices for managing agglomerations. It can mitigate technical debt and ensure the long-term maintainability of software systems. Finally, researchers can use our study design to extend our analysis.

II. Code Smells and Agglomerations

Code smell is a term used in software development to describe symptoms or indicators of potential issues that may impact the class quality and maintainability [1]. They may also affect system understandability, reusability and extensibility [2, 4]. In this work, we analyze nine smells, 3 at the class level and 6 at the method level. Large Class (LC), Data Class (DC), and Refused Bequest (RB) are at the class level. Meanwhile, Feature Envy (FE), Intensive Coupling (IC), Dispersed Coupling (DiCo), Long Parameter List (LPL), Shotgun Surgery (SS), and Long Method (LM) are at the method level. Their definition can be found in Fowler's, and Lanza and Marinescu's books [1, 15]. We selected these smells due to tool support and because they cover different modularity problems.

In our work, a code smell agglomeration occurs when two or more code smells co-occur in the same class. To simplify our analysis, we categorize the agglomerations according to their types: *Heterogeneous Agglomeration* is a class that presents at least two code smells of different types in its code. For instance, we can have a class that has simultaneously a Large Class and a Long Method. *Homogeneous Agglomeration* is a class that presents at least two smells, but *all* are of the same type. For instance, we can have a Homogeneous Agglomeration with two Feature Envy smells. If this class had an additional smell that is not a Feature Envy, it would be classified as Heterogeneous Agglomeration. *Isolated smell* is a class that has only one smell, for instance, a class that has only one Feature Envy. Finally, *Clean Class* is a class that does not present any of the detected smells. To avoid confusion, when we mention Heterogeneous and Homogeneous Agglomerations, we are referencing classes that present a Heterogeneous/Homogeneous Agglomeration. We also we use the expression "agglomeration types" to represent the Heterogeneous, Homogeneous, Isolated and Clean classes.

III. Study Design

Figure 1 presents an overview of our study design. Arrows indicate that the element was used as input for the next step. Step A is the system selection. In total, we selected 30 top-stared Java systems from GitHub. In Step B, we used four detection tools to identify nine code smells. Based on the output of these tools, we applied a voting method strategy, where we consider the agreement of two detection tools on the instance smelliness. In Step C, we use our script to categorize the instances in different agglomeration types. Sections III-B and III-C describe in detail how the systems were selected and how the ground truth was created.

In Step D, we request through our Python script the list of pull requests and commits associated with the selected systems. This step is required to ensure that all data collected are in the range of our 2-year time span concerning the major release date, and that commits were merged in the main branch. In Step E, we request and process each commit that had an associated pull request and their associated information, such as author, merge data, and *diff*. More details for Steps D and E are provided in Section III-D. In Step F, to obtain

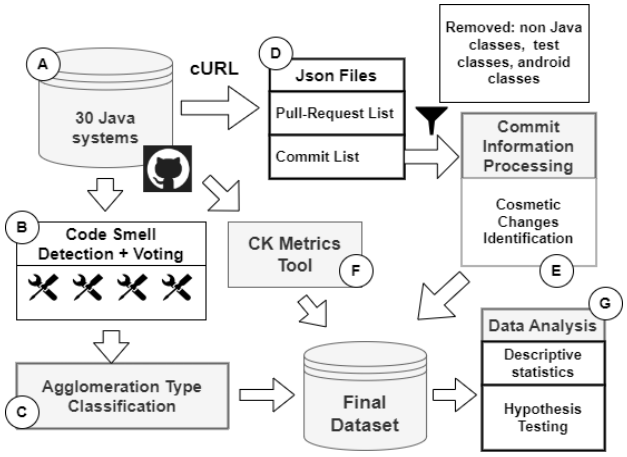

Fig. 1. Overview of the study Design

software metrics related to size, we used the CK metrics tool [23], since our analysis includes the rate of changes according to the agglomeration size. Finally, in Step G, we analyze our data using statistics and hypothesis testing to answer our research questions (Section III-A). More detail for Step G is presented in Section III-E. We highlight that Steps B, D, and F were conducted in parallel.

A. Research Questions

The impact of code smell agglomerations on code stability is still underexplored [6, 8, 21]. Our goal is to provide evidence through the analysis of the stability of the code of different agglomerations types, in the context of current development practices [3, 13, 24]. The following research questions (RQs) guide this work:

RQ1: *Do Heterogeneous and Homogeneous Agglomerations undergo more frequent changes compared to Isolated and Clean types?* We approach this question in three ways: (i) we analyze the number of commits that modified the agglomeration type; (ii) we compare the differences between lines of code added/deleted and churn (the sum of added and deleted lines); and (iii) we examine the number of classes from each agglomeration that undergo changes over the two-year timeframe. The goal of this question is to compare the change patterns of different agglomeration types, focusing on understanding how stable agglomerations are, to gain a comprehensive knowledge of their dynamics in software systems.

RQ2: *Do Heterogeneous and Homogeneous Agglomerations undergo changes in more intensity compared to Isolated and Clean types?* To answer this question, we present the proportion of the lines of code added/deleted/churn with respect to the number of lines of code (LOC) of its respective class. We further normalized them by the system's agglomeration type mean LOC.

B. System Selection

An important decision about building a dataset is the system in which to collect data (Figure 1 (A)). As we aim to

comprehend the up-to-date changes in agglomeration classes in contemporary development practices, our data collection spanned from April 2021 to April 2023. We specifically selected systems with merged commits in 2021 to capture the system dynamics. For generalization of results, we apply some inclusion criteria (IC) when selecting systems: (IC1) we selected top-star systems, indicating that the system is widely accepted by the open-source community; (IC2) we selected systems with at least two years of maturity, since it ranges our data analysis time span; (IC3) we selected systems with 90% of its code written in Java, due to limitations of the used code smell detection tools. We discarded repositories that were for educational purposes.

Table I presents information about the 30 systems we selected. The first column presents their names and versions. The second to fourth columns present, respectively, the number of classes (NOC), methods (NOM), and lines of code (LOC) of each system. Finally, the fifth column presents the total number of code smells (#CS) found for the systems, after we applied a voting strategy, detailed in Section III-C. We can observe from Table I that the systems vary greatly in terms of size. The smallest system in terms of LOC is the *elastic-search-analysis-ik* (2K LOC), while *guava* is the largest system (2M LOC). In terms of code smells, we have found more smells in *dbeaver* (1.3K), while for *java-faker* we only found 2. In total, we have considered 3.5M LOC, distributed in 50.7K classes.

TABLE I
FINAL DATASET

Name	NOC	NOM	LOC	#CS
arthas-3.4.3	834	4,733	39,973	151
cryptomator-1.6.1	590	2,690	16,350	12
dbeaver-21.0.2	6,449	36,575	348,608	1,374
easyexcel-2.2.11	249	1,629	10,639	56
elasticsearch-analysis-ik	28	203	2,051	8
fastjson-1.2.76	249	1,996	44,434	94
gson-2.8.8	231	924	11,721	10
guava-30.1.1	27,412	200,616	2,125,859	75
HikariCP-4.0.0	68	581	4,530	14
hutool-5.7.17	1,214	11,966	79,432	66
java-faker-1.0.2	105	751	3,602	2
jedis	749	6,219	27,404	23
jenkins-2.287	2,432	14,775	120,382	171
jitwatch-1.4.2	538	7,346	46,527	104
jsoup-1.14.2	246	1,551	17,449	18
junit4-4.13.2	310	1,541	10,769	8
libgdx-gdx-1.9.14	2,714	39,338	208,028	327
mall-1.0.2	747	14,322	100,990	15
mybatis-3.5.6	378	2,582	20,533	119
nanohttpd-2.3.1	75	405	3,821	8
netty-socketio-1.7.18	138	712	5,217	9
redisson-3.15.3	1,613	13,607	82,104	119
retrofit-1.6.0	118	403	4,790	12
rocketmq-4.9.2	996	7,536	70,871	352
Sa-Token-1.28.0	191	1,600	8,848	5
Sentinel-1.8.3	1,029	4,963	41,366	94
spring-cloud-alibaba-2.2.2	411	2,003	13,594	34
webmagic-develop-0.7.6	207	955	6,757	19
xxl-job-2.3.0	150	741	8,374	31
zxing-3.4.1	303	1,783	23,614	129
Total	50,774	385,046	3,508,637	3,459

C. Ground Truth Creation

Several code smell datasets have been used in the literature [25, 26]. However, they either are built using old system versions, or they are limited to a few smells analyzed. Thus, we opted to create a ground truth from top-star Java systems from GitHub using automatic code smell detection tools, since building a manual code smell ground truth is not trivial (Figure 1 Step B). It is a time-consuming activity, where different developers consider different aspects of the code [10, 17]. For these reasons, we opted to rely on the agreement of automatic detection tools, since previous studies indicated a good accuracy [27–31]. We selected the Java language since it has several tools that cover several smells [28], and employ different techniques to identify smelly instances, such as metrics, refactoring opportunities, machine learning, historical data, and textual information [26, 28, 30–35].

However, to mitigate tool bias, we used a vote strategy, in which two tools are used to detect the smell. If they agree that the instance has the smell, then we add it to our ground truth. Table II presents the tools used to detect each of the 9 smells considered in this study. An "✓" indicates that the tool was used to detect the smell indicated in the column. For instance, PMD is used to detect DC and LPL.

TABLE II
DETECTION TOOLS

Tool	LC	RB	DC	LM	FE	DiCo	IC	SS	LPL
JDeodorant	✓			✓	✓				
PMD			✓						✓
Organic		✓	✓	✓	✓	✓	✓	✓	✓
JSpIRIT	✓	✓				✓	✓	✓	

Later, we use a script to classify the instances in the agglomeration types described in Section II (Figure 1 Step C). We highlight that some tools have some limitations that were addressed in our script. For example, JSpIRIT does not present a full path for the detected smell, so we had to match the class names with possible matching candidates; for the LPL smell, the PMD tool does not present the method that has the smell. Finally, the code smells were detected only on the release of the system; consequently, a class may present other smells before/after the analyzed release. We highlight that we remove unrelated data, such as test cases, non-Java code, project configurations, and third-party code, from the list of smelly instances.

D. GitHub Data Mining

To achieve our goal, we first decided on a period to compare the change behaviors. We selected a two-year time span: one year before and one after the selected release. Our focus is not on understanding the life cycle of the class, but on the instability of the class in current development practices. Firstly, for each system, we have mined the list of commits in the selected time range and their respective pull-requests (Figure 1 Step D). With the help of the merged field, we consider only those commits that were merged in the main branch of the code. Second, we mined the commits and their associated

diff. We highlight that we have used a command line Python script that uses cURL to request and download all JSON files, using the subprocess library[1] to automatically request all files. To process the JSON files, we use the Pandas library[2].

Third, we parse our data to create a csv file with the class history (Figure 1 Step E). For each commit, we mined: the author and committer information; commit title and message; if it was merged; the class history type; and the lines of code added and deleted. We also parsed the *diff* files to remove information about the non-functional changes, *i.e.,* (i) white spaces added/deleted; (ii) added/deleted braces; (iii) split lines; (iv) indentation positioning. For this purpose, we saved, for each *diff*, the set of additions and deletions, and our analysis is made using only functional code. To identify non-functional changes, such as indentation, we have used Levenshtein Distance from the Jellyfish library[3], a method that compares how different two strings are. Finally, we merge the information about the commits, class size and code smells into one file for each class.

E. Data Analysis

Figure 2 presents a workflow of how the analysis was performed. Before analyzing our data, we separated it according to its agglomeration type (see Section II). First, we have two "Perspectives of Analysis": "By System" and "General" perspectives. "By System" analysis considers all 30 systems data individually, allowing us to understand if different systems have different change behaviors ("By system" rectangle). Moreover, we also analyze all the systems' data to verify if we can generalize our results ("General" rectangle). From now on, we call this dataset as *General Dataset*. For the "General Dataset", for the intensity calculation, the proportions of modified lines of code for each system on the "By System" dataset were combined into a unique dataset.

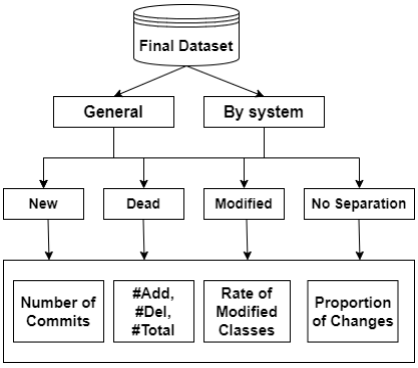

Fig. 2. Data Analysis Steps

We also analyze four different datasets separated by the "Class History Type". They are: "New", in which a commit added the class during its 2-year history; "Dead", in which a commit in the 2-year history deletes the class; "Modified", in which a class had commits that modified its content, but no commit creating/deleting the class in the considered 2-year time span; and "No Separation", in which we consider all three previous categories as a single dataset. This separation is possible since we analyze commits before/after release and, consequently, a class may have been added/deleted in our time frame. Our aim is to analyze how different these class history types affect our finding since it is expected that the "New" type adds several lines of code, while "Dead" deletes a significant amount of code. To avoid confusion when discussing results, we highlight that operations related to lines of code are described as "Modification Type": the number of lines added/deleted is treated as #Add/#Del. We highlight that for all analyses that consider lines of modified code, we have removed the non-functional changes.

Finally, these subsets of the data are used as input for our four analyses: (i) the difference in the number of commits; (ii) if there is a difference in the agglomeration types in terms of lines added, deleted and total (churn); (iii) we compare the number of agglomerations that were changed along our two-year time-span, with the number of instances found for the agglomeration type in the selected release; and (iv) the intensity of the changes. We define the "intensity" of the changes as the proportion of lines of code changed for a modification type, considering the mean LOC of the agglomeration type for each system. This provides us insights about how much of the size of a class is being added/deleted in the commit. For instance, we investigate if the Heterogeneous type had more lines #Add when considering its mean LOC.

To compare our agglomeration types, we use the Mann-Whitney U hypothesis testing [36], since our data do not follow a normal distribution, they are not paired and can be ranked. We opt not to use the multiple comparisons corrections [36] due to our hypothesis formulation and research questions: our goal is not to compare if the four agglomeration types have similar behavior *at the same time*, but to compare if a pair of categories have different distributions. Be i and j two agglomeration categories, with $i \neq j$. Be p the property of interest, $p \in P$, with $P = \{NumberofCommits, \#LinesAdded, \#LinesDeleted\ CodeChurn, ProportionOfModifiedLines, Intensity\}$. Be v the perspective of analysis, in which v can be a "General" or "By System" analysis. Finally, be s, $s \in S$, and $S = \{Modified, New, Dead, NoSep\}$. Our general hypothesis can be described as follows:

$H_{0-vspij}$ = Distribution of property p for categories i and j are equal.
$H_{1-vspij}$ = Distribution of property p for categories i and j are not equal.

To enhance our analysis, we present the Cliff's Delta to provide evidence of the size of the differences between the two distributions. They are interpreted as follows: Negligible ($|\text{delta}| < 0.147$); Small ($0.147 \leq |\text{delta}| < 0.330$); Medium ($0.330 \leq |\text{delta}| < 0.474$); and Large ($0.474 \leq |\text{delta}|$) [37]. Cliff's Delta value can be positive or negative, depending on the order of the comparison. For instance, if the result for

[1]https://docs.python.org/3/library/subprocess.html
[2]https://pandas.pydata.org/
[3]https://pypi.org/project/jellyfish/

the comparison Heterogeneous-Clean is positive, it indicates that Heterogeneous type has higher values than the Clean type. If the signal is negative, it indicates that the value for Heterogeneous is usually lower than the Clean ones. Finally, for the intensity calculation, we have used the Min-Max normalization method [24]. For the Mann-Whittney U we have used the Python library *sciypy.stats*[4], and for Cliff's Delta we used the Cliff's Delta[5] Python library.

IV. Results

This section presents the main findings of this paper. Due to space constraints, we focus on the most interesting results. Our full report is presented in our replication package [22].

A. Commit Overview

For the General dataset, we could reject the following H_0: Het=Clean (-0.58), Hom=Clean (-0.58) and Isol=Clean (-0.584). The Cliff's Delta indicates that, usually, the number of commits for the Clean classes is larger than for smelly classes. Meanwhile, for all 30 systems under analysis, we have similar results, indicating that Clean classes usually receive more commits than smelly classes (see our replication package for more details). This finding may be explained by different factors: (i) there are more Clean classes than smelly classes (see column NOC and #CS in Table I), consequently, it is expected that more Clean classes are changed compared to smelly ones. (ii) Developers may feel intimidated to change longer/complex classes. However, it is not sufficient to look only at the number of commits, since it does not present information about the size and proportion of the changes. In the next section, we present the results taking into account the modified lines of code.

Finding 1: Clean classes receive statistically more commits than smelly classes, and the difference is Large.

B. General Results for Class Type History

Table III presents the Mann-Whittney U results for the lines of modified code for the General dataset. The first column presents the dataset that is being analyzed. The second column presents the modification type that we could reject H_0 for the dataset of the first column. The third and fourth columns present, respectively, the agglomeration type pairs that we could reject the H_0 and their respective effect size.

We can observe in Table III that: (I) we could mostly reject H_0 for #Add and Churn when comparing to Clean and Isolated smells. However, most of them are positive towards the first type, indicating that, usually, lines of code are more added to Clean classes than to Homogeneous ones. (II) For the No Data Separation dataset, we had only two pairs with differences in behavior: the Het-Hom and Hom-Clean. For the pair Hom-Clean, we obtained negative delta values for the Homogeneous category, indicating that Clean classes add more lines of code than Homogeneous ones. Meanwhile, for the Het-Hom pair, we have found that Heterogeneous Agglomerations changes significantly add more lines of code than Homogeneous Agglomerations. (III) For the Modified dataset, for most pairs of agglomeration types, we could reject the H_0, and mainly when compared to the Heterogeneous agglomeration type. However, when observing its values, we can verify that they are Negligible ($|\text{delta}| < 0.147$) and Small ($0.147 \leq |\text{delta}| < 0.330$). We can mainly conclude from this finding that Heterogeneous Agglomerations usually present difficulty in maintenance, even when compared to Isolated classes. (IV) For the Dead dataset, we can observe that for the Heterogeneous and Isolated types, they have several lines of code added until developers finally delete them. (V) Finally, we can observe that separating the dataset according to the Class History Type has a great impact on the results found, and consequently, they should also be considered separately.

TABLE III
General Perspective of Modification Types by Class Type History

Dataset	Mod. Type	Agg. Types	Cliff's Delta
No Separation	#Add	Het-Hom	0.37
		Hom-Clean	-0.47
Modified	#Add	Het-Isol	0.11
		Het-Clean	0.27
		Isol-Clean	0.13
	#Del	Het-Isol	0.12
		Het-Clean	0.17
		Hom-Isol	0.2
		Hom-Clean	0.26
	Churn	Het-Clean	0.12
		Het-Clean	0.27
		Isol-Clean	0.13
Dead	#Add	Het-Isol	0.73
		Het-Clean	0.81
		Isol-Clean	0.47
	Churn	Het-Isol	0.73
		Het-Clean	0.81
		Isol-Clean	0.47

Finding 2: We could mostly reject H_0 for the Modified dataset, and for #Add and Churn modification types. Pairs with Clean type presented Negligible-Small effects towards the smelly agglomeration types. Finally, our analysis provides empirical evidence that Heterogeneous Agglomerations change the most when considering the number of lines added, deleted, and churn.

C. Class Type History Results by System

Figure 3 presents the results found for the hypothesis testing of the modification types for each system under analysis, without separating the data by their class history type. Figure 3 shows three stacked bar plots, one for each of the modification types (#Add, #Del and Churn). The x-axis presents the number of systems that could reject the H_0 for the pair on the y-axis. Meanwhile, the colors inside the bars represent the effect

[4]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html

[5]https://github.com/neilernst/cliffsDelta.

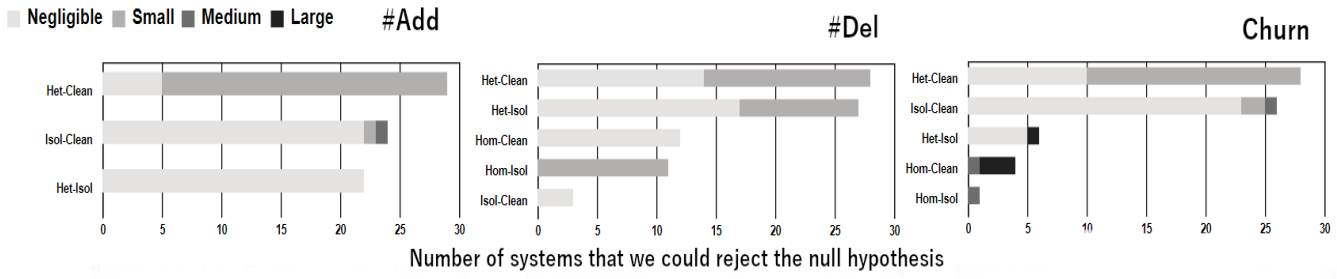

Fig. 3. Summary of Lines of Code Changed by System - No Separation

size. For instance, for the #Add modification type, we could reject for 28 out of 30 systems the H_0. Hence, we could find a difference between the number of lines of code added of Heterogeneous and Clean type (Het-Clean bar). From those, 5 had a Negligible effect, while 23 out of 28 had Small effects.

From Figure 3 and our data, we can observe that: (I) for the #Add modification type, we could reject for most systems the H_0 for Het-Clean (28 out of 30), Isol-Clean (24 out of 30) and Het-Isol (22 out of 30). However, most of the effects are Negligible ($|\text{delta}| < 0.147$) and Small ($0.147 \leq |\text{delta}| < 0.330$). For the Isol-Clean, we had one Medium effect for the *cryptomator* system (0.38). (II) For the #Del modification type, we could reject at least one H_0 for most pairs analyzed, except Het-Hom. For the Het-Clean we could reject the H_0 for 28 out of 30 systems, followed by Het-Isol (17) and Hom-Clean (12), but most effects are Negligible and Small. (III) For the Churn modification type, we could also reject for at least one system most pairs, except Het-Hom. For the Het-Clean, we reject the H_0 for 28 out of 30 systems, followed by Isol-Clean (26 out of 30). However, most of the effects are Negligible and Small. Consequently, for items I, II and III, we could find difference between the pairs. We only found a Large effect for Het-Isol (0.84 at *easyexcel*) and three for Hom-Clean (*dbeaver*, *elasticsearch* and *easyexcel*, with effects of 0.51, 0.5 and 0.5, respectively).

(IV) Interestingly, when consulting our raw data, for both *hutool* and *jedis* systems, the Isol-Clean pair for the #Del modification type were negative toward the Isolated class. When we analyzed the Coefficient of Variation for both Clean and Isolated smells for both systems, we found that for *jedis*, the distribution for the Isolated smells is indeed lower than the Clean ones. For the *hutool*, we identified that the removal of non-functional code is impacting directly on the observed effect sizes.

Finding 3: We have found a consensus on the difference between the lines of code #Add, #Del, and Churn for the pairs Het-Clean, Het-Isol and Isol-Clean. However, most of the effects are Negligible and Small. We also provide evidence, that for most systems, the Heterogeneous agglomeration is more unstable compared to other types for the three modification types.

When separating the dataset by its class history type, for *arthas* e *Sentinel* we could reject some hypotheses for the Dead dataset when comparing Isolated and Clean classes. In total, for the Dead dataset, we rejected 2 null hypotheses for #Add and 1 for Churn, and all effects were Large and positive.

Figure 4 presents the stacked barplots for the Modified dataset, organized the same way as Figure 3. From Figure 4, we can observe that we could reject the H_0 for a few systems. In fact, for all three modification types, none of them could reject the H_0 for more than 10 systems. Observing the results by system, we have found: (I) for the Modified dataset, we mostly could find differences when comparing against Clean classes and the pair Het-Isol. (II) We can also observe that the system *dbeaver* had the most H_0 rejection, followed by *easyexcel*. However, for the *dbeaver* system, most effects were Negligible. Meanwhile, for the *easyexcel*, we have found for the Isol-Clean pair that, for all modification types, the effects were negative and Large. We also highlight that for the *guava* system, we also had a negative effect, but Small, for the pair Isol-Clean (#Del). This is an interesting result, since it is expected that smelly classes proportionally suffer more modifications than Clean classes.

When analysing their distribution, we have found that: (I) for the *guava* system, the deviation of the Isolated classes is significantly higher than those of the Clean classes. For the *easyexcel*, all standard deviations were lower for the Isolated smells when compared to the Clean ones. Moreover, for #Add and Churn modification types, their standard deviation values were close to the Clean ones. (II) Finally, we verified that Isolated smells have a higher proportion of no functional modification (all #Add, #Del and Churn equal to zero), with a difference of values for the *easyexcel* of 49%, and for the *guava* system the difference is smaller: 24%. This indicates that Isolated smells add/remove more non-functional code than the Clean type, for these two systems.

Finding 4: When separating our datasets, we can observe that most of the differences are for the Modified dataset, but we could not reject the null hypotheses for more than 10 systems. We have also found that removing non-functional changes indeed impact the findings.

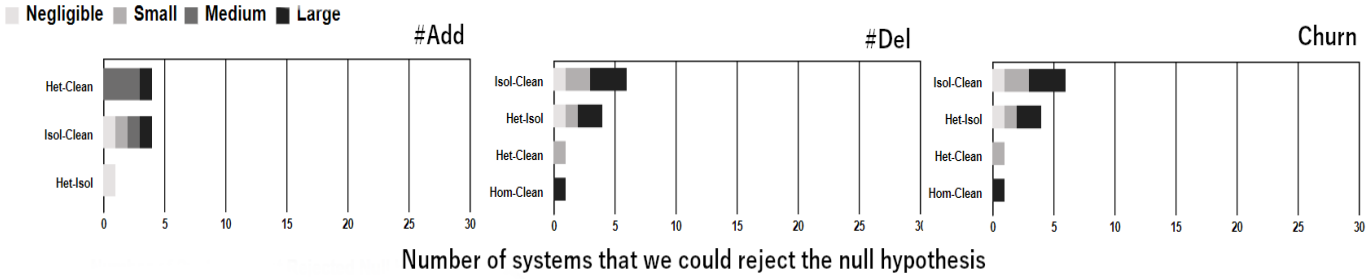

Fig. 4. Summary Intensity of Changes by System - Modified Dataset

D. Rate of Agglomerations Changed

Table IV presents the ratio of agglomeration classes that were both modified in our analyzed time span and existed at the selected system version, divided by the number of agglomerations found for its type. The first column presents the system name. The second to fifth columns present, respectively, the ratio obtained for the Heterogeneous, Homogeneous, Isolated, and Clean classes. Values inside the parenthesis indicate the number of the agglomeration type that changed in our two-year time span. For instance, a value of 1 indicates that all classes of the type changed in our two-year time span, while a value of 0 indicates that no classes of the type were changed. An empty cell indicates that no agglomeration type was found in the system release. Rates in bold indicate that it was the highest rate for the system. The last line presents the results for the General dataset. We highlight that the numerator and denominator are calculated according to the agglomeration type. For instance, for the Heterogeneous, the rate was calculated using the number of Heterogeneous classes that were modified and that existed in the release under analysis, divided by the number of Heterogeneous agglomerations found in the system.

We can observe from Table IV that the highest rate of changes was for the Heterogeneous type, and the lowest was for the Clean type. Smelly classes are rarer than Clean classes, and, consequently, they should be evaluated separately. For 18 out of 30 systems, 50% or more of its Heterogeneous agglomerations were changed in two years, while for the Homogeneous and Isolated both have 10 and 12 systems having at least 50% of their Homogeneous and Isolated smells changed in two years, respectively. For the General perspective, we can observe that 70% of all Heterogeneous agglomerations were modified in two years.

For most cases that achieved a 1 rate, the number of agglomerations was low, ranging from 1 to 7 classes. However, for some systems, we have a high number of Heterogeneous Agglomerations changing: *dbeaver* (169), *libgdx* (31), and *rocketmq* (38). For the Homogeneous type, most systems had few instances changed, ranging from 0 to 6. Notice that the system with the most Homogeneous Agglomerations did not change them significantly: *dbeaver* changed only 5 out of 84 instances. The results for the Clean classes are as expected, Clean classes are more common than smelly ones, and it is not expected that all of them will be changed in two years. However, we can observe some exceptions: for the *spring-cloud-alibaba*, *hutool* and *mybatis3*, 53%, 59%, and 72% of the Clean classes were changed, respectively.

TABLE IV
CHANGES BY RATE OF AGGLOMERATIONS FOUND

System	Het.	Hom.	Isol.	Clean
arthas	**0.94** (17)	0.6 (3)	0.77 (75)	0.38 (270)
cryptomator	**1.0** (1)		0.89 (8)	0.27 (158)
dbeaver	**0.73** (169)	0.06 (5)	0.59 (237)	0.17 (950)
easyexcel	**1.0** (4)	0.0 (0)	0.95 (37)	0.68 (138)
elasticsearch	0.0 (0)		**0.17** (1)	0.0
fastjson	**0.73** (11)	0.67 (4)	0.44 (12)	0.09 (18)
gson		**1.0** (2)	0.5 (2)	0.19 (42)
guava	**0.57** (4)	0.5 (2)	0.48 (23)	0.01 (274)
HikariCP	**0.67** (2)		0.29 (2)	0.09 (42)
hutool	**1.0** (7)	**1.0** (2)	0.70 (32)	0.59 (682)
java-faker			0.5 (1)	**0.50** (52)
jedis	0.0 (0)	0.0 (0)	**0.47** (7)	0.19 (136)
jenkins	0.78 (18)	0.1 (1)	**0.86** (62)	0.35 (825)
jitwatch	**0.05** (1)	0.0 (0)	0.0 (0)	0.01 (3)
jsoup	**1.0** (4)		0.43 (3)	0.11 (26)
junit4			0.0 (0)	0.06 (17)
libgdx	**0.70** (31)	0.55 (6)	0.26 (42)	0.13 (320)
mall	0.0 (0)	0.0 (0)	**0.14** (1)	0.03 (21)
mybatis-3	0.63 (5)	**0.8** (4)	0.24 (19)	0.72 (204)
nanohttpd	**1.0** (1)		0.75 (3)	0.47 (33)
netty-socketio	**1.0** (1)		0.33 (2)	0.08 (11)
redisson	**0.65** (13)	0.2 (2)	0.62 (21)	0.12 (184)
retrofit	0.0 (0)	0.0 (0)	0.0 (0)	0.0 (0)
rocketmq	**0.88** (38)	0.18 (2)	0.50 (64)	0.28 (232)
Sa-Token			0.0 (0)	**0.12** (22)
Sentinel	0.33 (3)	0.0 (0)	**0.35** (23)	0.23 (216)
spring-cloud	**1.0** (3)		**1.0** (27)	0.53 (202)
webmagic	**1.0** (1)	0.5 (1)	0.18 (2)	0.05 (10)
xxl-job	**0.33** (1)	0.0 (0)	0.10 (2)	0.05 (6)
zxing	0.43 (9)	**0.8** (4)	0.22 (11)	0.07 (16)
Mean	**0.63**	0.35	0.42	0.21
General	**0.7** (344)	0.21 (38)	0.51 (719)	0.10 (5073)

RQ1: *Do Heterogeneous and Homogeneous Agglomerations undergo more frequent changes compared to Isolated and Clean types?* For the General dataset, smelly classes change more frequently than Clean ones. We also found that Heterogeneous agglomerations change more frequently than other agglomeration types. For number of commits, we found evidence favorable to the Clean classes being unstable.

E. Rate of Changes Results - General Dataset

Figure 5 presents four boxplots, one for each of the class types. Each boxplot represents the distribution of Lines of Code (LOC) considering all systems in our dataset. Since we are analyzing the proportion of the changes concerning class size, it is important to highlight the differences in the categories' distribution. We can observe from the boxplots that the median size for Heterogeneous and Homogeneous agglomerations is more than 6 times the size of the Clean ones and 3 times the size of the Isolated ones. Consequently, when comparing the intensity of the changes, we have to consider the system size. For this purpose, we use the Min-Max normalization method concerning the system size.

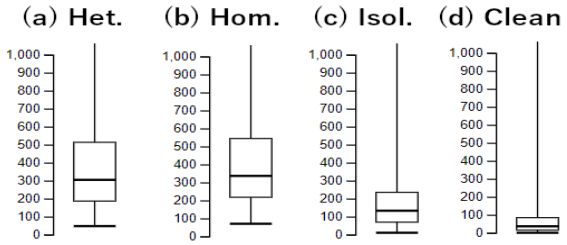

Fig. 5. All Systems LOC Boxplots

Table V presents the results of the proportion of lines changed for the General Dataset, *i.e.,* we are evaluating the intensity of the changes considering the mean value of the lines of code #Added, #Deleted and Churn for each system. The first column presents the categories being compared. The second column presents the Cliff's Delta for the H_0 that we could reject for the different types of modifications. We can see that pairs with Clean classes have a very high Cliff's Delta ($\geq$ 0.75), indicating that smelly classes usually adds and deletes in more intensity than clean ones. This finding is interesting, since Clean classes tend to have smaller sizes than smelly classes (see Figure 5), and consequently, changes to them are more sensitive to size. This finding indicates that developers change smelly classes in higher intensity than non-smelly ones. For #Del and Churn, the pair Het-Clean had the highest effect size (0.89 and 0.95, respectively). Meanwhile, pairs Hom-Clean and Isol-Clean both obtained an effect of 1 for the #Add modification type. Finally, when observing the pair Het-Hom, both #Add and Churn were negative towards the Heterogeneous classes, indicating that the Homogeneous classes change in more intensity than Heterogeneous ones.

Finding 5: When compared to the Clean type, smelly classes (Heterogeneous, Homogeneous and Isolated) change in more intensity. This is specially true for Heterogeneous and Homogeneous, both with larger effects than Isolated.

Table VI presents the results for the General Perspective when considering the Class History Type. The first column presents the agglomeration types being compared. The second and third columns present the Cliff's Delta for the H_0 that we could reject for the Dead and Modified datasets, respectively. The second and third columns are further separated to consider the different modification types: #Add, #Del, and Churn. We did not reject any case for the New history type.

TABLE V
INTENSITY OF CHANGES - NO SEPARATION OF GENERAL DATASET

Agg. Types	Cliff's D		
	#Add	#Del	Churn
Het-Hom	-0.27		-0.08
Het-Isol	0.26	-0.11	
Het-Clean	0.99	0.89	0.95
Hom-Isol	0.55		
Hom-Clean	1.0	0.75	0.9
Isol-Clean	1.0	0.75	0.9

TABLE VI
INTENSITY OF CHANGES - SEPARATION OF GENERAL DATASET

Agg. Types	Dead			Modified		
	#Add	#Del	Churn	#Add	#Del	Churn
Het-Hom				-0.27		
Het-Isol				0.26	-0.11	
Het-Clean	0.99		0.98	0.99	0.89	0.95
Hom-Isol				0.55		
Hom-Clean				1.0	0.75	0.9
Isol-Clean	0.99	0.98	0.99	1.0	0.85	0.97

From Table VI, we can observe that most rejections were for the Modified dataset. Curiously, for the Modified dataset, when compared to Homogeneous and Isolated, Heterogeneous type had negative effects for #Add and #Del, respectively. Observing our data, the effects for the Modified classes are very similar to those of the No Separation Dataset. For the Dead dataset, we have only found statistical differences for Het-Clean and Isol-Clean, all effects are large and positive towards the smelly class.

Finding 6: When separating the dataset, we could find similar results to those of the No Separation, indicating that the Modified dataset mostly impacts the results found.

F. Intensity of Changes by System

When analyzing the proportion of changes by system and with no data separation, we have found that for most systems we could reject the H_0 for the #Add modification type. Considering the systems that rejected the H_0, 11 out of 30 systems had Isol-Clean as their largest effect, and 8 for both Het-Clean and Hom-Clean. Most of the effects were Large, except Het-Isol (0.27) for *rocketmq* and Isol-Clean (0.32) for *spring-cloud*, with *rocketmq* also presenting a negative effect towards the Heterogeneous agglomeration in the Het-Hom pair (-0.92). We have found five rejections for #Del: Het-Clean (0.59) for *easyexcel* system, and Isol-Clean (0.98) for *easyexcel*, Het-Clean (0.3) and Isol-Clean (0.23) for *jenkins* system, and Het-Clean (-0.3) for *redisson*. Meanwhile, for Churn, we could reject for the pair Het-Hom (-0.8) for *arthas*,

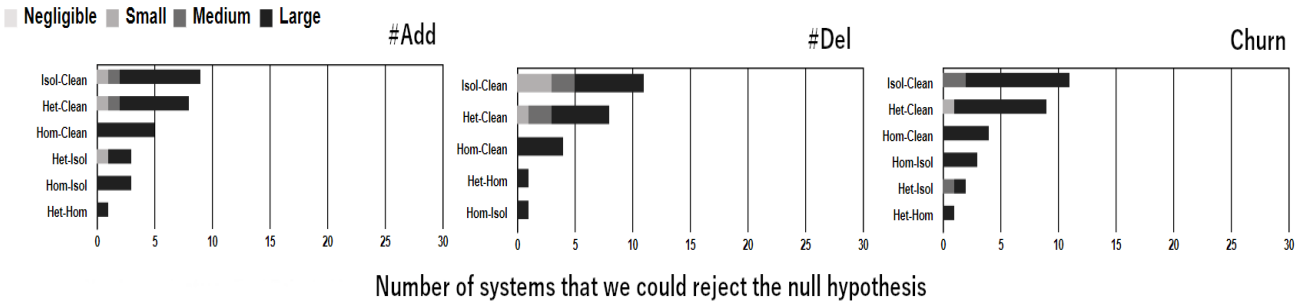

Fig. 6. Summary of Intensity of Changes by Systems - Modified

Isol-Clean (0.44) for *fastjson*, Isol-Clean (0.94) for *nanohttpd*, and Het-Clean(0.74) in *spring-cloud*.

Finding 7: We conclude that Heterogeneous, Homogeneous, and Isolated classes change in more intensity than Clean classes, with a high presence of Large effects for #Add and Churn.

Meanwhile, when separating the datasets according to their class history type, we obtained the following results. For the Dead dataset, we could only reject the H_0 for the Isolated-Clean pair, and for only five systems: for the #Add, we found differences for *arthas* (0.64), *libgdx* (0.96), *rocketmq* (0.91), and *sentinel* (0.95); for #Del we found difference only for *libgdx* (0.95); and finally, for Churn we found statistical differences for *arthas* (0.62), *jenkins* (0.97), *libgdx* (0.96), *rocketmq* (0.87), and *sentinel* (0.95). This also contributes to the evidence that for some systems, the smelly classes have so many lines added to their code, that developers prefer to delete them instead of refactor them.

Figure 6 presents a summary of the results found for the Modified dataset, and it is organized as Figures 3 and 4. We can observe that, in total, we could mostly reject the H_0 for the pairs Isol-Clean and for Het-Clean, in the three modification types. Curiously, we have found for *dbeaver* and *rocketmq* negative effects towards the smelly class, even when compared to Clean classes. When analyzing their entries we identified that, for both systems, their LOC distribution and variability is impacting the observed results.

RQ2: *Do Heterogeneous and Homogeneous Agglomerations undergo changes in more intensity compared to Isolated and Clean types?* In the General perspective, we could observe that smelly classes tend to change in more intensity in the three modification types, with the presence of Large Effects. We could also observe similar results for the Modified and No Separation dataset. Meanwhile, when observing the results by system, for all three modification types we could not reject H_0 for more than 50% of the systems. We provide evidence that, in general, smelly classes change in more intensity than clean ones.

V. Threats to Validity

We opted to collect commit information about the systems over a time-span of two years, one year before and one year after the release. Our findings may change if we consider a different time range. We selected this range to provide evidence of stability in current open-source development practices. Commits that are too old may not reflect the current functionalities of the class.

Our results may be affected by the systems that compose our dataset. To obtain generalized results, we took the following measures: (I) We selected 30 top-starred Java systems from GitHub. They vary in size, domains, maturity, and number of smells found. (II) We focused on systems with practical applicability, *i.e.,* its purpose is not educational. (III) We selected systems currently maintained by the open-source community. With these considerations, we believe we can provide evidence of how agglomerations change in the current open-source scenario.

Another important decision that can affect internal validity is the used detection tools. To mitigate this bias (I) we use currently maintained tools; (II) we use a voting strategy based on the ensemble concept of the machine learning context [38]. Consequently, we focus on the agreement of the tools. Furthermore, different perspectives of analysis can lead to different insights. In our work, we cover four different perspectives of changes and answer our research questions based on the combination of our findings. Even though our work has generalization limitations, we provide a robust and detailed study design, allowing further replications to consider new detection tools, systems, languages, and measurements.

One possible threat to our work is the choice of the statistical framework. Since our dataset is non-normal, we chose a non-parametric test, Mann Whittney-U, and a non-parametric measure of effect size, Cliff's Delta. We also calculated for our analysis their distribution to gain further insight into our findings. We did not use any p-value correction, since we compare the results in pairs, and corrections are used to compare multiple categories at once.

We consider the generalization of our results, presenting the results by the system and considering the history of all systems. We further analyze the impact on the results when a class was added or removed in the two-year time-span. We

believe that when considered in combination, these multiple analyses provide reliable conclusions about the changes in the agglomerations. To avoid errors, we have used well-known algorithms from the Python library to both mine and analyze our data. Finally, all the necessary steps and scripts used in our work are available in our online appendix, allowing replications and extensions of this work.

VI. Related Work

Several studies in the literature aimed at identifying how code smells and design pattern co-occurs [15, 16, 18, 19, 39–42]. Lanza and Marinescu [15] introduced the concept of Collaboration Disharmonies, in which code smells were identified through the use of metrics of coupling. Oizumi et al. [16] raised evidence of how code smells agglomerations help developers identify design problems. Oizumi et al. [18] studied how agglomerations and architectural problems are related, and found that agglomerations are a good indicator of architectural problems. Santana et al. [19] explored how different agglomerations impact five modularity metrics. They have found that agglomerations composed of two or more smell types impact the complexity and coupling metrics.

Lozano et al. [42] studied when code smells co-exist and co-disappear. Palomba et al. [41] identified which code smell agglomerations are more frequent through the use of association rules. Later, Palomba et al. [43] investigated how agglomerations are spread along the systems. Yamashita and Moonen [20] found that some agglomerations types impact the maintenance activity in an industrial context. Walter et al. [39] compared agglomerations in several system domains, and presented a comparison between the agglomerations found by them and in the literature.

Change analysis is multifaceted and several approaches to quantify it were proposed in the literature [44]. Khomh et al. [8] investigated the relationship between anti-patterns and change-proneness, and if anti-patterns suffered more structural changes than other classes. They discovered that some smells tend to change more frequently than classes that do/do not participate in other anti-patterns, and they usually impact more their interface. Palomba et al. [6] found, in the context of Android smells and object-oriented smells, that when a class has more than one smell, they are more prone to faults and to be changed.

Jaafar et al. [45] investigated how prone to faults classes that co-changes with anti-patterns are. To achieve their goals, they evaluated 10 anti-patterns. They found that presenting LC, LM, and RB, when co-changed with other classes, the likelihood of them presenting a fault is 2.5 times higher than other non-smelly classes. Mondal et al. [46] investigated if duplicated code exhibits more instability than non-duplicated code. For this purpose, they investigated eight different change metrics. They have found that cloned code has a higher likelihood of getting changed, mainly for Java and C languages.

More recently, Oliveira et al. [47] explored the frequency and impact of changes related to exception handling, *i.e.,* robustness changes, co-occur with code smells. They have found that a relationship between the changes and code smell occurrence exists, mainly for Long Method, Feature Envy, and Dispersed Coupling. They also found that 16.26% of the robustness changes co-occurred with the introduction of smells. Trautsch et al. [48] investigated the impact and differences of perfective (*i.e.,* aims at improving code quality), corrective (bug fixing), and other changes. They found that, usually, (i) perfective and corrective changes add fewer lines of code; (ii) perfective changes have a positive impact on the code quality.

Our work complements the existing literature by providing evidence of how different code smell agglomerations change along the system's history. We quantitatively analyze four different measurements of changes from two perspectives: general and by system. We also fill a gap in the literature on the change-proneness of code smell agglomerations, since isolated smells usually are the subject of research.

VII. Conclusion

Our main goal was to provide empirical evidence of how different agglomeration types behave in terms of code stability, in a two-year time-span. We have found empirical evidence that smelly classes (Heterogeneous, Homogeneous and Isolated smells) change more often than the Clean type. Nevertheless, the effects found were mostly Negligible and Small. When compared to Isolated smells, Heterogeneous agglomerations indeed change in more frequency. In terms of the intensity of the changes, we have found that smelly classes change their code in more intensity than Clean ones, and when comparing to the Isolated type, both Heterogeneous and Homogeneous agglomerations had a positive effect for the #Add modification type. Our findings indicate that even though agglomerations are rarer in the source code, they are changed in more frequency and intensity. In conclusion, we provide additional evidence to practitioners that agglomerations are unstable, and consequently, that they affect the maintenance and evolution process.

In future work, we plan to investigate how the different types of Heterogeneous, Homogeneous, and Isolated classes change when compared to Clean classes; and consider other change measurements. We also highlight some potential for further exploration of this research: (I) consider agglomeration changes in the industrial context, allowing us to compare the results we found for the open-source context; (II) analyse the complete history of the systems; and (III) consider other programming languages, and consequently, evaluate other agglomerations composed of smells that are specific to the languages.

References

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

[2] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, Aug 2013.

[3] A. Uchôa, C. Barbosa, D. Coutinho, W. Oizumi, W. K. Assunçao, S. R. Vergilio, J. A. Pereira, A. Oliveira, and A. Garcia, "Predicting design impactful changes in modern code review: A large-scale empirical study," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 471–482.

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

[5] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, Oct 2018.

[6] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," in *2018 IEEE/ACM 40th International Conference on Software Engineering. (ICSE)*, May 2018, pp. 482–482.

[7] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *2010 IEEE International Conference on Software Maintenance*, Sep. 2010, pp. 1–10.

[8] F. Khomh, M. Di Penta, Y. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, p. 243–275, Jun. 2012.

[9] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Transactions Software Engineering Methodology.*, vol. 23, no. 4, Sep. 2014.

[10] D. Falessi and R. Kazman, "Worst smells and their worst reasons," in *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2021, pp. 45–54.

[11] 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 *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 403–414.

[12] J. Tan, D. Feitosa, and P. Avgeriou, "Do practitioners intentionally self-fix technical debt and why?" in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 251–262.

[13] A. Decan, T. Mens, P. R. Mazrae, and M. Golzadeh, "On the use of github actions in software development repositories," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 235–245.

[14] M. Abbes, F. Khomh, Y. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *2011 15th European Conference on Software Maintenance and Reengineering*, March 2011, pp. 181–190.

[15] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2005.

[16] W. Oizumi, A. Garcia, L. d. S. Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 440–451.

[17] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 101–110.

[18] 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.

[19] A. Santana, D. Cruz, and E. Figueiredo, *An Exploratory Study on the Identification and Evaluation of Bad Smell Agglomerations*. Association for Computing Machinery, 2021, p. 1289–1297.

[20] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 682–691.

[21] E. V. d. P. Sobrinho, A. De Lucia, and M. d. A. Maia, "A systematic literature review on bad smells — 5 w's: which, when, what, who, where," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[22] [Online]. Available: https://zenodo.org/records/10939220

[23] M. Aniche, *Java code metrics calculator (CK)*, 2015, available in https://github.com/mauricioaniche/ck/.

[24] X. Han, A. Tahir, P. Liang, S. Counsell, and Y. Luo, "Understanding code smell detection via code review: A study of the openstack community," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 323–334.

[25] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010, pp. 336–345.

[26] L. Madeyski and T. Lewowski, "Detecting code smells using industry-relevant data," *Inf. Softw. Technol.*, vol. 155, no. C, mar 2023. [Online]. Available: https://doi.org/10.1016/j.infsof.2022.107112

[27] T. Paiva, A. Damasceno, J. Padilha, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *Journal of Software Engineering Research and Development (JSERD)*, 2017.

[28] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment*

in Software Engineering, ser. EASE '16, 2016.

[29] G. Santos, A. Santana, G. Vale, and E. Figueiredo, "Yet another model! a study on model's similarities for defect and code smells," in *Fundamental Approaches to Software Engineering*, L. Lambers and S. Uchitel, Eds. Cham: Springer Nature Switzerland, 2023, pp. 282–305.

[30] D. Cruz, A. Santana, and E. Figueiredo, "Detecting bad smells with machine learning algorithms: An empirical study," in *Proceedings of the 3rd International Conference on Technical Debt*, ser. TechDebt '20. Association for Computing Machinery, 2020, p. 31–40.

[31] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1037–1039.

[32] G. d. F. Carneiro, M. Silva, L. Mara, E. Figueiredo, C. Sant'Anna, A. Garcia, and M. Mendonça, "Identifying code smells with multiple concern views," in *2010 Brazilian Symposium on Software Engineering*, 2010, pp. 128–137.

[33] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *Int'l Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 612–621.

[34] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 268–278.

[35] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *2016 IEEE 24th international conference on program comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[36] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2020.

[37] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[38] H. J. K. M. D. Mining, *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2001.

[39] B. Walter, F. A. Fontana, and V. Ferme, "Code smells and their collocations: A large-scale experiment on open-source systems," *J. Syst. Software*, vol. 144, pp. 1–21, 2018.

[40] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter, "Inter-smell relations in industrial and open source systems: A replication and comparative analysis," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 121–130.

[41] F. Palomba, R. Oliveto, and A. De Lucia, "Investigating code smell co-occurrences using association rule learning: A replicated study," in *2017 IEEE Work. on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, Feb 2017, pp. 8–13.

[42] A. Lozano, K. Mens, and J. Portugal, "Analyzing code evolution to uncover relations between bad smells," 03 2015.

[43] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Information and Software Technology*, vol. 99, pp. 1–10, 2018.

[44] M. Kretsou, E.-M. Arvanitou, A. Ampatzoglou, I. Deligiannis, and V. C. Gerogiannis, "Change impact analysis: A systematic mapping study," *Journal of Systems and Software*, vol. 174, p. 110892, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412122030282X

[45] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, F. Khomh, and M. Zulkernine, "Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults," *Empirical Softw. Engg.*, vol. 21, no. 3, p. 896–931, jun 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9361-0

[46] M. Mondal, M. S. Rahman, C. K. Roy, and K. A. Schneider, "Is cloned code really stable?" *Empirical Softw. Engg.*, vol. 23, no. 2, p. 693–770, apr 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9528-y

[47] A. Oliveira, J. Correia, L. Sousa, W. K. G. Assunção, D. Coutinho, A. Garcia, W. Oizumi, C. Barbosa, A. Uchôa, and J. A. Pereira, "Don't forget the exception! : Considering robustness changes to identify design problems," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 417–429.

[48] A. Trautsch, J. Erbel, S. Herbold, and J. Grabowski, "What really changes when developers intend to improve their source code: a commit-level study of static metric value and static analysis warning changes," *Empirical Software Engineering*, vol. 28, pp. 1–40, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:248405991