AGLOMERAÇÕES DE ANOMALIAS DE CÓDIGO E SEU IMPACTO NA MODULARIDADE DE SOFTWARE

AMANDA DAMASCENO SANTANA

AGLOMERAÇÕES DE ANOMALIAS DE CÓDIGO E SEU IMPACTO NA MODULARIDADE DE SOFTWARE

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Eduardo Magno Lages Figueiredo

Belo Horizonte, Minas Gerais

Novembro de 2020

AMANDA DAMASCENO SANTANA

BAD SMELL AGGLOMERATIONS AND THEIR IMPACT ON SOFTWARE MODULARITY

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Eduardo Magno Lages Figueiredo

Belo Horizonte, Minas Gerais

November 2020

Santana, Amanda Damasceno

S232b Bad Smell Agglomerations and their Impact on Software Modularity / Amanda Damasceno Santana. — Belo Horizonte, Minas Gerais, 2020
xxii, 77 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of Minas Gerais
Orientador: Eduardo Magno Lages Figueiredo

1. Computação – Teses. 2. Bad smell – Teses . 3. Aglomeração – Teses. 4. Anomalias de código (Engenharia de software) – Teses. 5. Software -Reutilização – Teses. I. Figueiredo, Eduardo Magno Lages. II.Título.

CDU 519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Bad Smell Agglomerations and their Impact on Software Modularity

AMANDA DAMASCENO SANTANA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Eduardo Figueiredo
PROF. EDUARDO MAGNO LAGES FIGUEIREDO - Orientador
Departamento de Ciência da Computação - UFMG

André Cavalcante Hora
PROF. ANDRÉ CAVALCANTE HORA
Departamento de Ciência da Computação - UFMG

PROF. MARCELO DE ALMEIDA MAIA
Faculdade de Computação - UFU

Belo Horizonte, 30 de Outubro de 2020.

Dedico esta dissertação a minha mãe, Monica, que sempre me apoiou em todo o meu percurso de graduação e do mestrado, sempre me dando muito carinho, suporte emocional, e fazendo de tudo a seu alcance para que eu pudesse chegar aqui. Dedico também a minha irmã Aline, por todo o apoio dado ao longo desta dissertação, a seu carinho, atenção dadas as minhas apresentações, e por todas discussões a respeito do meu trabalho. Também dedico ao meu falecido pai, Velsser, que infelizmente não pode me ver entrando no mestrado, nem o concluindo. Contudo, sei que aonde é que você esteja, está torcendo por mim. Agradeço ao meu namorado, Newton, por todo apoio dado durante este percurso, por sempre ter me incentivado e me dito que tudo vai dar certo. Dedico também a Malu, Mabel e Ruffus, por seu amor incondicional. Obrigada, meus amores, pois sem vocês eu não teria conseguido chegar até aqui, e muito menos teria passado pelos meus momentos difíceis.

Agradeço ao meu orientador, Eduardo, por toda a paciência que teve comigo nesta longa jornada. Agradeço por todo conhecimento passado, pela compreensão nos momentos mais difíceis em minha vida, por sua positividade em relação ao meu trabalho, e por ter me dado a oportunidade de ingressar no meio científico. Agradeço também todos os membros do Labsoft, por todo o feedback dado, principalmente ao Cleiton e ao Daniel, que tivemos oportunidades de trabalharmos juntos durante meu mestrado.

Agradeço também a todos os professores que fizeram parte do meu percurso curricular, desde a graduação. Guardo com muito carinho seus ensinamentos, e toda a experiência me permitiu crescer como pessoa e como pesquisadora. Agradeço a equipe da Pós-Graduação, em particular a Sônia e o Clodoveu por todos os esclarecimentos. Finalmente, agradeço a Capes, cuja bolsa me possibilitou contribuir para a ciência brasileira.

Resumo

A maioria dos sistemas devem evoluir a fim de implementar novos requisitos dos stakeholders, ou para corrigir problemas existentes. Estas mudanças são complexas devido a diversos fatores, incluindo a necessidade de compreender o código fonte do sistema, atividade que é prejudicada pela presença de anomalias de código. Anomalias de código são sintomas de más decisões sobre o projeto do sistema ou simplesmente de seu código. Contudo, quando duas ou mais anomalias acontecem no mesmo pedaço de código, elas formam uma aglomeração. Consequentemente, desenvolvedores precisam se esforçar muito mais para realizar suas atividades de desenvolvimento e manutenção. Neste trabalho, nós avaliamos aglomerações formadas por quatro tipos de anomalias: Large Class, Long Method, Feature Envy e Refused Bequest. Nosso objetivo é avaliar como estas aglomerações estão espallhadas no código fonte, e como elas impactam na modularidade do software. Nossos resultados são alcançados através do uso de regras de associação e medidas de tamanho do efeito. Nós encontramos que classes com duas ou mais anomalias são frequentes no código fonte, até mesmo quando consideramos somente o mesmo tipo de anomalia. Elas também são altamente espalhadas no código fonte, mesmo quando o tamanho do sistema é levado em consideração. Também, nós descobrimos que elas realmente impactam a modularidade. Estes resultados significam que elas podem afetar a complexidade da classe, torná-las mais propensas a problemas, e mais acopladas a outras classes.

Abstract

Most systems must evolve to cope with new stakeholders requirements, or to fix existing problems. These changes are complex due to several factors, including the need of understanding the source code, activity that is impaired by the presence of bad smells. Bad smell is a symptom of bad decisions about the system design or code. When two or more bad smells occur in the same snippet of code, they form an agglomeration. Consequently, developers need to put more effort to perform their development and maintenance tasks. In this work, we evaluate agglomerations composed of four kinds of bad smells: Large Class, Long Method, Feature Envy, and Refused Bequest. We aim at evaluating how these agglomerations are spread in the source code, and how they impact on the software modularity. Our results are achieved through the use of association rules and effect size measurements. We have found that classes with two or more smells are frequent in the source code, even when the smells present in the class are of the same type. We also observed that agglomerations are the most spread in the source code, even when the size of the systems are taken into account. Agglomerations have a significant effect on most analyzed modularity metrics. This result means that they may affect class complexity, fault proneness, and coupling.

List of Figures

List of Tables

List of Symbols

1. *CBO* Coupling Between Objects
2. *CI* Confidence Interval
3. *CV* Coefficient of Variation
4. *DIT* Depth of Inheritance Tree
5. *FE* Feature Envy
6. *IQR* InterQuartile Range
7. *Isolated FE* Isolated Feature Envy
8. *Isolated LC* Isolated Large Class
9. *Isolated LM* Isolated Long Method
10. *Isolated RB* Isolated Refused Bequest
11. *Homogeneous FE* Homogeneous Feature Envy
12. *Homogeneous LM* Homogeneous Long Method
13. *KLOC* 1000 Lines of Code
14. *LC* Large Class
15. *LOC* Lines of Code
16. *LM* Long Method
17. *maxNest* max nest blocks
18. *NOC* Number of Classes
19. *NOM* Number of Methods

20. *OR* Odds Ratio

21. *RB* Refused Bequest

22. *RFC* Response set of a class of objects

23. *RQ* Research Question

24. χ^2 Chi Square Test

25. *std. dev.* Standard Deviation

26. *WMC* Weighted Method Class

Contents

Chapter 1

Introduction

Most software systems must evolve to cope with new requirements or to address existing code problems and bugs [30]. However, evolving systems is a challenging task, due to the growing system complexity, leading developers to take sub-optimal decisions about the design [58]. In order to make the required changes, developers need first to understand the source code, identifying how the modules of the system interact to avoid breaking the application. In a recent study [63], the authors found that developers spend up to 58% of their time understanding the source code in an industrial setting. This increased effort is affected by the existence of technical debts, that, beyond making the system harder to understand, increase the complexity of the interaction between system modules.

A *technical debt* (TD) is a metaphor in which developers (un)awaringly take sub-optimal decisions in order to speed the development process [10]. Even though these decisions bring benefits to developers in short term, in long term, due to the accumulation of debts, they lead to a more inflexible source code that is more difficult to comprehend, to extend, and to maintain. Consequently, the debts increases the software maintenance costs [6].

In a systematic mapping, Li et al. [31] raised primary studies that evaluate the impact of TDs in the system quality, and found several works on their impact on maintainability. Most of the studies identified by the authors focused on evaluating changeability. Besker et al. [5] evaluated how TD affects the project at different development stages. The authors found that understanding and measuring the TD effects take most of developers' time. They also have found that, overall, 36% of development time is used to address TDs.

In this dissertation, we are interested in evaluating bad smells, symptoms of TD occurrence [21]. *Bad smell* is a symptom that indicates that the code needs to be

refactored. They were proposed as a mean of identifying the portions of code that presents decaying quality. Bad Smell affect several properties of software modularity, such as understandability, extensibility, and reusability [22; 64]. Consequently, they affect directly on the lifecycle activities of development. To pay this debt, developers may refactor the smelly code. *Refactoring* is an activity in which modifications in the source code are made to improve its internal quality. Yet, it does not change the external behavior of the system [21]. These modifications may involve removal of duplication, code simplification, or clarification of unclear code [26]. Moreover, studies in the literature found that bad smells tend to persist for all versions of the system after being introduced in the source code [7]. This implies that removing bad smell from the code is not a trivial task.

Several studies tried to understand the impact of bad smell instances on different quality attributes, such as error proneness, change proneness [4; 23; 27; 43; 46], and maintenance effort [53; 65]. However, there are evidences that when two or more bad smell instances occur together in the source code, for example, in the same component or through a coupling relationship, they make the code even harder to understand and to maintain [1; 29; 39; 47]. When two or more bad smells occur together in the same piece of code, they form an *agglomeration*. Sobrinho et al. [11] raised in their literature review several works that aimed at co-studying bad smells. In this work, we extend the understanding of agglomerations filling some of the gaps found in their work. In this dissertation, we investigate how classes that are agglomerations, *i.e.*, classes that have two or more instances of bad smells, impact on the source code modularity.

Even though bad smells are extensively researched in the literature [11], bad smell agglomerations only recently started getting the necessary attention. Most of the studies that investigated these agglomerations focus on identifying which of them are the most common in the source code [49; 50; 61; 67]. Few studies try to understand how they impact on aspects of the source code quality [12; 41; 46] and their lifecycle [35; 45]. There is also an effort to understand the relationship between bad smell agglomerations, architectural problems, and design problems [12; 41; 42].

It is worth mentioning that these studies rely on small datasets, composed of few open source systems. They focus on evaluating bad smells that are extensively investigated in the literature, such as Large Class and Long Method [11]. The need of more understanding on the impact of such agglomerations on the source code modularity motivated us to investigate this research topic. This work contributes to expanding the current knowledge by providing an evaluation of bad smell agglomerations at the level of class in a larger dataset. It reports the density of agglomerations in the source code and how they are affecting the system modularity through the use of software

metrics. We also evaluate not only common co-studied bad smells, such as Large Class and Long Method, but also smells that are not largely co-investigated, such as Feature Envy and Refused Bequest.

1.1 Methodology

In this section, we aim at briefly explaining how this dissertation was conducted. We first created a dataset composed of 20 open-source Java systems from the Qualita Corpus [56], with different sizes and domains. To create this dataset, we identified with the help of six automatic detection tools four bad smells: Large Class, Long Method, Feature Envy and Refused Bequest. Each class in our dataset was classified according with the smell that it contains. We consider as an agglomeration a class with two or more bad smell instances. Since we are creating a new dataset, we evaluated this dataset, with the help of six post-graduated students that had experience in detecting bad smells, by calculating the agreement between our employed detection strategy and the opinion of the evaluators about the presence of the smell in the class/method.

After building and evaluating our dataset, we are now prepared to identify and verify how the agglomerations impact aspects of source code modularity. In order to identify the agglomerations, we used association rules and frequency statistics. The former was used to identify agglomerations composed of at least two bad smells types. The latter was used to identify agglomerations composed of only one smell type, but that has two or more instances of the smell. We opted to use a simple technique to identify agglomerations of only one smell type due to the restriction of the association rule that does not allow a bad smell being in the two sides of the rule at the same time. Consequently, the association rule does not detect agglomerations formed of only one bad smell type.

We then analyze how is the density of agglomerations in each system. We considered their density in terms of absolute numbers, number of classes of each system, and KLOC. Finally, we explored the impact of the agglomerations on the modularity metrics, comparing them to classes that do not have bad smell or that has only one bad smell instance. In order to compare them, we calculated five modularity metrics for each class, summarized them in terms of mean, medians, quartiles and dispersion, and then used the Cohen's d values [9] and the Coefficient of Variation (CV) [52] in order to compare the categories. We repeated this analysis in order to verify how agglomerations composed of two or more bad smell types compare to each other. This analysis was made to verify if such agglomerations present a similar behavior in terms

of density and impact.

1.2 Contributions

In this dissertation, we can highlight the following contributions.

- We have built and validated a dataset composed of 20 open-source Java systems with different sizes and domains. We also presented the list of the four bad smell found in this dataset.
- We proposed a new methodology to automatically construct bad smell datasets. This methodology combines the vote of three tools to assure that different detection strategies are being employed. It can also be easily extended to address new systems and bad smells.
- We expanded the current knowledge of bad smell agglomerations by providing evidences that bad smell agglomerations does impact the source code modularity in a larger dataset, and contemplating bad smells that were not co-studied in depth.

Partial results of this dissertation have been published or submitted to publication as follows.

- Santana, A. and Figueiredo, E. On the Impact of Bad Smell Agglomerations on Software Quality. On the Impact of Bad Smell Agglomerations on Software Quality. In: Master and PhD Workshop on Software Engineering (WTDSoft), co-allocated with CBSoft, 2019, Salvador. Proceedings of the Master and PhD Workshop on Software Engineering (WTDSoft). Porto Alegre: SBC, 2019. v. 1. p. 1-7.
- Santana, A., Cruz, D. and Figueiredo, E. 2020. An Exploratory Study on the Identification and Evaluation of Bad Smell Agglomerations. Submitted to an international conference.
- Silva, C., Santana, A., Figueiredo, E., and Bigonha, M.. Revisiting the Bad Smell and Refactoring Relationship: A Systematic Literature Review. In: In proceedings of the 23rd Iberoamerican Conference on Software Engineering (CIbSE), 2020, Curitiba. Experimental Software Engineering Track (ESELAW). Berlin: Springer, 2020. v. 1. p. 1-14.

- Cruz, D., Figueiredo, E., and Santana, A. Detecting Bad Smells with Machine Learning Algorithms: an Empirical Study. In: International Conference on Technical Debt (TechDebt), 2020, Seoul. Proceedings of International Conference on Technical Debt (TechDebt). New York: ACM, 2020. v. 1. p. 1-10.

1.3 Dissertation Outline

The remainder of this dissertation is structured as follows.

Chapter 2 presents the necessary background information that is used along this dissertation, such as the concepts of the bad smells evaluated and the bad smell agglomerations types. This section presents a classification for the agglomeration types, beyond the ones proposed in the literature. Finally, we aim at discussing related works and how this dissertation compares to them.

Chapter 3 presents in details the methodology employed in the creation of our dataset. First we explained the selection of which systems to evaluate, and how we detected the bad smells in order to obtain a list of bad smells for each system. We then manually evaluated a statistical sample of our dataset and calculated the agreement between the evaluation and the results obtained by our detection strategy. Finally, we discuss some threats to our dataset construction.

Chapter 4 presents in details the methodology employed to answer our research questions. We first present the research questions and their respective motivations. We then explain how bad smell agglomerations were discovered in our dataset. With these agglomerations in hand, we explain how the necessary information will be extracted in order to evaluate the impact of agglomerations on modularity metrics. We also explain how the agglomeration density was calculated and analyzed. Finally, we present some threats related to our methodology.

Chapter 5 presents the observed results and their implication for practitioners. We discuss the presence of bad smell agglomerations in the source code. We then analyze their density in the source code. With these steps concluded, we discuss how they are impacting the selected modularity metrics. Finally, we discuss our results and their implications for practitioners.

Chapter 6 presents the observed results for the agglomerations composed of two or more bad smell types. In this chapter, we aim at understanding if there is difference in their density and on how they impact the source code modularity. This analysis also contemplates agglomerations that did not appear in the association rules algorithm.

Chapter 7 presents our main findings and conclusions, along with possibilities of future work.

Chapter 2

Background

The main objective of this dissertation is to identify bad smell agglomerations at the class level and understand how they impact the source code modularity. Bad smells are structures in the code that suggest the need of quality improving. This improvement can be achieved using refactoring activities. However, when bad smells appear together in the source code, forming an agglomeration, the process of enhancing the code quality is severely affected by their existence. This is due to their interaction in the code.

In order to achieve our objective, we aim at explaining in this section the main concepts used in this dissertation. Section 2.1 describes and exemplify the bad smells under evaluation, and why they were chosen. Section 2.2 describes the possible types of bad smell agglomerations that may occur in the source code, and further expand the original classification by Oizumi et al. [39] to address the agglomerations found in this work. Section 2.3 describes the algorithm used to identify the bad smell agglomerations. Finally, Section 2.4 provides an overview of related works and how they differ from this dissertation.

2.1 Bad Smells

Bad smells are symptoms, or evidences, in the source code that indicate the need of refactoring, *i.e.*, the opportunity of improving the internal quality of the system [21]. Bad smells are not necessarily harmful [20; 43; 54]. They can even represent a good practice in some implementation domains, such as parsers and databases [19]. In their work, Taibi et al. [54] found which bad smells proposed by Fowler [21] are considered harmful in the opinion of experienced developers. Three of the four smells analyzed in this dissertation are among the most harmful ones. They are: Long Class, Long Method and Refused Bequest, with 100% of the developers agreeing that they are

harmful. From a scale of 1 to 5, the most harmful of the analysed smells was Long Class, with an value of 4.5. This work provided us with information to help to select which bad smell to co-study, and we opted to select those that were considered as harmful.

Despite of being greatly researched, Sobrinho et al. [11] found that only a small fraction of the 22 bad smells originally proposed by Fowler [21] have been extensively co-studied. This work motivated us in selecting which bad smell to co-study, trying to balance the choice between the most and the least studied bad smells. Beyond these decisions, we also focused in selecting those that are detected by at least three detection tools, as described later in Section 3.2. This assure us that our dataset is built using different detection strategies. It is worth to notice that when calculating the intersection between the results of the tools, the more tools we add to detect smell S, the narrower the results will be. Consequently, we are having more certainty that the smell S is a true positive. So we compromised, and decided that for the bad smell selection, each smell must be detected by exactly three tools.

Table 2.1 briefly conceptualizes the selected bad smells according to their definitions by Fowler [21]. Large Class and Long Method are among the most studied smells. We selected these smells to co-study them with Feature Envy and Refused Bequest, because they affect known principles of good object oriented design. For instance, Refused Bequest violates the Liskov Substitution Principle [32]. This principle indicates that a subclass should substitute its parent class, for instance, in a method call. This principle implies that the subclass correctly inherited its parent behavior. This situation is the opposite of when a Refused Bequest occurs in the hierarchy.

Table 2.1: Studied bad smell

Bad Smell	**Definition**
Large Class (LC)	It occurs when a class tries to do too much, presenting to many instance variables
Long Method (LM)	It occurs when methods are too large, frequently necessitating long explanations of how it works.
Feature Envy (FE)	It occurs when a method seems more interested in being in other class.
Refused Bequest (RB)	It occurs when a class does not want their parent behaviour.

In this dissertation, we consider that Large Class, Brain Class and God Class are the same bad smell. Their original definition varies only in small details, for example, the use of external variables. The same occurs with the Long Method and God Method smells. Both have a similar concept, and we opted to consider both as the same smell. This choice was due to the similarity on their definition. If tools uses different strategies

to detect them, we benefit from it by allowing the dataset construction to range different perspectives of the smell. This strategy indeed benefit us, since we could obtain in the dataset a reasonable number of Large Class and Long Method smells.

This paragraph will exemplify the two bad smells that are at the class level, the Large Class and Refused Bequest. One example of Large Class is the class *JspConfig*[1] from the *Tomcat* system. Beyond having 11 attributes, 7 methods and 462 lines of code, the class has 2 public inner classes. Out of the 7 methods, 2 of them are very large. However, the functionality of the class is consistent, aiming at creating and manipulating a Jsp configuration. An example of Refused Bequest is the class *ActiveXObject*[2] in the *HtmltUnit* system. The class extends the behavior of the class *SimpleScriptable*[3], a base class for Rhino host objects. *ActiveXObject* is an active host that allows people to instantiate java objects using JavaScript.

Even though the business of both classes seems consistent, *ActiveXObject* do not use the behavior of *SimpleScriptable*. The following evidences suggest this affirmation. (i) The class overrides only one method, the *getClassName()*, changing the returned string. (ii) *ActiveXObject* mostly uses methods and attributes of the *SimpleScriptable* as parameters for its methods, not using its inherited behavior. (iii) There is no reference to the super class. (iv) *ActiveXObject* uses more the behavior of the *Scriptable* than of its parent.

We now exemplify the smells at the method level, the Long Method and Feature Envy. As an example of the Long Method smell, the method *XYDifferenceRenderer.drawItemPass0*[4] from *JFreeChart* system, a method that draws on the screen a single data item. Beyond receiving 10 parameters and their functionality being distributed in more than 400 lines of code, the method uses heavily temporary variables, conditional statements and control sentences. Besides, most of the lines of code consist of creating new variables or making simple computations. These evidences suggest that the code could be refactored, using the *Extract Method* refactoring process. Finally, an example of Feature Envy smell is the method *CSVSaver.getCapabilities()*[5] from the *Weka* system. The class consist of, as the name suggest, saving in batches a CSV file. However, when observing this method, all it does is to enabling capabilities (*Capabilities* class). It seems that this method is more interested in being in the *Capabilities* class than in the *CSVSaver*.

[1] `https://github.com/amandads/Dissertation/blob/master/JspConfig.java`
[2] `https://github.com/amandads/Dissertation/blob/master/ActiveXObject.java`
[3] `https://github.com/amandads/Dissertation/blob/master/SimpleScriptable.java`
[4] `https://github.com/amandads/Dissertation/blob/master/XYDifferenceRenderer.java`
[5] `https://github.com/amandads/Dissertation/blob/master/CSVSaver.java`

2.2 Classification of Bad Smell Agglomerations

This work aims at understanding the presence and impact of bad smell agglomerations on the source code modularity. We study the agglomerations at the level of class: classes that contain at least two bad smell instances. In their book, Lanza and Marinescu [29] introduced the concept of collaboration disharmonies. The authors build a diagram of relationship between the smells, and found relations such as "is", "uses", "has", and "partially". In this work, we refer to collaboration disharmonies as bad smell agglomerations.

In the literature, it was identified two kinds of general agglomerations: intra-component agglomeration and inter-component agglomeration [39]. The first one concerns an agglomeration that occurs in the same component. The latter one concerns an agglomeration that occurs between two or more components. Here, we address the intra-component agglomerations. We adopt as component the individual classes of the systems, providing a more deeper analysis of the agglomerations behavior in different class.

The main motivation for choosing to study intra-component agglomerations is that a set of bad smells in the same piece of code provides an important evidence that developers should pay attention to the class when she/he modifies it, since it presents different problems, ranging different modularity aspects. For example, a class with both Large Class and Feature Envy implies that the class is complex, is large in terms of size and is not cohesive. Moreover, when analyzed together, the agglomeration could help developers see the extent of the modularity problem, reasoning similar to the work of Souza et al. [13]. For our analysis, it is important to further segregate the classification proposed by Oizumi et al. [39] to cope with the types of bad smells that we have found. We propose in this work two kinds of agglomerations: Heterogeneous and Homogeneous. They are described as follows.

Heterogeneous Agglomeration: it occurs when a class contains two or more instances of bad smells, but the class also contains at least two different bad smell types. For example, in the system *JMeter*, the class *ResponseAssertion*[6] is a Large Class, it refuses its parent behavior (Refused Bequest) and presents one Large Method, the *ResponseAssertion.equalsComparisonText(final String received, final String comparison)* method.

Homogeneous Agglomeration: it occurs when a class contains two or more instances of bad smells, but all these instances are of the same bad smell type. In this work, we idenfitied two types of Homogeneous Agglomeration: Homogeneous Feature

[6] `https://github.com/amandads/Dissertation/blob/master/ResponseAssertion.java`

Envy and Homogeneous Long Method. For example, the class *LUDecomposition*[7] in the *Weka* system has only two Feature Envy smells, the methods *solve(Matrix B)* that finds the matrix X that satisfies A*X = B; and *LUDecomposition(Matrix A)*, a constructor that makes several calculations and does not return anything.

Finally, we can describe the terminology employed in this dissertation to address other possibilities of the bad smell presence. *Isolated Bad Smell*: it occurs when a class contains only one instance of bad smell, *i.e.*, it is not an agglomeration. It can be further classified according to the smell: Isolated Feature Envy (Isolated FE), Isolated Long Method (Isolated LM), Isolated Long Class (Isolated LC), and Isolated Refused Bequest (Isolated RB). *Clean Class*: it defines a class with no instance of the considered bad smells.

2.3 Association Rules

Identifying agglomeration is not a simple task, since all combinations of bad smells must be considered, and not all of them are meaningful due to the context that are being evaluated. For example, a rare combination may be considered as noise in some contexts, such as costumer buying habits [24; 57]. In our context, rare combinations are interesting to be analysed, since the agglomeration presence depends on several different system characteristics, such as domain, developer's characteristics, and maturity level. Such agglomerations will be addressed separately in its own chapter. We aim at identifying relevant and frequent combinations of bad smells that occur together in the same class. Therefore, we opted to use association rules [24]. Together with meaningfulness metrics, association rules can identify significant group of items that appear together in the dataset [24].

To identify association rules, we used the Apriori Algorithm [2]. We formulate our problem as follows. Be t a transaction, *i.e.*, an instance in our dataset. Let T be our set of transactions. Be i_{j} a type of bad smell. Each transaction t has four binary variables i_{j} representing the presence/absence of the bad smell. These i_{j} are organized in the following way: Large Class, Refused Bequest, Feature Envy, and Long Method. For example, a transaction $t = [1, 1, 0, 0]$ indicates that the class has Large Class and Refused Bequest. Let L_{m} be a threshold for some metric m. We want to find the significant association rules observing all transactions in T. This can be achieved observing if the rules are above L_{m}. If the association rule is strong, it is considered in

[7] `https://github.com/amandads/Dissertation/blob/master/LUDecomposition.java`

the next step of the algorithm. In the final step of the algorithm, we expect to obtain a set of rules of the format $Antecedent \longrightarrow Consequent$.

The algorithm start calculating the metric L_{m} for all items. For example, it calculates the metric for people that bought [bread], [milk], and others, separately. If the item is not above the threshold, this means that the item is rare. Consequently, combinations with the item are rare too. For example, if in a market most people buy in the same transaction [milk and bread], but of all the considered transactions, only one person bought [detergent] in the transaction [milk, bread, detergent], combinations with [detergent] are rare. Consequently, rules that have [detergent] in the antecedent or consequent are not strong, and they are not considered in the next step of the algorithm.

This process is repeated until no new rules can be found, or the parameter that limits the maximum number of items in the Antecedent is reached. For the market example, possible rules could be $milk \longrightarrow bread$ or $bread \longrightarrow milk$, depending on the set of transactions and meaningfulness metrics used. The parameters used on the application of the Apriori Algorithm in this work is given in Section 4.2.

2.4 Related Work

Several studies tried to understand the relationship between bad smells in a class [29; 35; 39; 38; 49; 61; 67]. The first ones to identify that bad smells are related were Lanza and Marinescu [29]. The authors presented in their book the concept of Collaboration Disharmonies, in which design flaws interact and affect several entities at once. They summarized in a diagram the identified relationships between the flaws, using concepts of system modeling, such as "is", "has", and "uses". To identify such relationships, the authors used metrics of coupling.

Oizumi et al. [38] studied the relationship between agglomerations and architectural problems in seven systems, with five of them being industrial ones. The authors found that most of the architectural problems are related to agglomerations, concluding that agglomerations are better than isolated smells in indicating the presence of such problems. With a similar conclusion, Palomba et al. [46] found that when a class has more than one smell, it is more prone to faults and changes.

Oizumi et al. [39] studied how code anomaly relationships help developers to identify design problems. For this purpose, they used tests of statistical significance to verify the strength of the relationship. The authors considered as agglomerations, smells affecting two or more elements. The authors found that inter-component agglomerations

are more helpful for locating design problems than intra-component agglomerations. The authors also found that inter-component agglomerations are more helpful for locating design problems than intra-component agglomerations. They conclude that most of the systems analysed presented in their first version a high correlation between agglomerations and design problems. Later, Vidal et al. [59] proposed criteria to rank the agglomerations considering their type and their modifiability. These criteria were evaluated considering how these agglomerations help developers to identify architectural problems. In this dissertation, we focus on understanding if the intra-component agglomerations have a different behavior in terms of modularity metrics.

Lozano et al. [35] focused on understanding the lifecycle of bad smell agglomerations, *i.e.*, when these bad smells co-exists and co-disappears. They also raised evidences that agglomerations degrade the system's quality. Palomba et al. [49] investigated in a dataset composed of 30 systems the bad smell co-occurrence. The authors used association rules to detect these agglomerations. Later, Palomba et al. [45] investigated several versions of Eclipse and Apache to understand how these agglomerations behave in the source code. The authors found that most of the smelly classes contain agglomerations, and that these agglomerations tend to disappear together when it is removed from the code. In this dissertation, we also present an analysis of density, however, we focus on understanding how these agglomerations behaves in terms of modularity metrics. Our results complement the ones found by these three works.

Recently, Walter et al. [61] evaluated bad smell agglomerations on the Qualita Corpus [56], raising evidences that the system domain affects the results. The authors used nine tools to detect fourteen smells. They separated their dataset in order to address the tools bias. However, their separation is not consistent for all the smells, due to the quantity of tools that detect them. The authors found for each domain the bad smell agglomerations that are more frequent in the source code. We extend this work in Chapters 5, 6 by presenting an analysis of how the agglomerations impact modularity metrics and their density in the code. We also use a voting system to mitigate the tools bias, using strictly three tools to each smell analyzed. Furthermore, we address a novel type of agglomeration, the homogeneous one.

Yamashita and Moonen [66] investigated, in a industrial setting, the impact of bad smell agglomerations on the source code maintainability. For this purpose, they used Principal Component Analysis (PCA) in order to identify which bad smells affects maintainability aspects. The authors found that certain types of agglomerations do impact the maintenance activity. They also found which bad smell is present in each of the five factors found. The five factors were: hoarders, confounders, wide interfaces, data containers and unknown. Even though we do not use PCA in our analysis (we

used association rules, a technique in which its results is more easily understood), we complement this work by providing evidences that agglomerations do impact the modularity metrics on open-source systems. Consequently, they affect the maintainability, due to our choice of metrics.

In this dissertation, we extend the previous work [29; 35; 39; 38; 49; 61; 67]. Differently from other authors, we focus not only on identifying the agglomerations, but also on their modularity impact. In other words, we aim at understanding how agglomerations are concentrated in the systems and how they can impact on different aspects of modularity. We also further classified the agglomerations in order to consider their different types, *i.e.*, Heterogeneous and Homogeneous Agglomerations.

2.5 Concluding Remarks

This section explained the main concepts addressed in this dissertation, and discussed several works that address bad smell agglomerations, comparing them to ours. We conceptualize the selected bad smells and the classification used to differentiate the categories related to the presence of bad smells. Finally, we present the main concepts used in the association rule algorithm, used to identify strong heterogeneous agglomerations, and provide a simple, but didactic example of how the selected algorithm, the Apriori, works. The next chapter presents the decisions taken in order to construct our dataset. It also present the voting method to automatically construct a bad smell dataset. Finally, it present how the dataset was evaluated.

Chapter 3

A Dataset of Bad Smells

In the previous chapter we defined the concepts used along this dissertation. In this chapter, we describe how we created a curated dataset of bad smell. Defining which dataset to work with is an essential step in most works, since the conclusions are drawn from the information that it contains. This is true to our dissertation as well, in which we try to raise evidences of how bad smell agglomerations affect the source code modularity. Consequently, the obtained results are directly associated with the system, and our goal is to provide a more generalizable analysis. Therefore, carefully selecting which systems to evaluate, and constructing a reliable dataset is an important step in this work.

This chapter is organized as follows. Section 3.1 describes how the systems were selected, aiming at covering a heterogeneous group of system characteristics, such as size and domain. Section 3.2 presents how our bad smell dataset was constructed. Section 3.3 presents the methodology used to evaluate our data, and the results of this analysis. This step is necessary, since we opted to create a new dataset. We need to assess how much the strategy employed reflect the perspective of developers. Finally, we present in Section 3.4 the threats to the validity of our dataset construction.

3.1 Systems Under Evaluation

Several bad smell datasets are available in the literature [48; 49; 61]. Even though they are available, we opted to create a new one due to several factors. First, we want to assure that the detection strategy was homogeneous for every smell, and easily extendable to address new systems and smells. Second, we want a reasonable number of systems with different characteristics, such as size, domain, and different levels of

maturity. Creating a dataset with systems with different characteristics is essential to achieve a more generalizable result.

In order to create our dataset, we selected a subset of 20 open source systems written in Java from the Qualita Corpus dataset [56], a curated collection of systems widely used in the literature [11; 61]. These systems are widely used in several other industrial and open source projects, supporting different development stages. For example, HtmlUnit and JMeter supports testing. Meanwhile, JHotDraw can be used to model systems. We also selected systems that supports the domain of other systems, such as Spring and Common-Logging. Finally, they vary in terms of size and domain, allowing a more generalized analysis.

Table 3.1 presents the systems that we have selected. The first column presents the name of the system, and the second one presents the system version. The third, fourth, and fifth columns present the total number of lines of code, the number of classes and the number of methods of each system. The sixth column shows the domain they fit, according to the classification provided by the work of Tempero et al. [56]. Finally, the last column presents the number of bad smells found in each system through our detection strategy. As can be seen in Table 3.1, the systems vary from small, such as Commons-Logging that has only 5.5KLOC distributed in 73 classes and 464 methods, to large systems, such as Hibernate, with 431KLOC distributed in 6,018 classes and 41.5K methods. We have found that the number of bad smells found in each system varies greatly, in a range of 3, for the Commons-Logging, and 2,984 for Weka.

3.2 Dataset Construction

This section describes how the dataset of bad smell was created. Building a large dataset of bad smells manually is an expensive, tedious and subjective activity since all classes and methods should be manually evaluated and documented. Besides, different evaluators have different criteria to identify bad smells. Developers perspective about what can be considered as containing the bad smell under consideration may be affected by different aspects, such as system knowledge and developer experience [36]. For these reasons, we relied on automatic detection tools that are mostly used in the literature and are currently available for use [11; 15; 17; 40].

For instance, Fontana et al. [18] evaluated and compared three out of five tools that we have selected. Fernandes et al. [15], in their literature review and case study, raised detection tools that exist in the literature, and later evaluated if the tools agree in their results. Paiva et al. [44] found in their work that the detection tools have a

Table 3.1: Selected Systems

Name	**Version**	**TLOC**	**#Class**	**NOM**	**Domain**	**#BS Found**
CheckStyle	5.6	23,416	292	1,800	IDE	618
Commons-Codec	-	8,346	71	456	tool	49
Commons-IO	-	30,371	276	2,274	tool	11
Commons-Lang	-	27,852	195	1,580	tool	13
Commons-Logging	-	5,449	73	464	tool	3
Hadoop	1.1.2	184,251	1,315	12,099	middleware	188
Hibernate	4.2.0	431,475	6,018	41,529	database	207
HtmlUnit	2.8	100,759	853	8,014	testing	966
JasperReports	3.7.4	193,408	1,527	14,997	visualization	1,558
JFreeChart	1.0.13	143,062	934	10,442	tool	1,647
JHotDraw	7.5.1	79,668	671	5,892	graphic	1,063
JMeter	2.5.1	94,763	940	7,989	testing	1,254
Lucene	4.2.0	412,376	4,136	22,615	tool	729
Quartz	1.8.3	28,557	232	2,343	middleware	292
Spring	3.0.5	311,027	3,541	29,483	middleware	191
SquirrelSQL	3.1.2	6,944	56	532	database	5
Struts	2.2.1	143,196	1,958	13,244	middleware	1,668
Tapestry	5.1.0.5	97,206	1,553	7,809	middleware	715
Tomcat	7.0.2	178,133	1,287	14,260	middleware	1531
Weka	3.6.9	272,611	1,535	17,851	tool	2,984

higher agreement on the true negatives. These findings corroborate with the reasoning that by combining different tools that have a lower agreement on true positives, we are obtaining a more reliable ground truth. This is due to the combination of the results, in which at least two tools indicate that the bad smell is present.

Furthermore, due to the definition of bad smells being informal, different tools uses different detection strategies in order to detect them. For example, JDeodorant [17] tool uses slicing techniques to identify and suggest which piece of code should be refactored. In contrast, JSpIRIT [60] uses combinations of software metrics in order to identify the smells. It is worth to mention that in order to use combinations of software metrics, it is needed to select, for each metric, thresholds, *i.e*, values that separate the data in "normal values" and "outliers". As an example, we may cite the Lines of Code (LOC) metric. Tool T_1 may consider classes with more than 300 LOC as an outlier, when tool T_2 considers classes with more than 500 LOC as an outlier. In the literature, recent works propose machine learning techniques to identify bad smells. The main advantages of their use is that it can receive feedback and incorporate them into the

model [14], and the developer do not need to specify metric thresholds. However, to train the algorithms, a reliable ground truth of the smells is needed.

In order to obtain a homogeneous and extendable methodology for creating a reliable dataset, we opted to use a voting method. With this voting system, we expect to gain from the diversity of detection strategies, ranging different perspectives about what can be considered as a bad smell. In the voting method, each instance (a class or a method, depending on the bad smell granularity) received three votes from three different detection tools. Each vote represents if the instance contains or not the bad smell according to a tool, being summarized in a binary variable. If for a bad smell *S*, the instance received two or more positive votes, then the instance is added to our dataset of bad smells. If not, the instance is not considered as containing the bad smell *S*. This process is repeated for every smell evaluated. We choose the voting method due to empirical evidences that having more "thinking units" provide a more accurate prediction [24].

Table 3.2 presents which detection tools were used to detect each bad smell. The lines represent the tools, and the column the bad smell evaluated. A *X* at a cell in line *L* and column *C* indicates that the tool in *L* was used to detect the smell in *C*. In total, we selected five detection tools to create our dataset. For homogeneity of results, we strictly used three tools per bad smell. However, some of the tools detects more bad smells than those presented in the table, or detect smells that are not marked with a *X* in the table. In these cases, we opted to use the tools that were mostly used to detect such smell. For instance, although able to detect several bad smells, including the ones evaluated in this dissertation, PMD[1] was only used to detect Large Class.

Table 3.2: Detection tools used

Tool	Large Class	Refused Bequest	Feature Envy	Long Method
JDeodorant [16]	X		X	X
PMD[1]	X			
JsPIRIT [60]	X	X		
DECOR [37]		X	X	X
Organic [40]		X	X	X

After applying the voting method, Table 3.3 presents in each line the quantity of smells recorded in our ground truth. The first column shows the bad smell name. The second column shows the absolute number of bad smells found by the voting method described above. The third column presents the respective proportion of the bad smell on our ground truth. The last column presents the proportion of classes or

[1]https://pmd.github.io/pmd-6.23.0/

methods affected by the smells in our dataset. The denominator in the proportion is dependent on the granularity of the smell. For example, for the Large Class smell, the denominator is the total number of classes in all systems, and for the Long Method smell, the denominator is the total number of methods in all systems.

Table 3.3: Number of bad smell found

Bad Smell	**#Smell**	**#Smell/#TotalSmell**	**% of Smelly Elements**
Large Class	1,743	11.11%	6.34%
Refused Bequest	3,088	19.68%	11.2%
Feature Envy	8,791	56.03%	4.1%
Long Method	2,068	13.18%	0.9%
Total	15,690	100%	-

We can observe from Table 3.3 that about 56% of the bad smells found in the systems are Feature Envy. We can verify that about 30% of the smells are at class level and 70% are at method level. This proportion is expected, since a system has more methods than classes (see Table 3.1). It is interesting to note that even though classes that do not participate in an inheritance tree are more common in the system, the quantity of Refused Bequest found indicates that when inheritance is used, they often present a problem. Refused Bequest represents almost 20% of the total smells found, and 11.2% of the classes in our dataset is affected by this smell. Besides, as shown in the last column of Table 3.3, the total number of smelly instances is often smaller than 10% for most smells, except for Refused Bequest. Table 3.3 provide us motivation for an in depth study of the imbalance of our ground truth in the obtained results.

3.3 Dataset Evaluation

Since we created a new dataset, we opted to manually evaluate a sample of bad smells to verify that the results obtained by the voting method are aligned with the human perception. Even though human validation is subjective [65], and the manual inspection may apply criteria that are difficult to compute automatically, they influence greatly on the decision of what can be considered a bad smell.

The dataset manual evaluation was conducted in the following way. First, it was defined three questions that each of the six evaluators answered for each smell under evaluation (a total of 12 questions). These evaluators are post-graduate students and researchers of software engineering with experience in detecting bad smell. The defined questions help to identify each bad smell and help to guide the evaluators in their task.

Furthermore, they make the validation process homogeneous [51], since each evaluator answers the same set of questions. Besides, this methodology provides us with the necessary flexibility to accommodate different perspectives of the evaluators, since each one of them can interpret the question as desired. For example, some questions are related to the size of the system. The interpretation of what is considered *small* or *large* is subjective to the evaluator.

Table 3.4 presents in the first column the bad smell under evaluation, the second columns presents the three questions that each evaluator should answer in order to characterize the class/method as containing the bad smell. Finally, the third column presents the answer that is expected to consider the instance as containing the bad smell. These questions were based on two works proposed in the literature. For the Large Class smell, the questions were based on the work of Schumacher [51]. For the rest of the smells, the questions were based on the work of Lanza and Marinescu [29]. To consider the class/method as containing the smell S, at least two answers from the evaluator have to match the expected output (last column from Table 3.4). Simply put, for a smell S and instance I, the instance I is considered as containing S if:

((Evaluator Answer 1 = Expected Answer 1) AND (Evaluator Answer 2 = Expected Answer 2)) ***OR*** *((Evaluator Answer 1 = Expected Answer 1) AND (Evaluator Answer 3 = Expected Answer 3))* ***OR*** *((Evaluator Answer 2 = Expected Answer 2) AND (Evaluator Answer 3 = Expected Answer 3))* ***OR*** *((Evaluator Answer 1 = Expected Answer 1) AND (Evaluator Answer 2 = Expected Answer 2) AND (Evaluator Answer 3 = Expected Answer 3))*

For example, for the Long Method smell, if the evaluator answered two of the three questions as "Yes", the instance is considered as containing the Long Method smell. It is worth to notice that for Question 3 of the Refused Bequest smell, if the evaluator answered "No", this is counted as positive evidence that the class presents the smell.

It is worth to mention that each bad smell instance was checked by only one evaluator. Our focus is on the agreement between our methodology and the evaluators opinion, not in comparing the opinion of different evaluators. Since we have two smells at class level and two at method level, each evaluator answered six questions for each bad smell instance. This strategy aims at avoiding them to know beforehand which type of bad smell instance they are dealing with. Besides, for each question, the evaluator could have answered with three options: Yes, No and IDK (I don't know). When the evaluator could not respond the question, they were encouraged to answer as *IDK*,

since the *No* option affect directly the results of all questions. If an instance received two or more *IDK* answers for smell *S*, the instance was removed from the evaluation, since it did not provide us a positive or negative result. In total, we removed 9 instances because of the *IDK* values: 4 for Large Class, 1 for Long Method, 2 for Refused Bequest and 2 for Feature Envy.

Table 3.4: Bad Smell Evaluation Questions

Bad Smell	**Questions**	**Expected Answer**
Large Class	Q1. Does the class have more than one responsibility?	Yes
	Q2. Does the class have functionality that would fit better into other classes?	Yes
	Q3. Would splitting up the class improve the overall design?	Yes
Refused Bequest	Q1. Does the class use only a little of parent's behaviour?	Yes
	Q2. Does the parent class provide more than a few protected members?	Yes
	Q3. Does the class is too small/simple?	No
Feature Envy	Q1. Does the method use directly more than a few attributes of other classes?	Yes
	Q2. Does the method use far more attributes from other classes than its own?	Yes
	Q3. Do the used "foreign" attributes belong to very few other classes?	Yes
Long Method	Q1. Does the method have many conditional branches?	Yes
	Q2. Does the method is excessively large?	Yes
	Q3. Does the method use many variables?	Yes

In order to evaluate our dataset with statistical significance, for each smell under evaluation, we calculated a sample size that achieves a 90% confidence and maximum error of 10%. The formula that calculates the sample size was based on the Central Limit Theorem [25]. We used the four files that contains the list of instances affected by each smell, randomly ordered them 10 times with 10 different seeds, and sampled it according with the sample size calculated in the first step. With the samples in hands, we sorted them by system and separated them into different files. Each system was allocated to one evaluator, allowing them to focus on few systems at a time. Finally, the agreement between the evaluators and tools was calculated using the Fleiss Kappa [16]. Fleiss Kappa calculates the probability that the observed agreement between the raters exceeds the expected agreement, if all the raters made their rating in a random way. That is, it evaluates how reliable the ratings are. We opted for Fleiss because it is

a generalization of the Cohen Kappa [9]. This generalization is due to the consideration that an evaluation could be made by more than two evaluators, assumption that does not hold if we had used Cohen Kappa.

The results can be interpreted in the following manner. Poor: < 0.001. Slight: 0.00 - 0.20. Fair: 0.21 - 0.40. Moderate: 0.41 - 0.60. Substantial: 0.61 - 0.80. Almost Perfect: 0.81 - 1.00 [28]. With a confidence level of 95%, we obtained the following agreements: 0.467 for Large Class (Moderate), 0.651 for the Refused Bequest (Substantial), 0.353 for the Feature Envy (Fair), and 0.406 for the Long Method (Moderate). Even though there is room for improvement, the overall agreement was 0.467 (Moderate). A satisfying agreement due to evidences that tools mostly agree on the true negatives [18; 44], and we evaluate the candidates of true positives. Besides, tools are built compromising between recall and precision.

3.4 Threats to the Dataset Construction

It is worth to notice that the results obtained in the dissertation depend directly on how the dataset was constructed. To achieve a more generalized result, we opted to create a new dataset that contains systems from different sizes and domains. Also, we used automatic detection tools to detect the bad smells. However, the tools try to achieve a balance between recall and precision. In this work, this threat was mitigated in three ways. First, we selected tools widely used in the literature [11; 15] and their detection strategy varies. Second, we used a voting method, in which the bias created by the detection strategy employed by the tools is mitigated due to the vote system. Finally, we manually evaluated a statistical sample of our dataset and found a good agreement between the manual strategies and the automatic detection [44]. We think these steps are sufficient to address most of the bias introduced on the construction of the bad smell dataset. However, since we choose a sample of existent open-source systems, the results may not hold for all systems.

To construct and evaluate our dataset, we heavily rely on scripts to process the tools output and to calculate the agreement. For this purpose, we revised several times these scripts, in order to assure that all processed data are correct. These scripts were written in Python, and were revised by two people. For the agreement, we used the package *irr* from the R language.

3.5 Concluding Remarks

This chapter aimed at explaining how our dataset of bad smell was created. We presented the criteria to select the systems from the Qualita Corpus [56] in Section 3.1. In total, we selected 20 Java systems with different sizes and domains. Later, we presented the methodology used to build our dataset. Finally, we described and presented the results of our dataset evaluation. We found that, in general, the agreement between the voting system and the evaluators is moderate.

In the next chapter, we present the methodology used in this dissertation. It defines the research questions that guides the selection of which technique to use in order to answer them. The next chapter also presents how and what data to collect from our data. Besides, we also explain how the dataset was processed in order to serve as input for the association rule algorithm, the parameterization of the association rule algorithm, the methodology used to compute the effect of agglomeration on the modularity metrics, and how the agglomerations density was calculated. Finally, we present the threats to the validity of this dissertation in terms of methodology.

Chapter 4

Study Design

In the previous chapter, we focused on discussing how our dataset of bad smells was created. First, we described the systems to compose our dataset, and why they were selected. We then explained how we obtained our bad smell dataset, through the use of a voting system. Finally, the process of evaluating the bad smell dataset was described. This chapter aims at explaining how the empirical evaluation of this dissertation was conducted using this dataset. Figure 4.1 presents an overview of how the data was obtained and analyzed. The single line rounded box represents the steps we followed. Letters inside circles, next to the rounded boxes, represent the order that the steps were executed. Arrows indicate that a step was served as input for the next one.

Twenty Java systems were selected from the Qualita Corpus [56]. They serve as input for the calculation of the selected modularity metrics (Step D), described later in Section 4.3. The systems also serve as input to the construction of the bad smell dataset (Step A). In this step the voting system was applied, and the dataset is obtained. The next step was to evaluate the bad smell dataset (Step B). Steps A and B were presented in Chapter 3. With the evaluated data, we can begin to identify bad smell agglomerations in our dataset (Step C). After obtaining the list of agglomerations in Step C and their metrics in Step D, we can calculate how they affect aspects of the system modularity, evaluating their difference in terms of mean and variation (Step E). As output, we identified the bad smell agglomerations present in the systems; evaluated their density on the source code; and how they affect the system modularity. We also analyzed in depth the behavior in terms of density and impact of the Heterogeneous Agglomerations on software modularity.

Before explaining this chapter organization, it is worth to remember some concepts. *Heterogeneous Agglomeration* is a class that contains two or more bad smell types, and two or more bad smell instances. A *Homogeneous Agglomeration* is a class

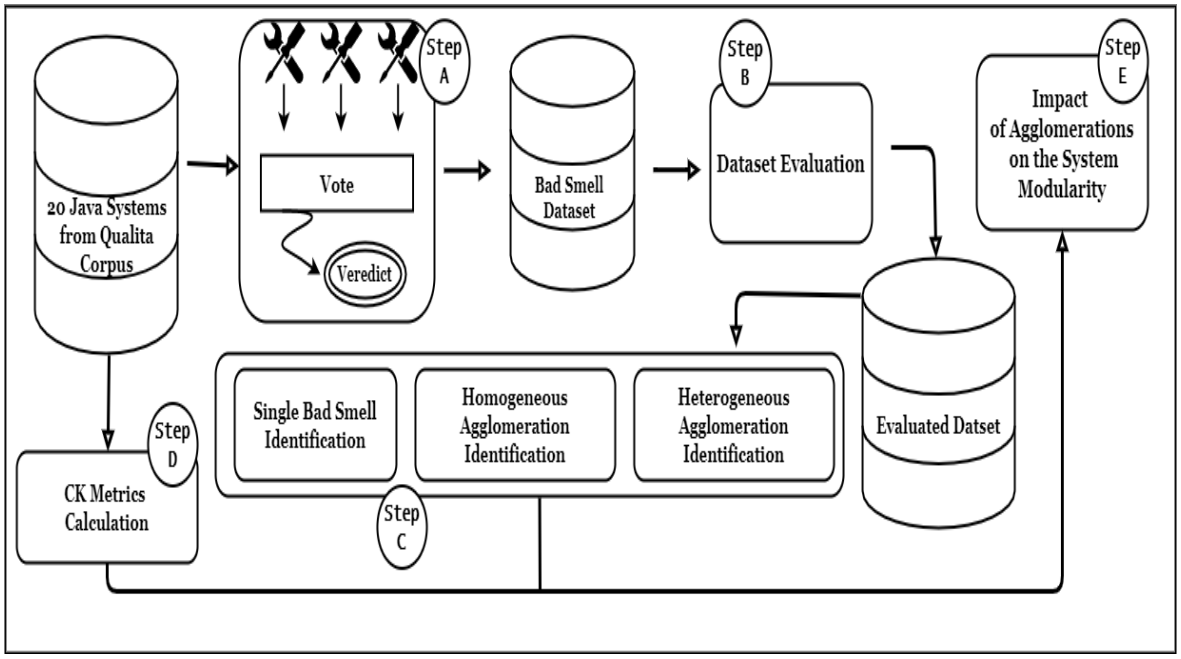

Figure 4.1: Methodology steps

that contain at least two smell instances, but all of them are of the same type. *Isolated classes* only contains one bad smell instance. Finally, *Clean classes* does not present any smell instance. This chapter is organized as follows. Section 4.1 presents the Research Question (RQ) that drives this work, with a brief explanation of the steps conducted to answer them. Section 4.2 explains how the Homogeneous and Heterogeneous Agglomerations were identified. Section 4.3 describes the use of metrics as a proxy of the source code modularity. Sections 4.4 explains how the density was calculated. Finally, Section 4.5 presents the main threats to the validity of our dissertation.

4.1 Research Questions

Our goal is to understand how bad smell agglomerations impact the source code modularity. For this purpose, we first have to identify such agglomerations, and then measure them according with a set of metrics. Finally, we can apply techniques that allow us to comprehend how the classes with the identified agglomerations compare with classes that have only a single smell instance or that are Clean. This goal is achieved with meaningfulness metrics for the association rules, frequency statistics to characterize the systems, measurements of effect, and analysis of variability of the data. This scope helped us to formulate the main research questions that guide the data collection and their analysis. We now present the RQs and a brief explanation of their purpose.

RQ1 Are there Heterogeneous Agglomerations that are more meaningful in the source code than other heterogeneous ones?

This question aims at understanding which bad smell agglomerations have a significant presence in the source code. For this purpose, we use association rules and metrics of meaningfulness to mine only the statistically significant agglomerations. Even though the technique appears to be too sophisticated to answer this question, it provides us a complete set of information about the dependencies between the bad smells, the strength of the rule, and allow us to extend our analysis for more bad smells and systems in the future. More details are present in Section 4.2.

RQ2 Are Homogeneous Agglomerations more frequent in the source code when compared to Heterogeneous Agglomerations?

RQ2 complements RQ1, since in the association rule, a rule cannot contain the same item in the consequent and antecedent parts, *i. e.*, a Homogeneous Agglomeration is not identified using the association rule algorithm. To answer this question, Homogeneous Agglomerations are found through the calculation of frequency statistics. Later, they are classified as Homogeneous Feature Envy or Homogeneous Long Method, according to the type of smell present in the agglomeration. This allow us to verify if they present a similar behavior in terms of appearance.

RQ3 Are bad smell agglomerations denser in the source code than isolated smells?

This question aims at understanding if the size of the system is influencing the presence of agglomerations. Upon observing the density, we can observe that if the agglomeration categories (described in Section 2.2) are mitigated when considering different levels of granularity. To answer this RQ, we classified each class according to their agglomeration category. We then group each class considering the system that the class belongs. Later, frequency statistics were used to understand how the categories density compares to each other at the level of system. We present three sets of boxplots that consider: (i) the absolute number of instances for each category, (ii) the density of affected classes by each category, and (iii) the density of each category per KLOC.

RQ4 How the bad smell agglomerations impact on aspects of the system modularity?

This RQ aims at raising evidences of how bad smell agglomerations impact on the source code modularity. We use source code metrics as a proxy of the system modularity. Each metric represents aspects of modularity. If there is a significant difference in the metric values, they may be used to provide evidence of the presence of agglomerations. This information could help practitioners to know if the existence of agglomeration on the code that they are working on can lead to technical debts. These debts should be refactored in order to pay them, before they are too difficult and costly

to do so. To answer this question, we present a heat map that helps to visualize the effect of these agglomerations on the aspects of modularity considered.

Finally, we present a complementary research question that aims at understanding in depth if there is a difference between the different types of Heterogeneous Agglomerations that appeared on the bad smell dataset. Beyond analyzing their differences, we evaluate if the different types of Heterogeneous Agglomeration reflect the aggregated heterogeneous type studied in RQ1, RQ3 and RQ4.

RQ5 Does the types of Heterogeneous Agglomerations have an uniform behavior in terms of density and impact on aspects of system modularity?

This RQ aims at understanding if there is a difference in the behavior of Heterogeneous Agglomerations in their density along the systems and how they impact on modularity. This comparison is made among only Heterogeneous Agglomerations, allowing us to verify if their behavior is uniform. For this purpose, we employ a similar strategy to the RQ2 and RQ4. However, we classified each Heterogeneous Agglomeration according with the bad smells that it contains.

One aspect that may be highlighted is the fact that we are evaluating bad smell agglomerations that may overlap with each other. For example, in RQ3 and RQ4, we compare Homogeneous FE and Isolated FE, and Homogeneous LM and Isolated LM. We opt to make this analysis to test our assumption that classes with more than one smell instance have a more degraded modularity than classes with only one smell instance. In allowing this comparison, we can verify if for these smells the assumption holds. For the in depth analysis of Heterogeneous Agglomerations, the overlap of bad smells in the agglomeration permits us to evaluate if the obtained results are being influenced by the smells that are present in class C_1 but not in C_2. We are interested in evaluating how the difference between sets of smells behaves in terms of density and impact on the aspects of modularity.

4.2 Identification of Agglomerations

In order to identify the Heterogeneous Agglomerations, we used the Apriori Algorithm. We first generated one file for each bad smell with all the classes that contained them. In order to generate the input for the association rule algorithm, we wrote a Python script that matches the instances that contains the different bad smells. Using dictionaries and reading each file one time, we obtained, for each class that contains at least one bad smell instance, a tuple composed of: 4 binary variables indicating the presence of Large Class, Refused Bequest, Feature Envy and Long Method, respectively; one

integer variable counting the number of Feature Envy in the class; and one integer variable storing the number of Long Method present in the class. The four binary variables are used in the Apriori Algorithm, and the two integers allow us to verify the density of method smells in a class.

However, Apriori requires as parameter metrics to use to make the decision of what rule to consider as strong. The most common metric to use in association rules is Support. Support alone leads to poor results [24], since it calculates the probability of co-occurrence of items in a dataset. To address this limitation, we consider as our decision maker in each step the metrics Lift and Confidence. The association rule algorithm also receives as parameter an integer that limit the number of items in the Antecedent part of the rule. We opted to not limit the quantity of items in the Antecedent, because this provide us additional information about the strength of the rule.

To illustrate this gain, is more didactic to provide an example. Let's suppose you have items A, B, C and D. After applying the association rules without constraining the Antecedent, you obtained the following rule: $A, B \rightarrow C$. When observing how the Apriori algorithm works, in the first iteration, the algorithm discovered that items A, B and C have a significant presence. Meanwhile, item D is rare and can be unconsidered. In the next step, the algorithm discovered that $A \rightarrow C$ and $B \rightarrow C$ are meaningful. Finally, in the final step we would obtain the rule $A, B \rightarrow C$. If we had limited the algorithm to contain only one item in the Antecedent, we would have obtained the two rules $A \rightarrow C$ and $B \rightarrow C$. However, they do not provide us the information that when A and B appears together with C, they are as strong as the two separated rules. Consequently, in constraining the number of items in the Antecedent, we are ignoring the interaction between the items A and B that in our example is really significant.

After presenting the reasons for the parametrization, Table 4.1 presents the three meaningfulness metrics used, their respective formula and threshold. The Confidence metric indicates the validity of the rule, ranging from [0,1]. This metric can be interpreted as the chance that a class presents the bad smell S, given that (a set of) bad smell Y is present in the class. The Lift measures the dependence between the two sides of the rules, ranging from [0,∞], with 1 indicating that the items are independent. A Lift above 1 indicates that both sides of the rule are positively dependent.

It is worth noticing that the considered thresholds were adapted from previous works [49; 61]. Since Walter [61] and Palomba's [49] thresholds are not flexible enough, being a fixed value, we have empirically tested new thresholds to verify if they could lead to other association rules. However, we could not find new rules when trying different combinations of confidences and lifts. To obtain the association rules and all

Table 4.1: Association Rules Measurements

Rule	**Concept**	**Formula**	**Threshold**
Support	% of samples that the rule satisfies in the dataset [24]	$P(A \cup B)$	above 0.05
Confidence	Degree of certainty of the detected rule [24]	$\frac{P(A \cup B)}{P(A)}$	above 0.6
Lift	The dependence between items in the rule [24]	$\frac{P(A \cap B)}{P(A) \times P(B)}$	above 0.80

statistical metrics that we investigate in this dissertation, we have used the package arules[1] from the R language, a consolidated implementation that was used in different works [49; 61; 67].

To complement the analysis of rule strength and items correlation, we also measured: (i) the Chi Square of the rule, which tests the independence between the Antecedent and Consequent, in which high values means that they are not independent. (ii) Imbalance Ratio, that measures the degree of the imbalance between the two sides of the rule, in which values close to 1 indicate that the conditional probabilities are very different. (iii) Odds Ratio (OR), that is used to compare the probability of finding A in transactions that contain B in comparison with transactions that contains A but do not contain B. Values of OR above 1 means that the probability of A occurring is higher when B occurs [55].

To identify Homogeneous Agglomerations, we have used frequency statistics. Even though they are less powerful than the meaningful metrics, this choice was made due to the fact that association rules have the restriction that items in the Antecedent have to be different from those on the Consequent. The agglomerations found were further classified as Homogeneous Feature Envy or Homogeneous Long Method.

4.3 Modularity Metrics

Attributes of quality are difficult to estimate directly from the source code, since they depend on the context of the system, on who is evaluating, and mainly, they express nonfunctional requirements. To address this complexity, we used source code metrics that help to indicate the system modularity, since it conceptualizes in a numerical value aspects of the system. For example, they can extract information about class/method cohesion, size, coupling, complexity and inheritance. Several object oriented metrics

[1]https://cran.r-project.org/web/packages/arules/index.html

that capture modularity have been proposed in the literature [8; 34]. Here, we focus on the ones that had their capability of representing aspects of modularity.

In order to make this analysis, each class of the systems has to be measured according to a set of metrics. Since it is known - through the bad smell dataset - which classes have agglomerations, they can be compared in terms of difference in their means and variations. These statistics allow us to understand if there is a difference in their general behavior. We made our analysis inspired by previous work [20] that studied the impact that refactorings have on source code metrics and found that some refactoring operations do impact on the metrics values.

First, we calculated the metrics for all classes of the 20 systems. Later, we classified each class in the metrics files through the use of a Python script. These files were merged in one, being grouped later in several files, according with its agglomeration type. These files contain the class name, the metric values and its classification. This grouping allows the identification of statistical differences between metric values for each category, since we can obtain frequency statistics for each category. For this analysis, we used Cohen's d [9], a measure of effect size. Cohen's d in the *t-test* assumes that the standard deviation of both samples under evaluation are similar. Since this assumption does not hold in our dataset, we used a pooled standard deviation that considers both samples means and sizes. Beyond this calculation, we also analyse the direction in which the d value is tending.

To complement our results, we also present for each pair metric-category the mean, standard deviation and the Coefficient of Variation (CV). The Coefficient of Variation [52] is a normalization that considers the standard deviation and the mean. It can be interpreted as how much the standard deviation deviates from the mean, providing a more consistent comparison between means that have different scales and that varies greatly in values. Through the ratio between two CVs, we can calculate the difference between the normalized variations of two categories.

Finally, we also make a correlation analysis using Pearson Correlation [52]. This analysis allow us to verify if the selected metrics are strongly correlated. Consequently, it allows us to identify if the obtained results for the Cohen's d and CVs are correlated. This measurement can be interpreted in the following way. Values close to 0 indicates that the two variables are not correlated, indicating that they are independent. Values close to -1 means that there is a negative correlation between the two variables, *i.e.*, when one of the variable increases in value, the other variable decreases its values. Similarly, values close to +1 indicates that the variables are positive correlated. If one variable had a increase in its value, the other one will have its value increased too.

Table 4.2 shows the selected metrics and what it calculates. We focused on

Table 4.2: Analysed Metrics and their Definition

Name	Definition
Coupling Between Objects (CBO)	It calculates how many classes are coupled to a class C.
Depth of Inheritance Tree (DIT)	It measures how deep the class is in the inheritance tree.
Response set of a class of objects (RFC)	It measures how many methods can be called given a call in an object of a class.
Weighted Method Class (WMC)	It is calculated summing all the weights for each method of a class. In our case, the weight is the number of lines of code of each method.
max nest blocks (maxNest)	It calculates the highest number of blocks nested together in a class C.

metrics that are associated with class, to avoid having to group the metric values to raise them to the class level. This grouping could lead to incorrect results, since the most common of them is to take means, not taking into account the class variation. Table 4.3 presents the metrics and the aspect of modularity that they measure. The last column shows the study that empirically evaluated their importance to the modularity aspect. The metrics were selected due to the existence of studies that provide evidences that they may quantify an attribute of modularity, such as complexity, coupling and fault proneness [3; 4]. We use these aspects as symptoms of degradation of the system modularity. Most of the metrics are from the CK metrics suite [8], a widely used set of metrics that capture aspects of modularity. We used the CK tool[2] to calculate the selected metrics.

Table 4.3: Metrics and its Modularity Aspects

Metric	Aspect of Modularity	Source
CBO	Coupling	[3]
WMC	Complexity/Fault proneness	[3; 4]
MaxNest	Complexity	[3]
DIT	Inherintance/Fault proneness	[3; 4]
RFC	Fault proneness	[4]

These metrics approach different views of modularity. High coupling means that the entities in the source code have a strong dependency between each other. This makes the code more difficult to change, since developers need to pay caution to all dependencies before changing the class. If not, the software system may break. High

[2]https://github.com/mauricioaniche/ck

complexity makes the source code harder to understand and to maintain. The metrics concerning the fault proneness means that the entity is more prone to presenting faults than other entities. This proneness may imply different modularity problems, such as high complexity, complex relationships between components, and others. Finally, the inheritance aspect means that, as the name may suggest, it may have a problem in the inheritance tree that the class belongs.

4.4 Density Calculation

In order to verify if size was impacting the existence of the intra-component agglomerations, we verified how such agglomerations are distributed along the systems. We focused our analysis in three measurements at different granularity levels: absolute number of agglomerations, the density of classes affected by the agglomerations, and their density across the lines of code (LOC). First, we calculated for each system their number of classes, their number of methods and their LOC. We then generated two different files: one that presents the agglomerations described in 2.2, allowing us to verify if different size measurements are affecting directly the density of agglomerations. The second file presents only the classes that are Heterogeneous Agglomerations, classified according with the bad smells present in the class. For example, RbLm is a class that contains only Refused Bequest and Long Method. This classification allows us to verify if there are Heterogeneous Agglomerations that are more dense in the source code, and if different measurements of size mitigate their presence in the code.

The next step was to calculate the ratio of agglomerations per class, per method, and per KLOC. For this purpose, we calculated for each agglomeration type, the number of agglomeration in system Sy divided by: the number of classes of the system Sy; divided by the number of methods of system Sy; and divided by the KLOC of system Sy, respectively. In order to complement this analysis, we also present the Inter Quartile Rate (IQR) measurement and the standard deviation, both calculated around the system mean values. The IQR is a measure of dispersion that captures the difference between the third and first quartiles. High values indicate a large variability in the results, *i.e.*, other system characteristics play a major role in the presence of bad smell agglomerations. Conversely, a low value indicates that the overall results are stable. This analysis was inspired on the Palomba et al. work [46].

4.5 Threats to Validity

Since we are dealing with a large volume of data from the dataset and statistical analysis, we opted to use consolidated algorithms from the R and Python languages, such as the *arules* from *R* and *pandas* from *Python*. To present the data, we also used known algorithms from the *d3* library from *JavaScript*. To avoid errors in transcription, all generated data were obtained through the use of scripts. The scripts and their results were revised several times to assure their correctness.

4.6 Concluding Remarks

In this chapter, we aimed at presenting this dissertation methodology. For this purpose, we first presented a figure summarizing the activities that took place in order to achieve our goals. Later we presented our five research questions, followed by a brief explanation of their respective importance and the analysis technique used to answer them. We then presented the parametrization of the Apriori Algorithm, and the motivation for choosing them. The next sub-chapter described the selected modularity metrics and the methodology used to evaluate the impact that the categories have on them. Finally, we presented how, and why, the agglomeration density was calculated.

In the next chapter, we will focus on answering the first four research questions that aims at verifying how the agglomerations described in Section 2.2 behave in the source code. For this purpose, we first identify which Heterogeneous and Homogeneous Agglomeration is frequent in the source code. We then evaluate their density in the source code considering different levels of granularity, and how they impact aspects of the source code modularity.

Chapter 5

Results and Analysis of Bad Smell Agglomerations

In the previous chapter, we focused in describing the methodology used along this dissertation. We defined the research questions that drives this work, with a brief explanation about how they were answered. We then explained the reasoning of the selected parameters of the Apriori algorithm. Later, we presented how the impact of bad smell agglomerations on software modularity was calculated. Finally, we presented how the density was calculated for each system, aiming at investigating how the size is influencing our results at three different levels of granularity.

This chapter aims at presenting our findings and answering the research questions related to bad smell agglomerations. Each section aims at answering one question as follows. Section 5.1 describes the results of the identification of Heterogeneous Agglomerations. Section 5.2 presents the analysis of the presence of Homogeneous Agglomerations in the source code. Section 5.3 evaluates the agglomeration density in comparison to isolated smells. Section 5.4 analyses the impact of agglomerations on aspects of software modularity. Finally, Section 5.5 concludes and discusses our key findings and implications for practitioners and researchers.

5.1 Understanding Heterogeneous Agglomerations

This section aims at answering the first research question, exploring which Heterogeneous Agglomeration is frequent in the source code, and how the bad smells that compose them are correlated. We answer: *RQ1: Are there Heterogeneous Agglomerations that are more meaningful in the source code than other heterogeneous ones?*. Table 5.1 presents the association rules found and their meaningfulness metric values.

The first column shows the detected rules, and the second to fourth columns present the association rules metrics, the Support, Confidence and Lift, respectively. The fifth column presents the result of the test of independence between the items in the rules, the Chi-Square (χ^2). The sixth column shows how imbalanced the sides of the rules are, the Imb. column. The seventh column shows how many instances of the rule were found, and their respective participation in all agglomerations found, including the Homogeneous Ones. Finally, the eighth column present the Odds Ratio (OR) for each rule found.

Rules can be interpreted in the following way: a rule is a Heterogeneous Agglomeration composed of at least two bad smell types. The smells in the Antecedent part of the rule (left side) are evidences that the smell on the Consequent part (right side) is also present in the agglomeration. That is, a class that contains the smell on the Antecedent have a high probability of also presenting the smell on the Consequent. Interestingly, every rule found by the Apriori has the *FE* smell on the Consequent. This means that every agglomeration of at least two smell types has a high probability of also presenting the *FE* smell. We may conclude that the presence of *FE* in an agglomeration is highly dependent on the presence of other bad smells.

To further investigate the presence of only *FE* on the Consequent part of the rule, we have changed the parameters and thresholds of the algorithm to verify if it yielded different results. However, even after these changes, we did not find rules that have a different bad smell on the Consequent. This result may be due to the large presence of the *FE* smell in the dataset compared to other smells. Our dataset is unbalanced, similar to the real world. This imbalance can be confirmed with the Imbalance Ratio [62] shown in column Imb. in Table 5.1. For all detected rules, the value is close to 1, indicating that the conditional probabilities of both sides of the rules are very different. This imply that the significant presence of Feature Envy smell may be directly affecting our results.

Table 5.1: Agglomerations Found by Association Rule

Rule	**Supp.**	**Conf.**	**Lift**	χ^2	**Imb.**	**Count**	**OR**
LC, RB → FE	0.062	0.756	1.485	142.58	0.807	406(22.72%)	2.131
RB, LM → FE	0.047	0.614	1.205	23.97	0.802	310(17.35%)	1.587
LC, LM → FE	0.043	0.679	1.322	51.15	0.843	283(15.84%)	3.262
LC, LM, RB → FE	0.024	0.734	1.441	44.58	0.920	157(8.79%)	2.736

To complement our analysis, we calculated the dependence between the sides of the rules with the χ^2 [33] with 1 degree of freedom and $\alpha = 0.05$. From the column χ^2 in Table 5.1, we can observe that all Antecedent and Consequent parts are highly dependent, with the rule $LC, RB \longrightarrow FE$ presenting the highest dependence value

($\chi^2 = 142.58$). Finally, we can observe that, for all rules found, the OR is above 1, indicating that the presence of the Antecedent is positively influenced by the presence of the Consequent in the agglomeration. The highest OR value was obtained for the rule $LC, LM \longrightarrow FE$.

We can also analyze if this result is directly associated with the bad smell definitions. Beyond having a large size in terms of LOC, many attributes and many methods, a Large Class has more than one responsibility [21]. By consequence, some of its methods may be interested in being in another class. The same occurs with Long Method and Refused Bequest. In Long Method, some of its functionality may be interested in another method from another class. In Refused Bequest, since it tends to not use their parent behavior, to improve the inheritance tree, all unrelated behavior of the class should be moved to another class. Consequently, Refused Bequest classes may present methods that envy other classes. As can be seen, the bad smell definitions are closely related, some of them even suggesting the presence of each other. Therefore, the association rules found match their definitions.

We also limited to 1 the number of items in the Antecedent, to force the appearance of different rules that did not involve the Feature Envy smell. Even doing this, we could not find other rules that had other smell on the Consequent. The rules found in this dissertation complement the ones found in the work of Walter et al. [61]. The authors also presented in a table, a comparison between the new agglomerations found by them, and those that were already found in the literature. Our results compare to theirs when analysing the dataset composed of the agreement between two and three tools, the quantity of tools that detected the smells in our dataset. All rules found in this dissertation are different from those found in their work and on their comparison table with the literature.

***RQ1**: We have found four meaningful Heterogeneous Agglomerations, with the Feature Envy smell always on the Consequent part of all rules. This result may imply that combination of two or more smells increases the probability of the Feature Envy smell occurring in the agglomeration. We can also observe that the bad smells evaluated are highly and positively correlated (Lift and χ^2). The most common Heterogeneous Agglomeration is the $LC, RB \rightarrow FE$, with a confidence of 0.756.*

5.2 Understanding Homogeneous Agglomerations

This section aims at answering our second research question, understanding if the presence of Homogeneous Agglomerations is significant in the source code. We an-

swer *RQ2: Are Homogeneous Agglomerations more frequent in the source code when compared to Heterogeneous Agglomerations?*. Since the association rules do not allow the detection of agglomerations composed only of the same type of smells, an additional analysis was conducted to verify their presence in the code. Similar to the last column from Table 5.1, Table 5.2 presents the number of classes that contain each type of Homogeneous Agglomerations. The first column from Table 5.2 presents the agglomeration type. The second one shows the absolute number of Homogeneous Agglomeration found in our dataset. The third column presents the participation of each Homogeneous Agglomeration in all agglomerations found, including the Heterogeneous ones. For instance, in Tables 5.1 and 5.2, we identified 1,787 different agglomerations in total, 1,156 Heterogeneous Agglomerations (Table 5.1) and 631 Homogeneous ones (Table 5.2).

Comparing the presence of the agglomeration in our dataset, the Homogeneous Feature Envy Agglomeration has a expressive presence in our dataset, with approximately 10% more instances than the second most common agglomeration, the LC, RB $\rightarrow$ FE. This finding implies that if we had used only association rules, we would have ignored a significant agglomeration. Consequently, when analysing bad smell agglomerations, it is necessary to evaluate if the Homogeneous Agglomeration are also present in the source code.

Table 5.2: Number of Homogeneous Agglomerations Found

Agglomeration	Count	Participation in All Agg.
Homogeneous Feature Envy	578	32.34%
Homogeneous Long Method	53	2.97%

***RQ2**: Homogeneous Feature Envy is the most frequent type of agglomeration in the source code, representing almost 32.5% of all agglomerations found, including the heterogeneous ones. The presence of Homogeneous Long Method is rare, being the rarest agglomeration type in the source code. This result implies that when evaluating agglomerations, it is necessary to consider the existence of Homogeneous Agglomeration.*

5.3 On the Agglomeration Density

After identifying the bad smell agglomerations, we can now answer the third research question: *RQ3: Are bad smell agglomerations denser in the source code than isolated smells?*. This analysis provides us information about the impact that size and quantity of bad smell agglomerations found have on our results. Figure 5.1 presents seven

box-plots that show the absolute numbers of classes in each system. These box-plots were organized according with the seven categories of bad smells. Figure 5.1.(a) represents the Heterogeneous Agglomerations. Figures 5.1.(b) and 5.1.(c) represent the Homogeneous Feature Envy (Homogeneous FE) and Homogeneous Long Method (Homogeneous LM), respectively. Finally, Figures 5.1.(d) to 5.1.(g) show the Isolated bad smell categories. They are, respectively, Isolated Feature Envy (Isolated FE), Isolated Long Method (Isolated LM), Isolated Large Class (Isolated LC), and Isolated Refused Bequest (Isolated RB). Figures 5.2 and 5.3 are organized in the same way as Figure 5.1. However, they depict the density considering the number of classes in each system, and the density considering the system KLOC, respectively.

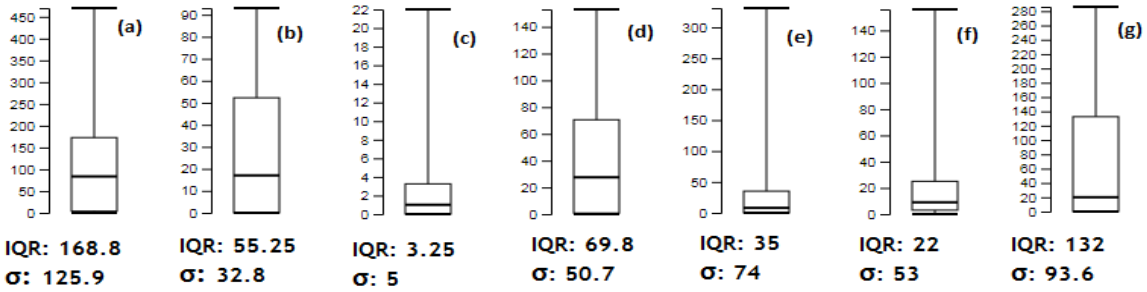

Figure 5.1: Absolute numbers of classes with bad smells. (a) Heterogeneous Agglomerations, (b) Homogeneous FE, (c) Homogeneous LM, (d) Isolated FE, (e) Isolated LM, (f) Isolated LC, and (g) Isolated RB.

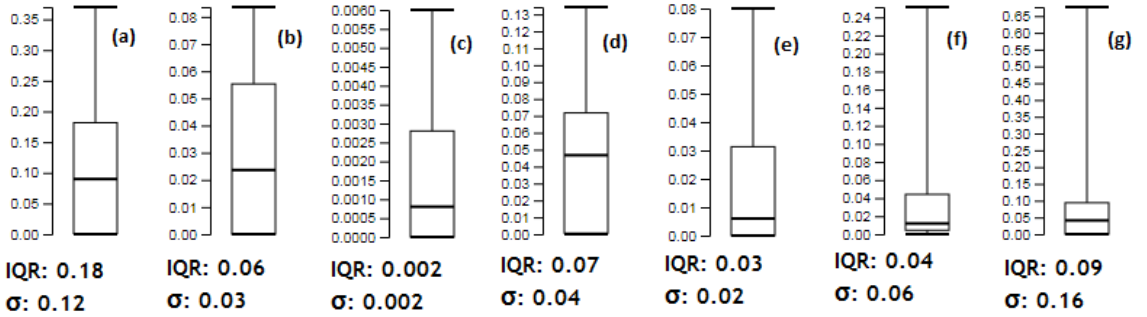

Figure 5.2: Percentile of classes presenting agglomerations. (a) Heterogeneous Agglomerations, (b) Homogeneous FE, (c) Homogeneous LM, (d) Isolated FE, (e) Isolated LM, (f) Isolated LC, and (g) Isolated RB.

To complement the box-plot analysis, we also calculated the Interquartile Range (IQR) and the standard deviation (σ), shown below each box plot. We can observe in Figures 5.1 to 5.3 that Heterogeneous Agglomerations are indeed more dense in the source code than other categories in terms of absolute numbers, percentage of classes affected by them and per KLOC, with means approximately 100, 0.1 and 1, respectively.

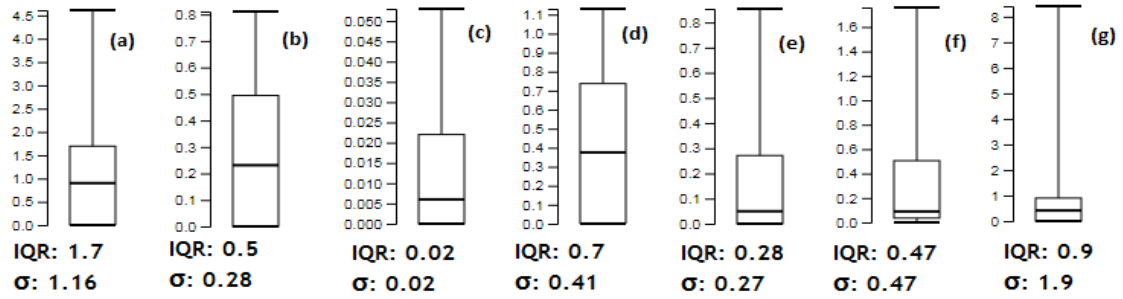

Figure 5.3: Density of bad smells per KLOC (a) Heterogeneous Agglomerations, (b) Homogeneous FE, (c) Homogeneous LM, (d) Isolated FE, (e) Isolated LM, (f) Isolated LC, and (g) Isolated RB.

In contrast, the Homogeneous Long Method is the least dense in the source code, with low means for all plots. We have found in Section 5.2 that Homogeneous FE is the most common agglomeration type in our dataset. However, when we observe Figures 5.1 to 5.3, we can point out that Isolated RB is more present in the systems than the Homogeneous FE, with means of 73.7 and 28.9 for the absolute numbers, and means of 0.89125 and 0.2626, respectively. This finding implies that Homogeneous FE is mitigated when considering system size than the Isolated RB, since there are more methods in a system than classes.

Analysing each category individually, we can observe from Figure 5.1 that Heterogeneous Agglomeration varies the most, with an standard deviation of 125.9, followed by Isolated RB (93.6) and Isolated LM (74). Both IQR and standard deviation show that the three categories suffer greatly with outliers. For example, Weka presents 470 Heterogeneous Agglomerations, and 268 Isolated RB. Meanwhile, Commons-Logging does not present Heterogeneous Agglomerations, but present 3 Isolated RB. This implies that certain domains of systems present more of these agglomeration categories than others. From Figure 5.2, we may observe that Isolated RB (g) are mitigated when considering the size of the system, *i. e.*, the number of classes. However, its variability is still high, with a standard deviation of 0.16, the highest one for this figure. Finally, from Figure 5.3, we can observe that the median for Heterogeneous Agglomeration is the highest, with a value of approximately 1 per KLOC. The median for Isolated RB follows the Heterogeneous Agglomerations, with the value of 0.4125.

It can be observed from the box-plots that size does affect our results. For example, Heterogeneous Agglomeration had the highest median for all plots. The absolute box-plot is a reflection of the quantity of bad smells found in each system. Observing in general values, some of them are really small. For example, for Isolated LM (e) and Homogeneous LM (c) the median is approximately 10 and 2, respectively.

This means that most systems had approximately only 10 and 2 classes affected by these agglomerations in all systems. The box-plots that consider the number of classes helped us to verify if the absolute number was mitigated when considering the number of classes of each system. Finally, the analysis of KLOC allowed us to understand if there is a high concentration of agglomerations in each KLOC. We concluded that for some categories, the behavior for all box-plots are similar, except for the Heterogeneous Agglomerations. Finally, we observed that for the Heterogeneous Agglomeration each KLOC, in median, has one instance of the bad smell.

***RQ3**: From the conducted analysis, we have found that Heterogeneous Agglomerations are denser in the source code, and the least dense is Homogeneous LM. We have also found that the Isolated Refused Bequest varies greatly on the absolute numbers, but it is less sensitive when considering the lines of code. Finally, we can not observe a category that have a high density of smells when considering the number of classes and the KLOC of each system.*

5.4 Impact of Agglomerations on Modularity

This section aims at investigating the impact of the bad smell agglomerations on the modularity metrics, answering the fourth research question, *RQ4: How the bad smell agglomerations impact on aspects of the system modularity?*. For this purpose, we calculated the Cohen's d for each pair of agglomerations, as explained in Section 4.3. To explain cohesively all pairs, we have taken the absolute value of each d calculated, and further classified each value according to the Cohen's d effect size. They are classified as follows. *Small:* when the value is below 0.5. *Medium:* values above 0.51 and below 0.8. *Large:* values above 0.81 [9]. To better complement our findings, the complete table of d values is present in Appendix A. We opted to compare each category of bad smell agglomeration in order to analyze in depth how the agglomerations compare to isolated and clean classes. This allow us to verify how the presence of more than one smell instance in a class may impact the modularity when compared to classes that have only one bad smell instance.

Table 5.3 complements this analysis by providing the data used to calculate the Cohen's d and the Coefficient of Variation (CV). The first column presents the metric and the second column presents in each line the agglomeration types under study. Finally, the third, fourth and fifth columns present in each line the mean, the standard deviation (σ), and the CV [52] for the pair Metric-Agglomeration Type. It is worth to mention that the mean and standard deviation are based on the entire system. We

opted to present these data in order to clarify what the results of the Cohen's d means, since readers can evaluate visually the difference between the values of each category and metric. We present this analysis per metric evaluated.

Table 5.3: Overall Variation of General Agglomerations

Metric	**Agg. Type**	**Mean**	**Std. Dev.**	**CV**
MaxNest	Heterogeneous	41.187	159.723	3.878
	Homogeneous FE	31.079	135.930	4.374
	Homogeneous LM	13.577	77.278	5.692
	Isolated FE	20.034	119.544	5.967
	Isolated LC	12.712	54.332	4.274
	Isolated LM	4.108	13.986	3.405
	Isolated RB	6.256	28.285	4.521
	Clean	21.393	85.470	3.995
RFC	Heterogeneous	50.887	43.961	0.864
	Homogeneous FE	24.697	23.858	0.966
	Homogeneous LM	27.558	15.983	0.580
	Isolated FE	16.115	13.794	0.856
	Isolated LC	71.132	70.179	0.987
	Isolated LM	24.483	13.810	0.564
	Isolated RB	11.552	13.320	1.153
	Clean	9.098	16.906	1.858
DIT	Heterogeneous	3.096	2.920	0.943
	Homogeneous FE	2.332	2.968	1.273
	Homogeneous LM	2.846	1.903	0.669
	Isolated FE	2.290	1.917	0.837
	Isolated LC	2.434	2.699	1.109
	Isolated LM	3.090	1.903	0.616
	Isolated RB	2.973	1.749	0.588
	Clean	2.134	2.036	0.954
WMC	Heterogeneous	61.055	65.424	1.072
	Homogeneous FE	26.007	42.938	1.651
	Homogeneous LM	34.865	32.660	0.937
	Isolated FE	14.837	15.490	1.044
	Isolated LC	100.862	106.481	1.056
	Isolated LM	19.485	22.342	1.147
	Isolated RB	13.025	13.818	1.061
	Clean	10.581	21.225	2.006
CBO	Heterogeneous	16.062	16.145	1.005
	Homogeneous FE	8.821	12.476	1.414
	Homogeneous LM	14.673	13.887	0.946
	Isolated FE	6.744	5.830	0.8964
	Isolated LC	22.125	21.261	0.961
	Isolated LM	12.397	7.840	0.632
	Isolated RB	7.151	5.135	0.718
	Clean	5.779	6.687	1.157

Figure 5.4 presents for each metric a heat map that shows the Cohen's d values between each possible pair of agglomeration types. The rows and columns represent the agglomerations. The color of the square represents the effect size. A black color shows that the effect size is Large. A dark gray shows that the effect size is Medium. A light gray shows that the effect size is Small. A white box represents that the difference is null due to the comparison of the same type of agglomeration. For example, for the CBO metric (e), the d value of Heterogeneous Agglomeration and Clean classes is Large, with a black box. However, for the same metric, the difference between Homogeneous LM and Isolated LM is Small (light gray box).

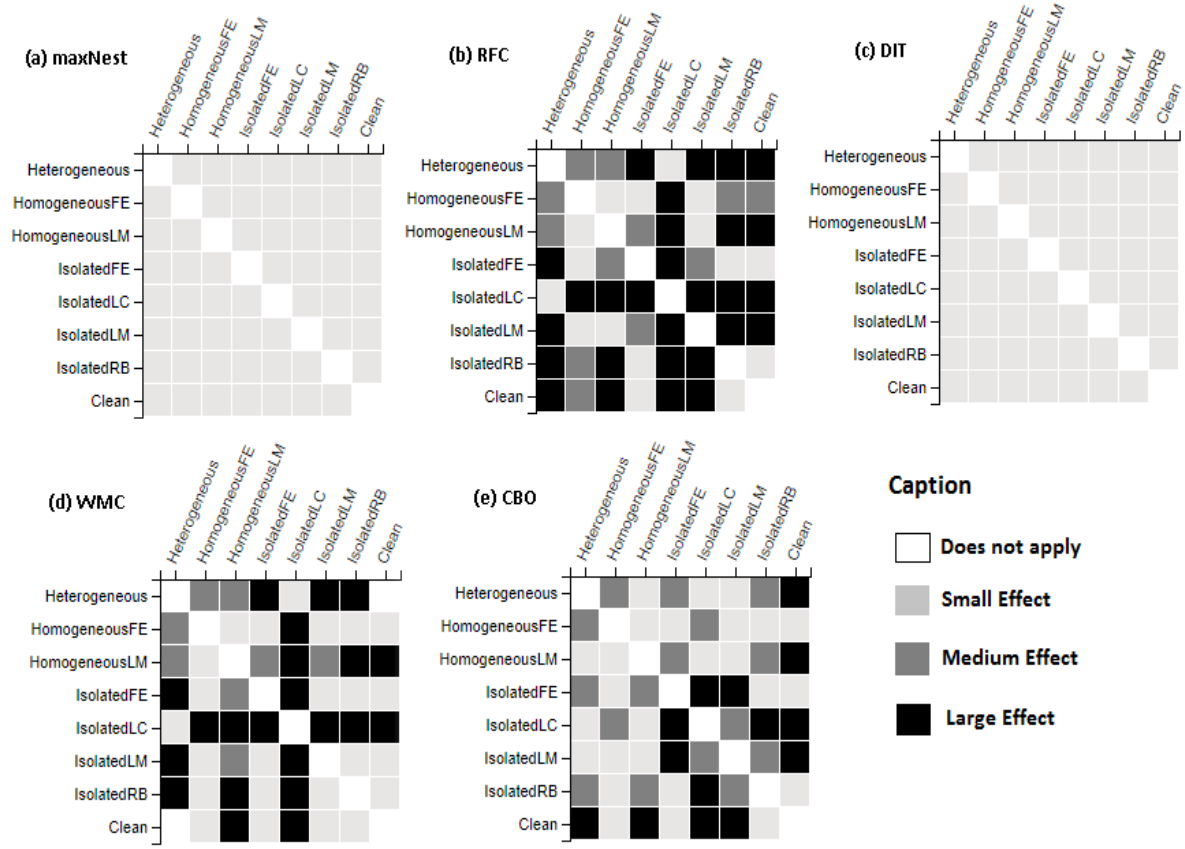

Figure 5.4: Pairwise Cohen's d per metric. (a) maxNest. (b) RFC. (c) DIT. (d) WMC. (e) CBO.

MaxNest: a higher maxNest indicates that the code is more complex [3], due to the nesting of blocks. In Figure 5.4.(a), we can observe that all effects are Small. The CVs values indicates that for most agglomerations categories, the standard deviation is approximately 4 times the mean, and they range from 3.405 to 5.967. Observing only the heat map, we can not take any conclusion. Observing the original values present in Appendix A, the highest d value was for the pair Heterogeneous-Isolated LM, with

a value of 0.3270. We can also observe that for most lines, the highest value for the pairs are with Heterogeneous Agglomeration. Even though we do not observe a large difference between the means, the mean of Heterogeneous Agglomeration is slightly higher than the other categories. Consequently, they are a bit more complex than other categories.

RFC: the largest the RFC value is, the highest is the probability of faults detection [4]. These faults may be introduced due to system complexity and poor understandability. We can observe in Figure 5.4.(b) that most of the Large effects are the pairs with the Isolated LC category. We also observe that Heterogeneous and Isolated LM have a high effect too. From Table 5.3, we can see that for most of the categories, the CV is less than one, indicating that these categories do not suffer from high variability. This result is interesting because the RFC measure is associated with the definitions of the LC and LM smells. We highlight the fact that for Heterogeneous Agglomeration, all rules found present Large Class, Long Method, or both. Observing the original values, the highest d value is for the pair Heterogeneous-Clean (1.2547). The signals for the Heterogeneous and Isolated LC are mostly positive, indicating that their means are indeed higher than other categories. It is interesting to notice that the d value for Heterogeneous-IsolatedLC is -0.3457, indicating that Isolated LC have a large mean value than the Heterogeneous ones. We can conclude that classes that have Heterogeneous Agglomerations or only presents Large Class are more fault prone.

DIT: this metric indicates that the module has long chains of inheritance and may increase the fault proneness [3]. From Figure 5.4.(c), we can not observe any trends in our data, similar to the MaxNest metric. This result is also reflected on the fourth column from Table 5.3, with most CVs bellow 1. This indicates that the standard deviation of each category is stable and similar to the mean. We expected to find that classes and rules that contained Refused Bequest presented a higher effect, but this expectation could not be confirmed. When observing the original values in Appendix A, the highest d values were in the pairs that considered the Clean classes. We can also observe that pairs with Heterogeneous Agglomerations and Isolated RB are slightly and positive higher than other categories. In conclusion, we can only observe a slight effect for those two categories and, consequently, they tend to be more fault prone than other categories.

WMC: this metric indicates classes that are more complex and fault prone [4; 3]. It can be observed from Figure 5.4.(d) that the highest effects are pairs with Isolated LC. This may be due to the definition of Large Class, that is directly associated with complexity. Another interesting observation that can be made is for the Homogeneous LM category, that had 3 Large effects and 3 Medium effects. This is interesting because

of the definition of this category, in which the class has at least two long methods, and the definition of the metric, that considers the line of code of each method. Excluding Heterogeneous and Isolated LC, all effects are positive. This finding is in line with the definition of both Homogeneous LM and the metric. That is, Long Methods have more lines of code, and consequently, a higher WMC. From Table 5.3, we can observe that most CVs are approximately 1, with the exception of Homogeneous FE and Clean classes. Finally, we can conclude that Heterogeneous, Isolated LC and Homogeneous LM classes are more complex and more fault prone than other classes.

CBO: a higher value for this metric means that the class is highly coupled to other classes. In Figure 5.4.(e), we can observe that most of the Large effects are of the pairs with Clean classes. However, when observing their original value in Appendix A, all of them are negative, implying that Clean classes have a CBO value lower than the other categories. This is expected, since in Clean classes we expect that they have a low coupling. Otherwise, having high coupling may indicate that a Feature Envy is occurring due to the smell definition. We can observe from Table 5.3 that most categories have a CV bellow 1, indicating that the categories variation is low. Beyond these findings, we can observe that the highest effects are associated with pairs that contains Isolated LC. This result may be a reflection of the nature of the Large Class smell, that is large in size and implements different functionality. Consequently, their coupling to other classes are higher. We can conclude that the coupling of Clean classes are lower than the other categories. Isolated LC also tends to be more coupled to other classes, since it implements much more functionality that it should. Consequently, they are more difficult to understand, due to their high level of interactions with other classes.

***RQ4**: From our analysis, we observed that Heterogeneous Agglomerations do impact on complexity, fault proneness, and inheritance. Their impact was not confirmed only for the CBO metric. We also found that Homogeneous LM is more complex and fault proneness when considering the WMC metric, and that Isolated LC does impact the source code modularity. In conclusion, agglomerations do impact more aspects of source code modularity than other categories.*

5.5 Discussion

In this section, we discuss our findings, and their implication for practitioners. It is interesting to notice that the presence of Homogeneous Agglomeration is not expected due to the definition of the smells that composes it. We discuss the presence of Homo-

geneous Feature Envy, since our findings indicate that they have a significant presence in the source code. It is worth to remember the definition of a *Large Class* and a *Feature Envy*. The former is when a class tries to do too much. The latter is when a method is more interested in being in another class, *i.e.*, this method should be moved to the class that it belongs. The presence of Homogeneous *FE* may be explained since the smell existence implies that the class is doing too much. Beyond the detection strategies that were used in the detection tools, we discuss the number of smells present in these agglomerations.

Figure 5.5 presents two box plots: (a) it indicates the number of *FE* smells in each FE Agglomeration; (b) it shows how the FE Agglomerations are distributed in each system. Complementing Figure 5.5, Table 5.4 shows the mean and standard deviation for each box plot. We can observe that the median and mean of the numbers of *FE* in the Homogeneous FE Agglomeration is low, with values of 2 and 3.349, respectively. This provides us evidence that the class is overall cohesive. We can observe in Figure 5.5.(b) that some systems do not contain this agglomeration. However, both Figure 5.5 and Table 5.4 present high variance in number of *FE*. This is expected, since some systems have more *FE* than others. The largest agglomeration contain 32 *FE* smells. However, 75% of the observed Homogeneous FE Agglomerations present 5 or less *FE*s. To understand the effect of the outliers, we calculated the quantile value at 90%. The result was 6, being in the range of the mean plus one standard deviation. Moreover, the results in Figure 5.5 are consistent with Table 5.4. We can conclude that the quantity does affect the presence of such agglomerations.

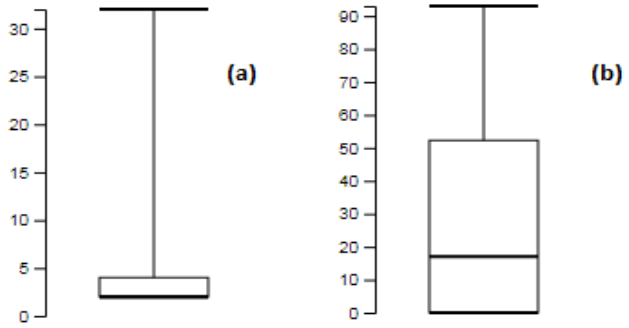

Figure 5.5: FE Agglomerations. (a) Presents the absolute number of the FE smell present in each Homogeneous FE found. (b) Presents the number of Homogeneous FE in each system.

These findings imply that identifying Homogeneous Agglomeration is a more subtle task, due to the number of smells that compose it being small. This task may

Table 5.4: Overall statistics for FE Agglomerations

Type	**Mean**	σ
Num. FE Per Class	3.349	2.509
FE Agg. Per System	28.9	32.831

be supported with the use of tools that detect such agglomerations. We can also observe from our analysis that Homogeneous Agglomerations have a significant presence on the source code (Section 5.2) and they indeed impact on the system modularity (Section 5.4).

The existence of Homogeneous Long Method Agglomeration is a paradox due to their definition. That is, a class should have only one dominating method. A possible explanation for this result is the detection strategy employed by detection tools. Even though we used a total of 5 tools to detect the smells, 4 of them use metrics to automatically classify each instance. This provides us insights that current metric-based strategies do not consider that a class can only have one dominating method, and this is not a direct problem with our construction of the dataset. Consequently, this restriction should be addressed by tool's developers.

Now we discuss how the selected metrics are affecting our results. The selection of which modularity metrics to evaluate were greatly influenced by their use in the literature [8; 11] and if their impact were evidenced by other studies [3; 4]. After selecting them, they were calculated using a tool that was used in other studies in the area [11]. We may also point out that some of the metrics analyzed are used in the detection strategy of the tools. However, when analyzing the Coefficient of Variation, we could not observe for these metrics a high difference between the categories. Consequently, we believe they do not bias our analysis.

We have also calculated the Pearson Correlation [52] to understand if the metrics are highly correlated, with degree of freedom of 21585 and with 95% of confidence. Table 5.5 presents in the first column the pairs of metrics. The second column presents the index of correlation r for the pair (metricA,metricB). The third column presents the t value used to estimate the Confidence Intervals (CI) in column four. The fourth column presents the Confidence Interval for the correlation between the two items. Finally, we present in the last column the obtained p-value.

We can observe from the Confidence Interval that all correlations are significant, because they do not include 0 in their interval, and all p-values calculated are smaller than 0.001. We can observe that the highest correlations are for the pairs CBO-RFC and WMC-RFC, with r of 0.74 and 0.78, respectively. All the correlations found are positive. The CIs can be interpreted as the interval that contains the real correlation

between the two variables with 95% of confidence. We can observe that all intervals are small in range. We can conclude that most pairs have a small correlation. As consequence, our findings are not being affected severely by the multicolinearity of the metrics.

Table 5.5: Correlation between pairs of metrics

Pair (x,y)	**Correlation r**	**t value**	**Confidence Interval**	**p-value <**
CBO and WMC	0.5265409	90.994	(0.5168311, 0.5361152)	2.2e-16
CBO and DIT	0.1882055	28.154	(0.1753056, 0.2010408)	2.2e-16
CBO and RFC	0.7420599	162.64	(0.7360015, 0.7479915)	2.2e-16
CBO and MaxNest	0.2143688	32.444	(0.2016053, 0.2270595)	2.2e-16
WMC and DIT	0.04234253	6.2265	(0.02901890, 0.05565111)	4.58e-10
WMC and RFC	0.7844838	185.85	(0.7792992, 0.7895610)	2.2e-16
WMC and MaxNest	0.278678	42.632	(0.2663281, 0.2909364)	2.2e-16
DIT and RFC	0.1278837	18.944	(0.1147394, 0.1409832)	2.2e-16
DIT and MaxNest	0.1046213	15.456	(0.09140884, 0.11779689)	2.2e-16
RFC and MaxNest	0.2676165	40.806	(0.2551875, 0.2799570)	2.2e-16

5.6 Concluding Remarks

In this chapter, we discussed the main findings of our research and answered RQ1 to RQ4, questions related to understand how agglomerations impact on the source code modularity. We have found that, beyond having a significant presence in the source code, Heterogeneous Agglomeration does impact more aspects of modularity than classes with only one bad smell. We also found that Homogeneous Agglomerations have a significant presence in the source code. This finding is novel in the study of agglomerations. Finally, we analyzed the presence of such agglomerations and found that when considering number of classes and KLOC, the densities are higher.

In the next chapter, we aim to study in depth the association rules found in Section 5.1. We want to evaluate which rule impacts the most the source code modularity. For this purpose, we calculate their density and the Cohen's d. We also compare them to other possible agglomerations that appeared in our dataset, but were not considered as meaningful by the association rule algorithm.

Chapter 6

Understanding Heterogeneous Agglomerations

In the previous chapter, we have found that Heterogeneous Agglomerations are more frequent in the source code than other classes that contain bad smells. In total, we derived four association rules from our dataset, using the association rules technique. We also found that such agglomerations are indeed more dense in the source code, and they do impact on the software modularity. These evidences raise the need of understanding if all Heterogeneous Agglomerations are similar in terms of density and impact on the modularity metrics.

This chapter aims at answering our last research question: *Does the types of Heterogeneous Agglomerations have an uniform behavior in terms of density and impact on aspects of system modularity?* The answer to this question is shown in the following two sections. Section 6.1 provides an analysis of how the different Heterogeneous Agglomerations are distributed in our systems. This analysis allows us to verify if some of these agglomerations are affected by the system size. Finally, Section 6.2 provides an analysis of how they affect our modularity metrics. This help us to understand if there is a kind of Heterogeneous Agglomeration that impacts the most the metrics in comparison to other types. Section 6.3 concludes this chapter.

6.1 Heterogeneous Types Density

This section aims at answering the first part of our last research question, *i.e.*, how the different types of Heterogeneous Agglomeration density behaves in our dataset. With this question, we aim at verifying (i) if the association rules found are mitigated when the system size is considered, and (ii) if the category of Heterogeneous Agglomeration

in the previous chapter considered only the association rules found. We aim at understanding if other types of agglomerations that did not appear in the association rules algorithm are more dense in the source code than those that were found by it.

In order to understand how different types of Heterogeneous Agglomeration behaves along the systems, since the smells that compose them may play an important role in terms of numbers, we analyze the density for all possible Heterogeneous Agglomerations present in our dataset. For this purpose, Table 6.1 presents in the first column the abbreviation of the types of agglomerations found. The second column presents the smells that compose the agglomeration. Finally, the last column presents the method by which they were found. The first four agglomerations were found by the association rules algorithm. For example, *LcRbFe* is one of the four Heterogeneous Agglomeration found by the association rule algorithm, and is composed of the Large Class, Refused Bequest and Feature Envy smell. In contrast, the combination *RbFe* was found in our dataset, but is not strong enough to be outputted in the Association Rule, and is composed of two smells: Refused Bequest and Feature Envy.

Table 6.1: Heterogeneous Agglomeration Types and the Smells that Composes it

Type of Heterogeneous Agglomeration	Bad Smells	Where the combination was found
LcLmFe	Large Class, Long Method, Feature Envy	Association Rule Algorithm
RbLmFe	Refused Bequest, Long Method, Feature Envy	Association Rule Algorithm
LcRbFe	Large Class, Refused Bequest, Feature Envy	Association Rule Algorithm
LcRbLmFe	Large Class, Refused Bequest, Long Method, Feature Envy	Association Rule Algorithm
LcRbLm	Large Class, Refused Bequest, Long Method	Dataset
LcFe	Long Class, Feature Envy	Dataset
LcRb	Large Class, Refused Bequest	Dataset
LcLm	Large Class, Long Method	Dataset
RbFe	Refused Bequest, Feature Envy	Dataset
RbLm	Refused Bequest, Long Method	Dataset
FeLm	Feature Envy, Long Method	Dataset

Figures 6.1, 6.2 and 6.3 present eleven box-plots representing the absolute value of Heterogeneous Agglomerations in all analyzed systems, their density in terms of classes and KLOC, respectively. To complement the analysis of the figures, Table 6.2 shows the InterQuartile Range (IQR) and standard deviation (s.d.) for each category of Heterogeneous Agglomerations. Columns two and three presents the variation of the

box-plots in terms of absolute values. Columns four and five presents the variation of the presence of the agglomerations considering the number of classes in each system. Finally, columns six and seven presents the variation for the box-plots that considers the number of agglomerations per KLOC.

When observing Figure 6.1, all medians of the association rule agglomerations are small. Only the medians of *LcFe* and *RbFe* are expressive compared to the other categories, with approximately 10 and 20 agglomerations per system, respectively. For all other agglomerations, most medians are in the range of 0 to 2, per system. When other size measurements are considered, from Figures 6.2 and 6.3, we can observe that these agglomerations are highly dense when considering the number of classes and on the KLOC, being reflected in the low values for most categories. Only *RbFe* is maintained as the most dense, with a density of 0.02 classes being affected by them, and each 4 KLOC presenting one of this agglomeration.

In terms of absolute value, as can be seen in Figure 6.1, the systems in our dataset presents more *RbFe* and *LcFe*, with mean of 34.85 and 14.05, respectively. Both agglomerations presents the highest variation in the second and third column from Table 6.2. However, when mitigated by the number of classes and per lines of code, *LcFe* is more stable and presents a similar behavior of the other categories, *i.e.*, small values and low variation. However, when observing the *RbFe* and Figure 6.3, the category is not mitigated by KLOC. Even more, the third quartile is at 0.5245, meaning that, in general, each 2 KLOC present one *RbFe* agglomeration.

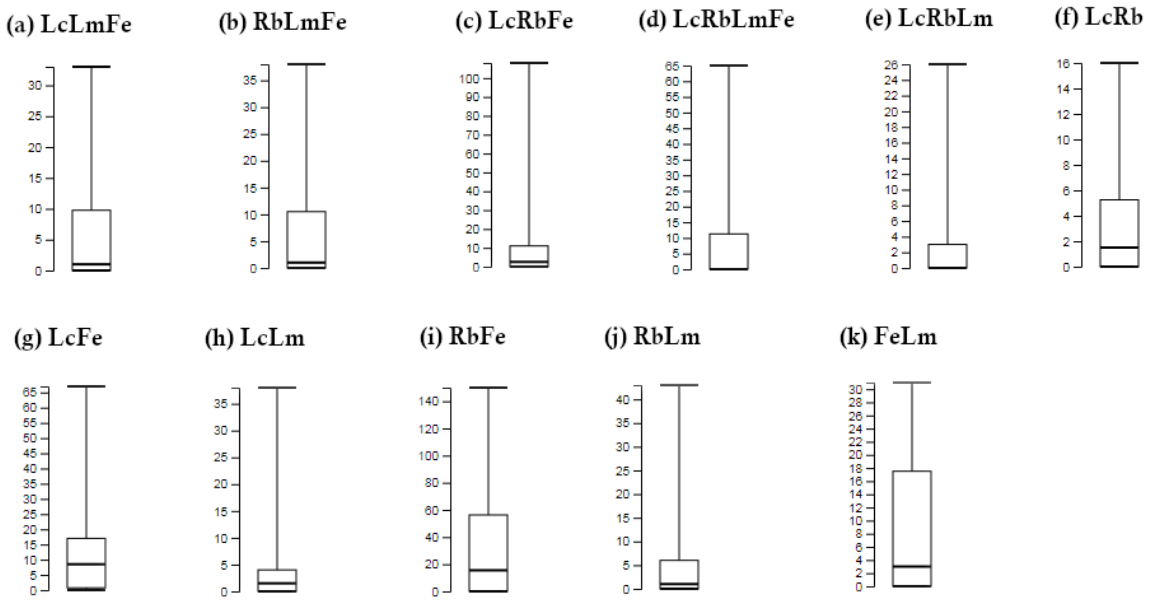

Figure 6.1: Density of Heterogeneous Types in All Systems.

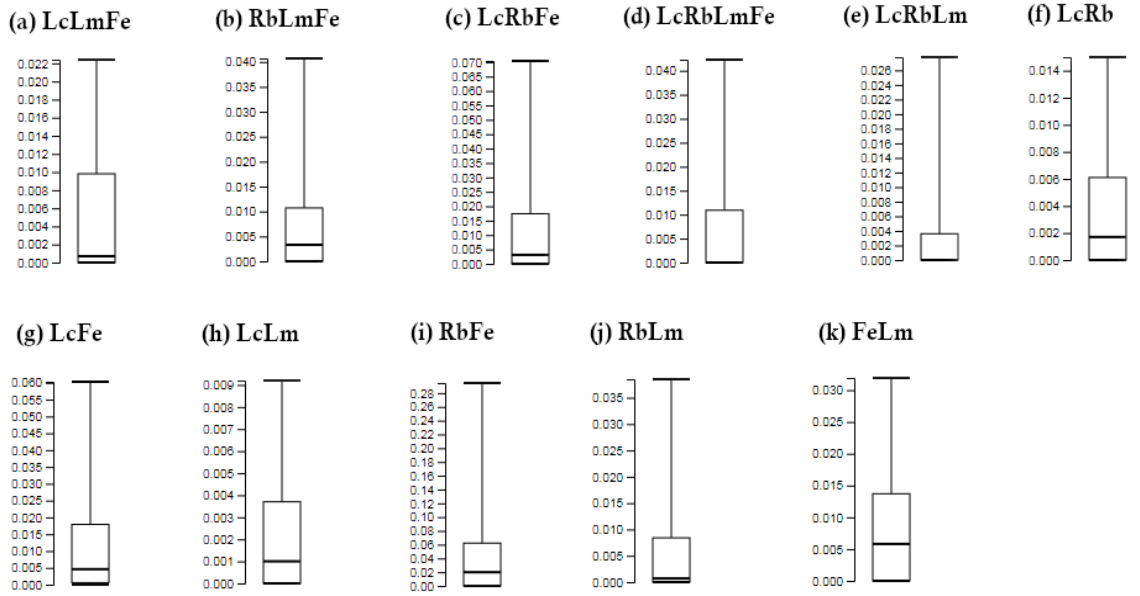

Figure 6.2: Density of Classes Affected by Heterogeneous Agglomerations Segregated by Type.

From Figures 6.1 to 6.3, we can observe that when isolated, the different types of agglomerations do not present a different behavior in term of density. This observation is also reflected in Table 6.2, with low IQRs and standard deviations for most categories. This finding may be due to the large quantity of different Heterogeneous Agglomerations, eleven in total. We can conclude that they are more concentrated when observed together. We cannot conclude that there is a dominant category of Heterogeneous Agglomerations that is much more concentrated than others. We may also conclude that the categories' density is homogeneous, and their collective contribution is that affect the results obtained in the aggregated Heterogeneous Agglomeration analysis. From the three figures, we can also observe the presence of several systems that are outliers. For example, Weka presents 108 LcRbFe agglomerations and 67 LcFe agglomerations. Meanwhile, Hibernate, even though the system is larger than Weka in lines of code and number of classes, has 0 LcRbFe and 10 LcFe agglomerations.

***RQ5**: From our analysis, we can conclude that the density behavior for all categories of Heterogeneous Agglomeration is similar. The highest concentrated agglomeration was RbFe. However, when the number of classes are considered, the RbFe is mitigated.*

6.2 Impact of Heterogeneous Agglomerations

Since we found that Heterogeneous Agglomerations do impact the modularity metrics more than other types of classes, we can further investigate how the different types

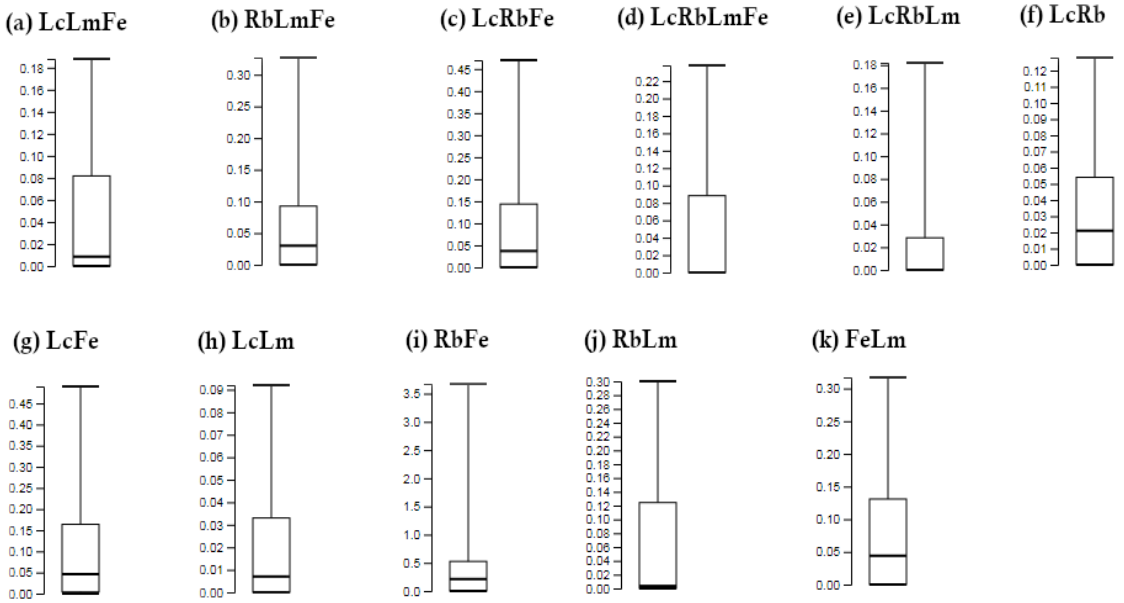

Figure 6.3: Density of Heterogeneous Agglomeration Types per KLOC.

Table 6.2: Variation Measurements for Heterogeneous Agglomerations

Agg. Type	IQR Absolute	s.d. Absolute	IQR per Class	s.d. per Class	IQR per KLOC	s.d. per KLOC
LcLmFe	9.75	9.24	0.01	0.007	0.08	0.06
RbLmFe	10.5	11.31	0.01	0.011	0.09	0.09
LcRbFe	11.0	25.33	0.02	0.018	0.14	0.138
LcRbLmFe	11.25	14.93	0.01	0.01	0.09	0.07
LcRbLm	3.0	6.098	0.0	0.006	0.03	0.04
LcRb	5.25	5.048	0.01	0.004	0.05	0.039
LcFe	16.25	18.83	0.02	0.018	0.16	0.13
LcLm	4.0	8.42	0.0	0.003	0.03	0.026
RbFe	56.25	43.22	0.06	0.069	0.52	0.812
RbLm	6.0	12.405	0.01	0.01	0.12	0.095
FeLm	17.25	10.418	0.01	0.010	0.13	0.096

of Heterogeneous Agglomeration found in our dataset behaves. With this purpose, we answer the last part of our fifth research question, that aims at investigating if there is a Heterogeneous Agglomeration that contributes the most to the degradation of modularity. Mainly, from previous findings, we highlight that the presence of Large Class in such agglomerations may impact the results found. We further analyze the impact of Heterogeneous Agglomerations on the modularity metrics through the use of Cohen's d and the Coefficient of Variation (CV). We expect to raise exploratory evidences if the existence of certain bad smells on the agglomeration influences the most the metric values.

Figure 6.4 presents five heat maps with the Cohen's d values for each category

of Heterogeneous Agglomeration found in our data. Each heat map concerns one of the evaluated metric. For legibility purposes, the agglomerations are categorized in a similar way as of in the density section. For example, *LcFe* is an agglomeration composed of only Large Class and Feature Envy. To complement the heat map, the complete table with the original Cohen's d value can be found in Appendix B. We also present in Table 6.3 the mean, standard deviation and CV for each pair Metric - Heterogeneous Agglomeration Type. The first column presents the metric. The second column present the Heterogeneous Agglomeration type. The third column presents the mean for the pair Metric-Heterogeneous Agglomeration Type. Similarly, the fourth and the last column presents the standard deviation (Std. Dev.) and Coefficient of Variation (CV) for each pair, respectively.

maxNest: From the heat map in Figure 6.4.(a), we can observe that most pairs have a Small Effect, except for three pairs: LcFe-RbFe, LcFe-RbLm, and LcFe-FeLm. It is interesting to notice that there is an intersection between two of these pairs, both agglomerations containing the Feature Envy smell, except for LcFe-RbLm. When the original d value is observed in Table B.5 (Appendix B), the highest value is for LcFe-RbLm (0.5742), followed by LcFe-FeLm (0.56) and LcFe-RbFe (0.5023). It seems that when there is a Long Method in the second pair, the difference is higher for these pairs. From Table 6.3, we can observe that there exist a high variability in this metric, with high standard deviation for all categories. The CV also reflects this variability, with FeLm being the most variable in terms of its mean, with a CV of 7.120. All categories have a CV above 2. This means that the standard deviation for all categories is at least two times their mean, indicating the presence of several outliers. We can conclude that, for this metric, most effects are Small. However, it seems that when a agglomeration that do not have the Long Method smell is compared to one that present it, the class that does not contain it tends to present a higher maxNest. This means that they are slightly more complex than classes that do not contain the LM smell.

RFC: From heat map in Figure 6.4.(b), we can observe a predominance of Small Effects for most categories. However, some pairs present a significant presence of Large Effects, mainly those paired with the LcLmFe, RbLmFe, LcRbFe, LcRbLmFe, RbFe, LcLm, RbLm and FeLm. Most of the Large Effects are due to combinations composed of these agglomerations. This is interesting, since the four association rules achieved some Large Effects, and they are mostly positive (first four lines in Table B.4 in Appendix B). When observing the original d values, pairs with RbFe presents the highest values, and the signal of the value raises evidences that classes that contains them tend to have a smaller RFC value than other agglomerations. The highest d value found was for the pair LcRbLmFe-RbFe (1.6648). From Table 6.3, we can observe that

Table 6.3: Variation of Heterogeneous Agglomerations

Metric	Agg. Type	Mean	Std. Dev.	CV
MaxNest	LcLmFe	59.705	177.414	2.972
	RbLmFe	17.461	82.178	4.706
	LcRbFe	99.079	287.885	2.885
	LcRbLmFe	24.181	103.750	4.291
	LcRbLm	35.246	107.568	3.052
	LcFe	111.511	261.664	2.347
	LcRb	40.639	134.372	3.306
	LcLm	36.757	92.485	2.516
	FeLm	6.363	45.304	7.120
	RbFe	15.411	68.816	4.465
	RbLm	4.955	20.424	4.122
RFC	LcLmFe	80.156	50.602	0.721
	RbLmFe	41.039	25.567	0.623
	LcRbFe	76.736	55.810	0.727
	LcRbLmFe	87.051	47.615	0.547
	LcRbLm	75.579	44.723	0.592
	LcFe	71.164	55.047	0.776
	LcRb	58.639	43.526	0.742
	LcLm	53.716	26.885	0.501
	FeLm	31.935	23.329	0.731
	RbFe	27.097	18.070	0.667
	RbLm	31.854	16.102	0.505
DIT	LcLmFe	2.090	2.450	1.172
	RbLmFe	3.868	2.355	0.609
	LcRbFe	3.215	2.732	0.850
	LcRbLmFe	3.058	1.383	0.452
	LcRbLm	3.333	1.200	0.360
	LcFe	2.248	2.559	1.138
	LcRb	3.458	1.491	0.431
	LcLm	2.662	3.620	1.360
	FeLm	3.023	7.113	2.353
	RbFe	3.307	1.702	0.515
	RbLm	3.627	1.545	0.426
WMC	LcLmFe	115.467	91.316	0.791
	RbLmFe	31.033	25.692	0.828
	LcRbFe	103.624	73.569	0.710
	LcRbLmFe	109.284	70.263	0.643
	LcRbLm	97.070	89.556	0.923
	LcFe	106.374	72.430	0.681
	LcRb	81.833	39.128	0.478
	LcLm	71.784	28.852	0.402
	FeLm	25.988	33.570	1.481
	RbFe	22.586	18.975	0.840
	RbLm	25.246	22.952	0.909
CBO	LcLmFe	19.762	16.316	0.826
	RbLmFe	13.770	6.834	0.496
	LcRbFe	21.996	18.742	0.852
	LcRbLmFe	25.277	17.632	0.698
	LcRbLm	21.211	13.297	0.627
	LcFe	19.107	30.304	1.586
	LcRb	18.709	13.953	0.746
	LcLm	18.800	14.527	0.773
	FeLm	12.222	9.384	0.768
	RbFe	10.696	6.133	0.574
	RbLm	14.052	6.265	0.456

both standard deviation and CV are small, with all CVs bellow 1. This means that this metric does not have a high variability when considering their mean value. From these analysis, we can conclude that the association rules found does impact the most on the RFC metric, due to the positive signals of the d values. Consequently, classes that contain one of these agglomerations (LcLmFe, RbLmFe, LcRbFe, and LcRbLmFe) are more fault prone than other Heterogeneous Agglomerations.

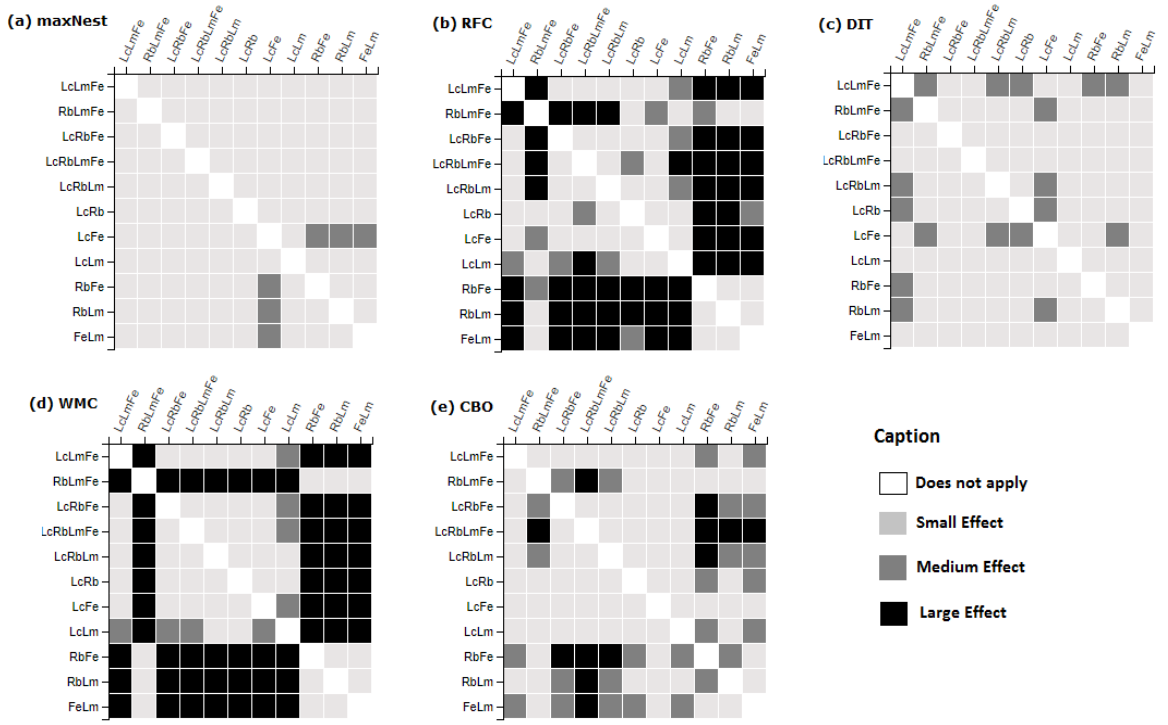

Figure 6.4: Pairwise Cohen's d per metric for Heterogeneous Agglomerations.

DIT: Figure 6.4.(c) shows the heat map for the heterogeneous categories. We can observe that some pairs have a Medium Effect, but the predominance of Small ones are higher. The pairs that achieved Medium Effects are mainly with LcLmFe and LcFe. Observing the original values in Table B.3 (Appendix B), the highest effects are from LcLmFe, all of them positive in their signal. Similar to the RFC metric, the DIT variation is small, with most CVs bellow 1. However, FeLm presented the highest variation, with a CV of 2.353. We can conclude that classes that contains LcLmFe impacts the most on the DIT metric, however, this effect is Medium. Consequently, this agglomeration is more fault prone when considering this metric.

WMC: Heat map in Figure 6.4.(d) shows the d values for all the Heterogeneous Agglomerations types. We can observe a dominance of Large Effects for the pairs with

RbLmFe, RbFe, RbLm, and FeLm. All other Large Effects can be observed when comparing to these four pairs. Observing the original values in Table B.2 (Appendix B), the effect of RbLmFe is mostly positive, indicating that classes that contain this agglomeration have a much higher WMC value. Moreover, classes that contains RbFe, RbLm, and FeLm have a much lower WMC, explained by the high d value and negative signal. This result is interesting, because all classes have a non-null intersection when considering the smells that compose them. Only when three of these smells appears together that the effect is Large. We can also observe in Table 6.3 that the categories are stable, with low standard deviation and CV, except for the FeLm smell. All CVs are below 1. We can conclude that RbLmFe impacts the most the WMC metric, presenting much larger values than the other agglomerations. Consequently, they are more complex and fault-prone than other categories.

CBO: Figure 6.4.(e) presents the heat map for the Cohen's d value of all Heterogeneous Agglomeration types. We can observe that most effects are Small, but pairs with LcRbLmFe, RbFe, and FeLm present several Large and Medium effects. When observing the original d value in Table B.1 (Appendix B), the Large Effects are positive, and for RbFe and FeLm, the Large Effects are negative. We can also observe from Table 6.3 that most categories, except LcFe, have a low variation, with most of them having a CV below 1. However, when observing the heat map, all combinations of pairs that contain LcFe have a Small Effect. We may conclude that the high variation is not affecting our results. We can also conclude that LcRbLmFe impacts the most on this metric. Consequently, this agglomeration is more coupled to other classes than other agglomerations.

***RQ5**: From our analysis, we can conclude that each metric has a set of agglomerations that affects the most them. However, it is interesting to notice that for all metrics, at least one agglomeration found by the association rule algorithm is among the ones that have a positive impact on the metric. This indicates that when a class present these agglomerations, their metric values are generally larger than those from other agglomerations categories. Consequently, they are more complex, fault prone and present a higher coupling than other Heterogeneous Agglomeration types.*

6.3 Concluding Remarks

In this chapter, we aimed at (i) investigating if the different types of Heterogeneous Agglomerations have different densities than the others, and (ii) how they can impact the source code modularity. We have found that these agglomerations are highly dense in the source code, with most of them presenting a similar behavior, except the RbFe,

that was only mitigated when considering the number of classes. We could also verify that the agglomerations found by the association rules algorithms do impact the most on all the analyzed metrics. In the next chapter, we discuss our main conclusions and possibilities for future works.

Chapter 7

Conclusions

In this dissertation, we aimed at exploring the existence and impact of bad smell agglomerations. Previous works [1; 29; 67; 39] identified that when bad smells occur together in the source code, they make the code harder to understand, to modify and to evolve. This is due to the intrinsic characteristics of the smells that hurts good modularity practices. They may affect several aspects at a time, such as coupling, complexity, inheritance and the single responsibility principle. This raised our topic of investigation: *How different bad smell agglomerations may impact aspects of source code modularity?*. To answer this question, we have found in the literature two studies that evaluated the software metrics that represent aspects of modularity [3; 4]. These metrics help us to raise initial evidences that bad smell agglomerations may impact the software modularity. This impact can be measured through the differences between the metric values for different bad smell classes and agglomerations. Consequently, we used five software metrics as proxies of software modularity. The metrics are: Coupling Between Objects (CBO), Weighted Method Class (WMC), Depth of Inheritance Tree (DIT), Response set of a class of objects (RFC), and max nest blocks (maxNest).

Before exploring their impact, we have to identify bad smell agglomerations. We have built a dataset composed of 20 Java systems from the Qualita Corpus [56]. We chose them carefully considering their different sizes, domains and maturity. This allowed us a more generalizable result. With systems in hands, we used a voting strategy in order to obtain our ground truth. We investigated four smells: Large Class, Long Method, Refused Bequest, and Feature Envy. We chose them because (i) they represent different symptoms of modularity degradation. Besides, (ii) Large Class and Long Method are among the most studied smells [11]. In contrast, Refused Bequest and Feature Envy needed more investigation. We then chose to co-study them in order to expand the current knowledge by investigating their interaction. (iii) Finally, the

selected smells were detected by at least three detection tools. This restriction assures us that, for every smell, at least two detection strategies have agreed that the instance is positive/negative.

After building the dataset, the classes in the systems were classified according with the quantity and types of smells that it presents. We classified them as Heterogeneous Agglomerations, Homogeneous Feature Envy, Homogeneous Long Method, Isolated Large Class, Isolated Long Method, Isolated Feature Envy, Isolated Refused Bequest, and Clean. This classification allowed us to compare different kinds of classes. Finally, we calculated the selected metrics for each class in our dataset.

We then extracted the necessary information to answer our research questions. In order to identify the Heterogeneous Agglomerations, we used the association rules technique. This mining technique allows the identification of only strong rules, *i. e.*, agglomerations that have a strong relationship between the smells that compose it. Moreover, this technique has the restriction that the same smell cannot appear in the two sides of the rules at the same time. That is, the algorithm does not discover agglomerations formed by the same type of smell. To address this limitation, we used frequency and variation statistics to characterize these agglomerations, even though they are less powerful. In our data, we have found the Homogeneous Feature Envy and the Homogeneous Long Method agglomerations.

We have identified four association rules: $LC, RB \longrightarrow FE$; $RB, LM \longrightarrow FE$; $LC, LM \longrightarrow FE$; and $LC, LM, RB \longrightarrow FE$. All of them have a significant presence in the source code. We have found that all the smells belonging to these agglomerations are highly correlated and dependent, as could be observed in the χ^2 test. With the frequency statistics, we have found that the Homogeneous Feature Envy is the most frequent agglomeration type, even when comparing with the association rules found. We also analyzed how many FE smells, in median, an agglomeration has. The median is approximately 2.5, and the 90th-quartile was 6. If we used association rules alone, we would have ignored such a significant agglomeration type. Moreover, this technique does not provide us with all information that association rule does.

After identifying which smell agglomerations were frequent in the source code, we analyzed their density in the code. We observed, for each system in the dataset, their absolute number, percentage of classes being affected by them, and how many agglomerations each KLOC has in average. This information is summarized in three box-plots for each agglomeration category, and their respective IQR and standard deviation were presented. We have found that in all systems and levels of granularity, the Heterogeneous Agglomeration is the most dense in the code. Their variation is also high for all plots and for the two measures. The category also suffers greatly from

outliers. This told us that the presence of this category is affected by other software characteristics beyond their size.

We then analyzed the impact of such agglomerations on five modularity metrics, evaluating the Cohen's d values and the Coefficient of Variation (CV). For the maxNest metric, all effects were Small. However, when observing the values, the Heterogeneous Agglomeration is slightly higher than the other categories. For the RFC metric, the Heterogeneous Agglomeration and Isolated Large Class had the highest effect for this metric. For the DIT metric, all effects were Small. However, the d values and CVs were higher for the Heterogeneous and Isolated Refused Bequest categories. Heterogeneous, Isolated Large Class and Homogeneous Long Method obtained the Large effects for the WMC metric. Finally, for the CBO metric, the category that obtained the highest d values was the Isolated Large Class. In contrast, Clean classes presents the lowest values for this metric. In conclusion, we could observe that Heterogeneous Agglomerations and Isolated Large Class affects the most the system modularity.

Since we have found that Heterogeneous Agglomerations does impact the source code modularity metrics, we identified the need of deep exploring such agglomerations. For this purpose, we used a similar methodology to verify if size influenced our results and if there is a type of Heterogeneous Agglomeration that is significant different from the others in terms of metric values. From the analysis of density, we could conclude that all categories had a similar behavior in terms of median and variation. The most concentrated category was Refused Bequest - Feature Envy, that was mitigated when considering other size measurements. In terms of impact on the source code modularity, we have found that each metric had a different set of Heterogeneous Agglomerations that mostly affects them. However, at least one of the Heterogeneous Agglomeration found by the association rules algorithm appeared as the most impacting category for all evaluated metrics.

We can conclude from our analysis that (i) it is important to consider the presence of bad smell agglomerations composed of the same smell type, the Homogeneous Agglomerations, since they can be frequent in the source code and may impact the modularity. (ii) The choice of using association rules algorithm was good, since all agglomerations found impacted the most the system modularity when analyzed separately, justifying its use. (iii) We provided exploratory evidences that, beyond being more concentrated in the source code, the further investigation of the occurrences of such agglomerations are indeed beneficial to practitioners.

We conclude this chapter by presenting some topics of investigation that appeared along this dissertation, but we could not fully address them. In answering them, they will contribute to reduce the gap that exists in terms of bad smell agglomerations.

- Expand the dataset through to addition of: (i) new open-source and industrial systems. (ii) evaluate other sets of bad smells, since in our analyzes we focus on only four of them. (iii) Evaluate how the agglomerations behave when other modularity metrics are considered.

- Mine the versions of each system in order to identify if classes containing bad smells agglomerations are more fault and change prone.

- Investigate the presence of agglomerations in different levels of granularity. In this work, we evaluated them in terms of classes. A similar analysis could be done to address smells at method level, for instance.

- Investigate the existence and impact of inter-component agglomerations, *i.e.*, agglomerations that are coupled to each other.

- Evaluate the spreadness of the agglomerations considering the package that they are located. This would allow to understand if there is a concentration of agglomerations on the source code.

- Evaluate how the system domain contributes to the existence of bad smell agglomerations. This suggestion is motivated by other studies [61] that verified that some domains contain more agglomerations than others.

- Investigate how automatic detection tools could support developers in identifying the most harmful agglomerations.

- Evaluate how machine learning approaches can help developers to automatically detect bad smell agglomerations.

- Study how the bad smell agglomerations behave in different system versions by analyzing: when they are created, if they are removed, or if their persist along the development cycle.

Appendix A

Original Values of Cohen's d Comparison

Table A.1: Cohen's d values for CBO

Agg. Type	Het.	Hom. FE	Hom. LM	Isol. LC	Isol. RB	Isol. FE	Isol. LM	Clean
Het.	-	0.5019	0.0922	-0.3212	0.7438	0.7677	0.2888	0.8322
Hom. FE	-0.5019	-	-0.4433	-0.7633	0.1750	0.2133	-0.3432	0.3040
Hom. LM	-0.0922	0.4433	-	-0.4150	0.7185	0.7445	0.2019	0.8161
Isol. LC	0.3212	0.7633	0.4150	-	0.9682	0.9867	0.6071	1.0373
Isol. RB	-0.7438	-0.1750	-0.7185	-0.9682	-	0.0741	-0.7915	0.2302
Isol. FE	-0.7677	-0.2133	-0.7445	-0.9867	-0.0741	-	-0.8182	0.1539
Isol. LM	-0.2888	0.3432	-0.2019	-0.6071	0.7915	0.8182	-	0.9083
Clean	-0.8322	-0.3040	-0.8161	-1.0373	-0.2302	-0.1539	-0.9083	-

Table A.2: Cohen's d values for WMC

Agg. Type	Het.	Hom. FE	Hom. LM	Isol. LC	Isol. RB	Isol. FE	Isol. LM	Clean
Het.	-	0.6334	0.5065	-0.4505	1.0158	0.9722	0.8504	1.0378
Hom. FE	-0.6334	-	-0.2322	-0.9220	0.4070	0.3460	0.1905	0.4554
Hom. LM	-0.5065	0.2322	-	-0.8380	0.8710	0.7836	0.5497	0.8817
Isol. LC	0.4505	0.9220	0.8380	-	1.1569	1.1306	1.0578	1.1759
Isol. RB	-1.0158	-0.4070	-0.8710	-1.1569	-	-0.1235	-0.3478	0.1365
Isol. FE	-0.9722	-0.3460	-0.7836	-1.1306	0.1235	-	-0.2418	0.2291
Isol. LM	-0.8504	-0.1905	-0.5497	-1.0578	0.3478	0.2418	-	0.4086
Clean	-1.0378	-0.4554	-0.8817	-1.1759	-0.1365	-0.2291	-0.4086	-

Table A.3: Cohen's d values for DIT

Agg. Type	Het.	Hom. FE	Hom. LM	Isol. LC	Isol. RB	Isol. FE	Isol. LM	Clean
Het.	-	0.2601	0.1015	0.2355	0.0514	0.3264	0.0026	0.3822
Hom. FE	-0.2601	-	-0.2068	-0.0361	-0.2637	0.0168	-0.3048	0.0778
Hom. LM	-0.1015	0.2068	-	0.1764	-0.0692	0.2911	-0.1280	0.3612
Isol. LC	-0.2355	0.0361	-0.1764	-	-0.2368	0.0616	-0.2808	0.1254
Isol. RB	-0.0514	0.2637	0.0692	0.2368	-	0.3720	-0.0641	0.4417
Isol. FE	-0.3264	-0.0168	-0.2911	-0.0616	-0.3720	-	-0.4187	0.0787
Isol. LM	-0.0026	0.3048	0.1280	0.2808	0.0641	0.4187	-	0.4849
Clean	-0.3822	-0.0778	-0.3612	-0.1254	-0.4417	-0.0787	-0.4849	-

Table A.4: Cohen's d values for RFC

Agg. Type	Het.	Hom. FE	Hom. LM	Isol. LC	Isol. RB	Isol. FE	Isol. LM	Clean
Het.	-	0.7404	0.7053	-0.3457	1.2110	1.0673	0.8103	1.2547
Hom. FE	-0.7404	-	-0.1408	-0.8859	0.6804	0.4405	0.0110	0.7545
Hom. LM	-0.7053	0.1408	-	-0.8562	1.0879	0.7664	0.2058	1.1221
Isol. LC	0.3457	0.8859	0.8562	-	1.1796	1.0878	0.9223	1.2153
Isol. RB	-1.2110	-0.6804	-1.0879	-1.1796	-	-0.3366	-0.9531	0.1613
Isol. FE	-1.0673	-0.4405	-0.7664	-1.0878	0.3366	-	-0.6063	0.4548
Isol. LM	-0.8103	-0.0110	-0.2058	-0.9223	0.9531	0.6063	-	0.9967
Clean	-1.2547	-0.7545	-1.1221	-1.2153	-0.1613	-0.4548	-0.9967	-

Table A.5: Cohen's d values for maxNest

Agg. Type	Het.	Hom. FE	Hom. LM	Isol. LC	Isol. RB	Isol. FE	Isol. LM	Clean
Het.	-	0.0682	0.2201	0.2387	0.3045	0.1499	0.3270	0.1545
Hom. FE	-0.0682	-	0.1583	0.1775	0.2528	0.0863	0.2791	0.0853
Hom. LM	-0.2201	-0.1583	-	0.0130	0.1258	-0.0642	0.1705	-0.0959
Isol. LC	0.2387	-0.1775	-0.0130	-	0.1490	-0.0789	0.2168	-0.1212
Isol. RB	-0.3045	-0.2528	-0.1258	-0.1490	-	-0.1586	0.0962	-0.2378
Isol. FE	-0.1499	-0.0863	0.0642	0.0789	0.1586	-	0.1871	-0.0131
Isol. LM	-0.3270	-0.2791	-0.1705	-0.2168	-0.0962	-0.1871	-	-0.2822
Clean	-0.1545	-0.0853	0.0959	0.1212	0.2378	0.0131	0.2822	-

Appendix B

Heterogeneous Cohen's d Values

Table B.1: Cohen's d values for CBO

Type	LcLmFe	RbLmFe	LcRbFe	LcRbLmFe	LcRbLm	FeLm	LcFe	LcLm	LcRb	RbFe	RbLm
LcLmFe	-	0.4791	0.1271	0.3247	0.0973	0.5665	0.0269	0.0625	0.0694	0.7356	0.4620
RbLmFe	-0.4791	-	0.5832	0.8606	0.7038	0.1885	0.2430	0.4429	0.4495	0.4734	0.0431
LcRbFe	-0.1271	-0.5832	-	0.1803	0.0483	0.6595	0.1147	0.1908	0.1990	0.8104	0.5685
LcRbLmFe	-0.3247	-0.8606	-0.1803	-	0.2604	0.9243	0.2489	0.4011	0.4132	1.1046	0.8484
LcRbLm	-0.0973	-0.7038	-0.0483	-0.2604	-	0.7810	0.0899	0.1733	0.1836	1.0155	0.6887
FeLm	-0.5665	-0.1885	-0.6595	-0.9243	- 0.7810	-	-0.3096	-0.5377	-0.5455	-0.1926	-0.2294
LcFe	-0.0269	-0.2430	-0.1147	-0.2489	-0.0899	0.3069	-	0.0130	0.0169	0.3847	0.2310
LcLm	-0.0625	-0.4429	-0.1908	-0.4011	-0.1733	0.5377	-0.0130	-	-0.062	0.7266	0.4242
LcRb	-0.0694	-0.4495	-0.1990	-0.4132	-0.1836	0.5455	-0.0169	0.0062	-	0.7435	0.4305
RbFe	-0.7356	-0.4734	-0.8104	-1.1046	-1.0155	0.1926	-0.3847	-0.7266	-0.7435	-	0.5414
RbLm	-0.4620	-0.0431	-0.5685	-0.8484	-0.6887	0.2294	-0.2310	-0.4242	-0.4305	-0.5414	-

Table B.2: Cohen's d values for WMC

Type	LcLmFe	RbLmFe	LcRbFe	LcRbLmFe	LcRbLm	FeLm	LcFe	LcLm	LcRb	RbFe	RbLm
LcLmFe	-	1.2510	0.1422	0.0756	0.2027	1.2930	0.1099	0.6412	0.4761	1.3993	1.3466
RbLmFe	-1.2510	-	1.3174	1.4792	1.0024	0.1688	1.3864	1.4917	1.5348	0.3740	0.2375
LcRbFe	-0.1422	-1.3174	-	0.0787	0.0800	1.3577	0.0377	0.5698	0.3698	1.5084	1.4383
LcRbLmFe	-0.0756	-1.4792	-0.0787	-	0.1517	1.5127	0.0408	0.6982	0.4827	1.6847	1.6079
LcRbLm	-0.2027	-1.0024	-0.800	-0.1517	-	1.0511	0.1142	0.3801	0.2205	1.1507	1.0987
FeLm	-1.2930	-0.1688	-1.3577	-1.5127	-1.0511	-	-1.4240	-1.4631	-1.5319	-0.1248	-0.0258
LcFe	-0.1099	-1.3864	-0.0377	-0.0408	-0.1142	1.4240	-	0.6274	0.4216	1.5826	1.5100
LcLm	-0.6412	-1.4917	-0.5698	-0.6982	-0.3801	1.4631	-0.6274	-	-0.2923	2.0148	1.7852
LcRb	-0.4761	-1.5348	-0.3698	-0.4827	-0.2205	1.5319	-0.4216	0.2923	-	1.9268	1.7641
RbFe	-1.3993	-0.3740	-1.5084	-1.6847	-1.1507	0.1248	-1.5826	-2.0148	-1.9268	-	0.1263
RbLm	-1.3466	-0.2375	-1.4383	-1.6079	-1.0987	0.0258	-1.5100	-1.7852	-1.7641	0.1263	-

Table B.3: Cohen's d values for DIT

Type	LcLmFe	RbLmFe	LcRbFe	LcRbLmFe	LcRbLm	FeLm	LcFe	LcLm	LcRb	RbFe	RbLm
LcLmFe	-	0.7401	0.4335	0.4866	0.6445	0.1754	0.0631	0.1851	0.6747	0.5771	0.7505
RbLmFe	-0.7401	-	0.2562	0.4197	0.2863	0.1595	0.6590	0.3950	0.2081	0.2730	0.1213
LcRbFe	-0.4335	-0.2562	-	0.0724	0.0561	0.0355	0.3653	0.1724	0.1106	0.0407	0.1856
LcRbLmFe	-0.4866	-0.4197	-0.0724	-	0.2126	0.0068	0.3938	0.1445	0.2783	0.1608	0.3880
LcRbLm	-0.6445	-0.2863	-0.0561	-0.2126	-	0.0608	0.5430	0.2489	0.0924	0.0176	0.2122
FeLm	-0.1754	-0.1595	-0.0355	-0.0068	-0.0608	-	-0.1451	-0.0640	-0.0846	-0.0549	-0.1173
LcFe	-0.0631	-0.6590	-0.3653	-0.3938	-0.5430	0.1451	-	0.1321	0.5779	0.4874	0.6524
LcLm	-0.1851	-0.3950	-0.1724	-0.1445	-0.2489	0.0640	-0.1321	-	-0.2876	0.2281	0.3466
LcRb	-0.6747	-0.2081	-0.1106	-0.2783	-0.0924	0.0846	-0.5779	0.2876	-	0.0943	0.1110
RbFe	-0.5771	-0.2730	-0.0407	-0.1608	-0.0176	0.0549	-0.4874	-0.2281	-0.0943	-	0.1965
RbLm	-0.7505	-0.1213	-0.1856	-0.3880	-0.2122	0.1173	-0.6524	-0.3466	-0.1110	-0.1965	-

Table B.4: Cohen's d values for RFC

Type	LcLmFe	RbLmFe	LcRbFe	LcRbLmFe	LcRbLm	FeLm	LcFe	LcLm	LcRb	RbFe	RbLm
LcLmFe	-	0.9757	0.0642	0.1404	0.0958	1.2238	0.1701	0.6525	0.4559	1.3965	1.2868
RbLmFe	-0.9757	-	0.8223	1.2040	0.9482	0.3720	0.7019	0.4832	0.4931	0.6298	0.4308
LcRbFe	-0.0642	-0.8223	-	0.1989	0.0229	1.0474	0.1005	0.5255	0.3616	1.1967	1.0932
LcRbLmFe	-0.1404	-1.2040	-0.1989	-	0.2484	1.4700	0.3087	0.8622	0.6229	1.6648	1.5535
LcRbLm	-0.0958	-0.9482	-0.0229	-0.2484	-	1.2236	0.0880	0.5925	0.3839	1.4214	1.3014
FeLm	-1.2238	-0.3720	-1.0474	-1.4700	-1.2236	-	-0.9279	-0.8653	-0.7647	-0.2319	-0.0050
LcFe	-0.1701	-0.7019	-0.1005	-0.3087	-0.0880	0.9279	-	0.4028	0.2524	1.0756	0.9698
LcLm	-0.6525	-0.4832	-0.5255	-0.8622	-0.5925	0.8653	-0.4028	-	-0.1361	1.1621	0.9874
LcRb	-0.4559	-0.4931	-0.3616	-0.6229	-0.3839	0.7647	-0.2524	0.1361	-	0.9465	0.8168
RbFe	-1.3965	-0.6298	-1.1967	-1.6648	-1.4214	0.2319	-1.0756	-1.1621	-0.9465	-	0.2769
RbLm	-1.2868	-0.4308	-1.0932	-1.5535	-1.3014	0.0050	-0.9698	-0.9874	-0.8168	-0.2769	-

Table B.5: Cohen's d values for maxNest

Type	LcLmFe	RbLmFe	LcRbFe	LcRbLmFe	LcRbLm	FeLm	LcFe	LcLm	LcRb	RbFe	RbLm
LcLmFe	-	0.3056	0.1647	0.2444	0.1667	0.4120	0.2318	0.1622	0.1212	0.3292	0.4336
RbLmFe	-0.3056	-	0.3855	0.0718	0.1858	0.1673	0.4850	0.2206	0.2081	0.0270	0.2089
LcRbFe	-0.1647	-0.3855	-	0.3461	0.2937	0.4499	0.0452	0.2915	0.2601	0.3997	0.4612
LcRbLmFe	-0.2444	-0.0718	-0.3461	-	0.1047	0.2226	0.4388	0.1280	0.1371	0.0996	0.2571
LcRbLm	-0.1667	-0.1858	-0.2937	-0.1047	-	0.3500	0.3812	0.0151	0.0443	0.2197	0.3912
FeLm	-0.4120	-0.1673	-0.4499	-0.2226	-0.3500	-	-0.5600	-0.4174	-0.3418	-0.1553	-0.0401
LcFe	-0.2318	-0.4850	-0.0452	-0.4388	-0.3812	0.5600	-	0.3809	0.3407	0.5023	0.5742
LcLm	-0.1622	-0.2206	-0.2915	-0.1280	-0.0151	0.4174	-0.3809	-	-0.0337	0.2619	0.4748
LcRb	-0.1212	-0.2081	-0.2601	-0.1371	-0.0443	0.3418	-0.3407	0.0337	-	0.2363	0.3713
RbFe	-0.3292	-0.0270	-0.3997	-0.0996	-0.2197	0.1553	-0.5023	-0.2619	-0.2363	-	0.2060
RbLm	-0.4336	-0.2089	-0.4612	-0.2571	-0.3912	0.0401	-0.5742	-0.4748	-0.3713	-0.2060	-

Bibliography

[1] 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 *15th European Conference on Software Maintenance and Reengineering*, pages 181–190, 2011.

[2] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 207–216, 1993.

[3] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini. Do design metrics capture developers perception of quality? an empirical study on self-affirmed refactoring activities. *arXiv preprint arXiv:1907.04797*, 2019.

[4] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[5] T. Besker, A. Martini, and J. Bosch. The pricey bill of technical debt: When and by whom will it be paid? In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 13–23, 2017.

[6] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 47–52, 2010.

[7] A. Chatzigeorgiou and A. Manakos. Investigating the evolution of bad smells in object-oriented code. In *Seventh International Conference on the Quality of Information and Communications Technology*, pages 106–115, 2010.

[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[9] J. Cohen. *Statistical power analysis for the behavioral sciences*. Academic Press, 2013.

[10] W. Cunningham. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, 1992.

[11] 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 Transaction on Software Engineering*, 2018.

[12] L. da S. Carvalho, R. Novais, and M. Mendonça. Investigating the relationship between code smell agglomerations and architectural concerns: Similarities and dissimilarities from distributed, service-oriented, and mobile systems. In *Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse*, page 3–12, New York, NY, USA, 2018. Ass. for Computing Machinery.

[13] L. da S. Sousa. Spotting design problems with smell agglomerations. In *IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 863–866, 2016.

[14] D. Di Nucci, F. Palomba, D. Tamburri, A. Serebrenik, and A. De Lucia. Detecting code smells using machine learning techniques: are we there yet? In *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621. IEEE, 2018.

[15] 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*, New York, NY, USA, 2016. Ass. for Computing Machinery.

[16] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

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

[18] F. Fontana, P. Braione, and M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11:5: 1–38, 2012.

[19] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. In

IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, pages 609–613, 2016.

[20] F. A. Fontana, V. Ferme, and S. Spinelli. Investigating the impact of code smells debt on quality code evaluation. In *Third International Workshop on Managing Technical Debt (MTD)*, pages 15–22, 2012.

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

[22] J. Garcia, D. Popescu, Ge. 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*, page 146–162, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] 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*, 23(4), 2014.

[24] J. Han and M. Kamber. Data mining: Concepts and techniques. *Morgan Kaufmann Publishers*, 2, 2001.

[25] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley, 1990.

[26] J. Kerievsky. *Refactoring to Patterns.* Pearson Higher Education, 2004.

[27] F. Khomh, M. Di Penta, YG. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[28] J. Landis and G. Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

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

[30] M. M. Lehman and L. A. Belady. *Software Evolution - Processes of Software Change.* Academic Press London, 1985.

[31] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[32] B. Liskov and J. M. Wing. Family values: A behavioral notion of subtyping. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1993.

[33] B. Liu, W. Hsu, and Y. Ma. Pruning and summarizing the discovered associations. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 125–134, New York, NY, USA, 1999. Association for Computing Machinery.

[34] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, Inc., USA, 1994.

[35] A. Lozano, K. Mens, and J. Portugal. Analyzing code evolution to uncover relations between bad smells. *IEEE 2nd International Workshop on Patterns Promotion and Anti-Patterns Prevention*, 03 2015.

[36] M. V. Mantyla, J. Vanhanen, and C. Lassenius. Bad smells - humans as code critics. In *20th IEEE International Conference on Software Maintenance*, pages 399–408, 2004.

[37] N. Moha and Y.G. Guéhéneuc. Decor: a tool for the detection of design defects. In *Proceedings IEEE/ACM International Conference on Automated Software Engineering*, pages 527–528, 2007.

[38] W. Oizumi, A. Garcia, T. Colanzi, M. Ferreira, and A. Staa. On the relationship of code-anomaly agglomerations and architectural problems. *Journal of Software Engineering Research and Development*, 3(1):11, 2015.

[39] 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 *IEEE/ACM 38th International Conference on Software Engigeering (ICSE)*, pages 440–451, 2016.

[40] W. Oizumi, L. Sousa, A. Oliveira, A. Garcia, A. Agbachi, R. Oliveira, and C. Lucena. On the identification of design problems in stinky code: experiences and tool support. *Journal of the Brazilian Computer Society*, 24(1):13, 2018.

[41] W. Oizumi, L. Sousa, A. Oliveira, A. Garcia, A. B. Agbachi, R. Oliveira, and C. Lucena. On the identification of design problems in stinky code: experiences and tool support. *Journal of the Brazilian Computer Society*, 24(1):13, 2018.

[42] W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. v. Staa. When code-anomaly agglomerations represent architectural problems? an exploratory study. In *Brazilian Symposium on Software Engineering*, pages 91–100, 2014.

[43] 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 *IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[44] T. Paiva, A. Damasceno, J. Padilha, E. Figueiredo, and C. Sant'Anna. Experimental evaluation of code smell detection tools. *Workshop on Software Visualization, Evolution, and Maintenance (VEM)*, 2015.

[45] 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*, 99:1–10, 2018.

[46] 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 *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 482–482, 2018.

[47] 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 *IEEE International Conference on Software Maintenance and Evolution*, pages 101–110, 2014.

[48] F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Landfill: An open dataset of code smells with public evaluation. In *IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 482–485, 2015.

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

[50] B. Pietrzak and B. Walter. Leveraging code smell detection with inter-smell relations. In *International Conference on Extreme Programming and Agile Processes in Software Engineering*, pages 75–84. Springer, 2006.

[51] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw. Building empirical support for automated code smell detection. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, 2010.

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

[53] 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*, 39(8):1144–1156, 2013.

[54] D. Taibi, A. Janes, and V. Lenarduzzi. How developers perceive smells in source code: A replicated study. *Information and Software Technology*, 92:223–235, 2017.

[55] P.N. Tan, V. Kumar, and J. Srivastava. Selecting the right objective measure for association analysis. *Information Systems*, 29(4):293–313, 2004.

[56] 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 *Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, 2010.

[57] S. Tufféry. *Data mining and statistics for decision making*. John Wiley & Sons, 2011.

[58] J. Van Gurp and J. Bosch. Design erosion: problems and causes. *Journal of systems and software*, 61(2):105–119, 2002.

[59] S. Vidal, W. Oizumi, A. Garcia, A. Pace, and C. Marcos. Ranking architecturally critical agglomerations of code smells. *Science of Computer Programming*, 182:64–85, 2019.

[60] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi. Jspirit: a flexible tool for the analysis of code smells. In *34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6, 2015.

[61] B. Walter, F. A. Fontana, and V. Ferme. Code smells and their collocations: A large-scale experiment on open-source systems. *Journal of Systems and Software*, 144:1–21, 2018.

[62] T. Wu, Y. Chen, and J. Han. Re-examination of interestingness measures in pattern mining: a unified framework. *Data Mining and Knowledge Discovery*, 21(3):371–397, 2010.

[63] 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*, 44(10):951–976, 2018.

[64] A. Yamashita and S. Counsell. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software*, 86(10):2639–2653, 2013.

[65] A. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251, 2013.

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

[67] 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 *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 121–130, Sep. 2015.