information

MDPI

Article

An Empirical Study on the Evolution of Design Smells

Lerina Aversano *, Umberto Carpenito and Martina Iammarino

Department of Engineering, University of Sannio, 82100 Benevento, Italy; umbertocarpenito@gmail.com (U.C.); iammarino@unisannio.it (M.I.)

* Correspondence: aversano@unisannio.it

Received: 30 April 2020; Accepted: 30 June 2020; Published: 4 July 2020

Abstract: The evolution of software systems often leads to its architectural degradation due to the presence of design problems. In the literature, design smells have been defined as indicators of such problems. In particular, the presence of design smells could indicate the use of constructs that are harmful to system maintenance activities. In this work, an investigation on the nature and presence of design smells has been performed. An empirical study has been conducted considering the complete history of eight software systems, commit by commit. The detection of instances of multiple design smell types has been performed at each commit, and the analysis of the relationships between the detected smells and the maintenance activities, specifically due to refactoring activities, has been investigated. The proposed study evidenced that classes affected by design smells are more subject to change, especially when multiple smells are detected in the same classes. Moreover, it emerged that in some cases these smells are removed, and this occurs involving more smells at the same time. Finally, results indicate that smells removals are not correlated to the refactoring activities.

Keywords: technical debt; design smells; software maintenance; software evolution

1. Introduction

A usually discussed issue in software evolution is the tradeoff between short-term and long-term goals. Very often, the need to fast delivery to stakeholders can lead to design problems and rework costs. Specifically, design violations taken to deliver fast might delay future development, compromising software maintainability, and evolution.

In the literature, there are numerous researches about design problems and their impact on software project evolution, highlighting that these problems increase the cost of change over time until software becomes almost un-maintainable. Design smells have been defined as indicators of such problems. In particular, the presence of design smells could indicate the use of constructs that are harmful to system maintenance activities. Architectural smells are ultimately instances of poor design decisions [1,2], at the architectural level.

Design and architectural smells negatively affect system life cycle properties, such as understandability, testability, extensibility, and re-usability [3]. Indeed, design smells are closely related to system evolution, as often an organization necessarily deals with accumulated design problems when adding new features to a software system. Unmanaged, design smells can lead to significant technical problems and increased maintenance and evolution efforts.

More specifically, during the evolution of the software, the quantity and complexity of the interactions between the software elements increase, with a consequent effect on the project structure.

This paper proposes an empirical study conducted on the evolution history of eight software systems, analyzing 17,252 commits.

In particular, by focusing on addiction-related smells, an investigation was conducted into the evolution of design smells, through the detection of instances of multiple types of design smells.

Among all the smells defined in the literature [4], the focus of this article is on abstraction design smells, encapsulation design smells, modularization design smells, and hierarchy design smells.

The impacts of these smells on the refactoring number and the relative modifications carried out on a software system were studied.

The identification of refactoring activities has been used to see if this brings an improvement in the design of existing source code.

The empirical study conducted confirmed that the classes affected by design smells are more subject to changes, highlighting that especially when more smells are detected in the same classes, these are more frequently subject to changes.

This result may be due to the fact that, most smells, are introduced once the affected class is transferred to the repository for the first time. In addition, it emerged that the removal of design smells occurs simultaneously for multiple smells.

Finally, the results showed that the removal of smells is not related to the presence of refactoring.

The rest of the paper is organized as follow: in Section 2, some background material on software design, open source software development, and mining software repositories are provided, Section 3 discusses related works in the literature, Section 4 describes empirical study design, while the results of the study are reported in Section 5, Section 6 discusses the threats that could affect the obtained results, finally, conclusions are given in the last section.

2. Background

2.1. Software Design

Thanks to the numerous studies carried out, software design turns out to be a critical problem, capable of strongly influencing the quality of the software. The presence of design problems and incorrect constructions can negatively affect the main features of the software, such as the comprehensibility, testability, extensibility, reusability, and maintainability of the software [5].

A very useful indicator of design problems is knowledge of smells [6].

In literature, based on the extent of the impact caused by smell, smells can be divided into three main categories: code smell, architectural smell, and design smell.

Architectural smells and code smells are indicators of bad code or design that can lead to quality problems, such as breakdowns, technical debt, or difficulties in maintenance and evolution.

Smell codes usually have a limited local impact, because they involve a class or a file and require only simple refactoring activities. On the other hand, however, the architectural smells concern the system level and represent the violation of the design principles or of the decisions that affect the internal qualities of the software resulting in negative effects on maintenance and evolution costs.

Design smells indicate design aspects that violate fundamental principles and negatively affect software design. Furthermore, it is not only their knowledge that is fundamental but also their management, because being related to the evolution of the system, the smell of the incorrectly managed design leads to significant technical problems and increases maintenance and evolution efforts by developers.

In particular, this study focuses on design smells, categorized into four groups on the base of their similarity: abstraction, encapsulation, modularization, and hierarchy.

It has been chosen to focus attention on design smells because in the literature there are numerous studies on other types of smells, instead, some issues related to design smells remain unsolved.

More specifically, this study considered: the imperative abstraction smell that arises when an operation is transformed into a class; unnecessary abstraction is a bad smell that introduces an abstraction that is not necessary for software design; unutilized abstraction is an abstraction that is not used, finally, a multifaceted abstraction happens when the elements of the abstraction are not cohesive.

A deficient encapsulation smell occurs when the declared accessibility of one or more members of abstraction is more permissive than required, and an unexploited encapsulation smell occurs when the client class relies on the use of explicit controls, instead of taking advantage of the variation of the types already encapsulated in a hierarchy.

Broken modularization refers to the presence of separate and diffused data and/or methods among multiple abstractions, which instead should have been localized in a single abstraction; insufficient modularization refers to a large or complex abstraction, which could be further modular; the hub-like modularization smell occurs when an abstraction has dependencies with a large number of other abstractions, and cyclic dependent modularization is when two or more abstractions create a close coupling between the abstractions, depending on each other directly or indirectly.

Wide hierarchy is a smell that occurs when intermediate types are missing because a hereditary hierarchy is too large; broken hierarchy occurs when a supertype and its subtype conceptually do not share an "IS-A" relationship resulting in interrupted substitutability; deep hierarchy arises when there is an excessively deep hereditary hierarchy; multipath hierarchy refers to the presence of a subtype that inherits from a supertype both directly and indirectly from a supertype which leads to unnecessary inheritance paths in the hierarchy; cyclic hierarchy occurs when a supertype in a hierarchy depends on any of its subtypes; rebellious hierarchy is a smell that arises when a subtype rejects the methods provided by its supertypes, instead missing hierarchy arises when conditional logic is used by a design segment to explicitly manage the variation in the behavior in which a hierarchy could have been created and used to encapsulate those variations.

As previously mentioned, another fundamental aspect of software design concern refactoring, whose idea is to recognize that it is difficult to make good code and good design from the beginning and, as the requirements change, the design must be changed.

Refactoring, originally introduced by Martin Fowler, is generally motivated by the detection of a code smell, for example, a method may appear excessively long and complex, or contain a lot of duplicate code in another method [4].

Refactoring, therefore, provides techniques for evolving design in small steps. The principles of refactoring are based on changing the internal structure of the software system to make it easier and understandable, without changing its observable behavior.

It is applied to improve some non-functional features of the software such as readability, maintainability, reusability, extensibility of the code as well as the reduction of its complexity, possibly through the subsequent introduction of design patterns.

2.2. Open Source Software Development

Open-source software development is characterized by the public availability of its source code.

In 1997, Eric S. Raymond [7] made the distinction between conventional closed source and open source development. According to Raymond, the former has centralized planning with a rigorous organization and a process from start to finish. The second is based on different approaches and organizations that favor a more fluid development process, characterized by greater collaboration within the team, continuous integration and testing, and greater involvement of the end-user.

Successful open source communities have developed processes where the code can be sent and integrated asynchronously, communication is well documented, and features are integrated in small increments to detect problems early in the development cycle.

According to Bar and Fogel [8], in the development of closed source software, programmers often spend a lot of time managing and creating bug reports and spend a lot of time on development plans. Instead, in developing open-source software, these problems are solved by integrating software users into the development process or even letting these users build the system themselves.

The development of open-source software can be divided into several phases defined by Sharma et al. [9]: initiation, execution, release.

In Open Source Software development, the participants are distributed among different geographic regions, therefore tools are needed to help the participants to collaborate in the development of the source code. For this reason, concurrent versions systems (CVS) are used to help manage the files and codes of a project when several people are working on the project at the same time. CVS allows multiple people to work on the same file simultaneously.

In particular, the systems that were considered for this work were available on the Github platform, a hosting service for software projects.

2.3. Mining Software Repositories

It consists of using data mining techniques to analyze data in software repositories to extract information produced by developers during the development process, which is useful and usable [10].

When mining software repositories, the extracted data can be useful to discover hidden models and trends, support development activities, maintain existing systems, or improve the decision-making process related to the development and future evolution of the software.

Generally, the data is used to produce quality software systems, and improve their management. The different types of software repositories include source code control repositories, bug repositories, and code repositories.

The mining software repository is based on the use of tools that extract data from projects.

3. Related Works

The impact of code level smells defined by Fowler [4] has been widely investigated in the research literature. In particular, several studies analyzed their effects on maintainability [11,12], program comprehension [13], change and fault-proneness [14,15]. Currently, there are also several tools used to automatically detect [16,17] code smells, exploiting different sources of information. Several papers discuss code smells fix through the application of refactoring operations [18,19].

At architectural and design levels smells are ultimately instances of poor design decisions [2]. The presence of construct design problems contributes to the system erosion [1,2]. These smells have a negative impact on system life cycle properties, such as understandability, testability, extensibility, and re-usability [3].

Several research papers in the literature deal with the detection of architectural smells, and part also with the influence of architectural smell on issue related activities.

The work of Le et al. [20] presented one of the largest empirical studies on architectural changes in long-lasting software systems, based on the analysis of 14 systems at the version level, thanks to the use of ARCADE, a software workbench that allows detecting different aspects of architectural change.

These results show that the versioning scheme of a system is not an accurate indicator of architectural change and that the architecture of a system may be relatively unstable in view of a release.

Fontana et al. [21] conducted a large-scale empirical study based on the investigation of the correlations between code smells and architecture smells. Specifically, they investigated 102 Java projects, using the SonarQube plug-in "Antipatterns-CodeSmells" for code odors and Arcan for the architecture of code smells, with the aim of understanding whether the architecture smells are independent of smell code or derive from each other. Their results show that there is no correlation between the two categories of smell because the presence of code smells does not directly imply the presence of architectural smells and vice versa.

In [22], it is described a system-level multiple refactoring algorithm, which can identify the move method, move field, and extract class refactoring opportunities automatically according to the principle of high cohesion and low coupling. The algorithm works by merging and splitting related classes to obtain the optimal functionality distribution from the system-level.

Brunet et al. [23] studied the evolution of architectural violations in 76 versions selected from four subject systems showing like the number of architectural violations is constantly growing over

time, some previously identified violations reappear, and in all the systems studied a critical core is identified and this core does not change over time.

Arcan [24] is a static analysis tool targeted at the detection of three architectural smells, including cycles and hubs. Arcan creates a graph database containing the structural dependencies of a Java system and then runs several detection algorithms (one per smell) on this graph. At last, there are some commercial tools for detecting architectural smells, such as Designite [25], which identifies seven architecture smells, including cycles and other dependency-based smells. As far as we are aware, all the previous tools have no predictive capabilities.

Pietrzak and Walter [26] analyzed different relations that exist among smells and provide tips on how they could be exploited to improve the detection of other smells.

Numerous research papers empirically evaluated the effects related to the presence of smells. Le et al. [17] presented an empirical study to date of architectural decay and its impact on software systems. For each version of the system, different architectural recovery techniques have been applied considering different types of smells. They examined the relationships between the smells collected and about 42.00 issues extracted from the repositories of the various systems in question. This has shown how architectural decay can cause significant problems for each software system. Mo et al. [27] presented an empirical study of hotspot patterns that cause high maintenance costs. The aim of the study shows that these patterns not only identify the most error-prone and change-prone files, but also the root causes of bug-proneness and change-proneness in specific architecture problems. Tufano et al. [12] presented an empirical investigation into when and why code smells are introduced in software projects. The study conducted over the commit history of open source projects demonstrated that most of the time the smells identified since their creation.

In [15], the authors analyzed the magnitude and effects of smell co-occurrence, i.e., the co-occurrence of different types of smells on the same code component. They observed that some code smells frequently co-occur and that method-level code smells may be the root cause for the introduction of class-level smells.

In [28], it is empirically validated the frequent collocations of 14 code smells detected in numerous Java open source systems. The authors highlighted as smell collocations lead to recurring patterns that could help to prioritize the classes to be refactored.

The presence of code smells has been investigated also in industrial projects [29]. The study reports an empirical evaluation of inter-smell relations by analyzing larger systems, and by including both industrial and open-source ones. Palomba et al. [15] reported a large-scale empirical investigation on the diffuseness of code smells and their impact on code change- and fault-proneness, showing as smells characterized by long and/or complex code are highly diffused, and that smelly classes have a higher change- and fault-proneness than smell-free classes. Table 1 summarizes the main differences that emerged from the literature analysis.

Table 1. Related work analysis.

	Revision Commit Level	Design Code Smells	Refactoring Analysis	Issues Analysis
Fontana et al. [21]	Revision	Code	No	No
Brunet et al. [23]	Revision	Code	No	Yes
Walter et al. [28]	Revision	Code	No	No
Yamashita et al. [29]	Revision	Code	No	No
Palomba et al. [15]	Revision	Code	Yes	No
Tufano et al. [12]	Commit	Code	Yes	No
Proposed study	Commit	Design	Yes	Yes

All of these studies were conducted at a significantly high-level scope than our work, comparing the different releases of the analyzed software systems. The main contributes respect to the existing literature is:

- A fine-grained analysis, at commits level, that is, the smells detection and the refactoring detection have been performed for each commit;
- Focus on design smells, indeed, the large part of existing papers deals with the analysis of code smells;
- Analysis of design smells removal and addition for each commit;
- Relationships with refactoring activities and the presence of issues.

4. Empirical Study Design

Three research questions, concerning different aspects of the design smells evolution in a software system, are formulated for being addressed in the presented study. As previously stated, the study focuses on the presence of different types of smells in software design and the related evolution of the software components. Thus, the goal of the study is to analyze the evolution of design smells in open source software projects, to understand how they are related to maintenance activities, and specifically to refactoring and the presence of issues. Then, the study analyzes all the history of the software system considered, commit by commit, and, in particular, the research questions are the following:

4.1. Context and Research Questions

The study aims at addressing the following research questions:

RQ1—To what extent are design smells subject to change?

This question aims at investigating how differently design smells affect the evolution of a software system, in terms of changes performed on classes. This research question is useful to understand which design smells lead to a higher frequency of change for the impacted classes. Moreover, an investigation on the co-occurrences of smells in classes has been performed to understand their impact. For answering this question, a quantitative investigation, supported by graphical representations concerning bar charts graphics showing the changes of the analyzed software system, has been performed.

RQ2—To what extent are design smells removed during the evolution of the system?

This question aims to analyze the condition occurring when the design smells are removed. In particular, the aim is to investigate if some smells are removed together. This analysis aims to point out if specific maintenance activities are performed to reduce the number of smells affecting a software system. Descriptive statistics are used to support this analysis.

RQ3—To what extent can smell removal be related to refactoring actions?

This question aims to investigate the relationship between smell removals and refactoring. The purpose is to examine the removal of the smell in the software system considering its complete evolution history. To this aim, a quantitative investigation is described in the results section. The investigation is supported by tables with the percentage, and results of the statistical test.

The study looks for empirical evidence to support the well know hypothesis about the negative impact of design smells on the evolution of software systems. Specifically, the focus is on the relationships between detected design smells and the refactoring actions performed on the software system.

4.2. Data Extraction and Analysis Methodology

The proposed study reports results involving eight Java open-source projects. These systems have been considered since these systems satisfy the following criteria: the programming language is

Java; the Git repository is active and there is more than one release, there are variations in application domains, sizes, revisions, and the systems are also used by other studies.

Specifically, Log4j is a Java library used for managing logs in the Java environment; JavaAssist is a Java library used to modify Java bytecode allowing you to change the implementation of a class at run time; Guice is an open-source software framework that provides dependency injection support using annotations to configure Java objects; Junit4 is a framework used for writing repeatable tests; Atlas is a framework consisting of an extensible set of basic government services that allow companies to operate effectively thus allowing integration with the entire ecosystem of corporate data; Commons Digester allows the configuration of XML modules; Zookeeper provides an open-source distributed configuration service, a synchronization service for large distributed systems; finally Commons Net is a library that allows the implementation of the client-side of many basic Internet protocols, guaranteeing access to the fundamental protocol.

Table 2 shows the characteristics of the selected projects that are, the number of the total commits analyzed for projects, and the first and last commit date.

Table 2. Systems analyzed.

System	Commit	First-Last Commit Date
Log4j	3275	14 December 2020 | 4 June 2015
JavaAssist	888	22 April 2003 | 15 April 2019
Guice	1725	25 August 2006 | 4 June 2019
JUnit4	2397	22 April 2003 | 16 April 2019
Atlas	2766	7 December 2014 | 5 July 2019
Commons-digester	2145	3 May 2001 | 25 August 2017
Zookeeper	1939	3 November 2007 | 17 July 2019
CommonsNet	2117	3 April 2002 | 9 July 2019

Figure 1 depicts the toolchain used for gathering the data required for conducting the analysis. The first step was the extraction of the change history of the software systems considering each commit. Specifically, all the commits log messages have been analyzed and parsed to extract all the changes performed on the files of the code repository. All the data obtained have been organized and stored in a data set.

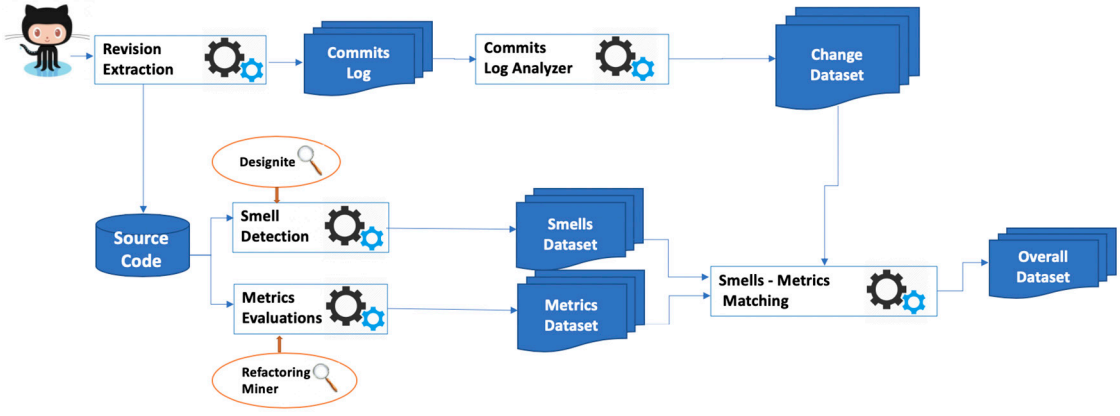

Figure 1. Toolchain used for the gathering the required data.

At the same time, the source code at each commit has been downloaded and analyzed for understanding its evolution over time, commits per commits.

This means that, at each commit of the system, the source code has been analyzed as follows:

- Detection of the design smells—to this aim the Designite tool has been used; Designite is a software design quality assessment tool. It supports comprehensive design smells detection but

also provides a detailed metrics analysis. Further, it offers various features to help identify issues contributing to design debt and improve the design quality of the analyzed software system. Table 3 reports the design smells detectable by Designite [19];

- Detection refactoring actions—to this aim the Refactoring Miner tool has been used, in particular, version 1.0; Refactoring Miner [30] is an open-source tool that classifies the different refactorings in the history of Java projects. Refactoring Miner takes as an input the list of commits and returns a list of refactoring operations applied between consecutive commits. Refactoring Miner can detect 15 types of refactorings from different types of code elements (see Table 4).

Table 3. Design smells detected.

	Type
Abstraction Design Smells	Imperative Abstraction, Unnecessary Abstraction, Multifaceted Abstraction, Unutilized Abstraction
Encapsulation Design Smells	Deficient Encapsulation, Unexploited Encapsulation
Modularization Design Smells	Broken Modularization, Insufficient Modularization, Hub-Like Modularization, Cyclically Dependent Modularization
Hierarchy Design Smells	Wide Hierarchy, Deep Hierarchy, Multipath Hierarchy, Cyclic Hierarchy, Rebellious Hierarchy, Missing Hierarchy

Table 4. Refactorings detected by Refactoring Miner.

Level	**Type**
Package	Change Package (Move, Rename, Split)
Type	Move Class, Rename Class, Extract Superclass/Interface
Method	Extract Method, Inline Method, Rename Method, Move Method, Pull Up Method, Push Down Method, Extract and Move Method
Field	Pull Up Field, Push Down Field, Move Field

The output of these two steps is made up of two datasets. The first contains all the information obtained from the Designite tool, the name of the project analyzed, the commit id, the author, the date of the commit, the file, and the type of design smell identified.

The second, on the other hand, contains all the information about refactoring obtained by using the Refactoring Miner tool. In particular, the name of the project, the commit id, the analyzed file, the presence or absence of refactoring activities, and possibly the type of refactoring found.

As shown in Figure 1, these datasets have been merged, obtaining an integrated dataset, containing all the data required for analysis. The obtained dataset contains for each commit, and for each file of that commit analyzed, all the information on the smells and refactoring previously collected. To check if there was an addition or removal of smell in that specific commit file, the algorithm compares the information of that file in the commit before the one under consideration. If one or more smells were present in the previous commit, and in the next one these smells are no longer present then in the dataset, than it is labelled as removed; if on the contrary, if the previous commit the smells detected by the tool was not present, than it is labelled as added; finally, if the information relating to a file between two successive commits does not change, it is labelled as no change.

The software systems analyzed, the tools, and the data collected are available for replication in our online appendix (https://drive.google.com/drive/folders/1oka_i5KFm1gHRwZHudu-etraVUb1hKbj?usp=sharing).

Moreover, a qualitative analysis was carried out to carry out a more in-depth investigation to understand if the refactoring actions contribute to the removal of the smells.

In detail, considering all the systems, starting from the initial dataset, a significant sample of files (80) was selected, in which there was the co-occurrence between the removal of smell and the presence of refactoring. Manually, to understand if there was an explicit reference to the refactoring actions,

each file was inspected by two authors, who assessed the presence of refactoring where the removal had occurred, by analyzing the text of the commit message, and inspection of the source code. Just in the 13% of cases, the authors agreed on the presence of the refactoring action to support the removal of the smell, thus determining a Cohen's k inter-rater is equal to 0.7, which is a strong Cohen agreement. Subsequently, the authors resolved the discrepancies, discussing cases on which they had not agreed.

5. Results

This section reports the analysis of the results of our experiment in the face of the question raised during the experimental design. The main goal of this study is to better understand what happens to the refactoring related activities of a system if design smells are detected. Then, the first step is to quantify the number of design smells identified in the various classes and how many of these are subject to change (RQ1). Therefore, to understand how many of these smells are removed during the evolution of the system (RQ2) and what relationships can have the refactoring actions in this change (RQ3).

5.1. RQ1—To What Extent Are Design Smells Subject to Change?

Figure 2 reports for each system, respectively for Log4j, JavaAssist, Guice, JUnit4, Atlas, Commons-Digester, Zookeeper, and Commons-Net the percentage of classes modified that have at least one smell and the percentage of clean classes. The figure shows how the large part of these files changed has at least one smell, suggesting that developers always change classes that have at least one smells.

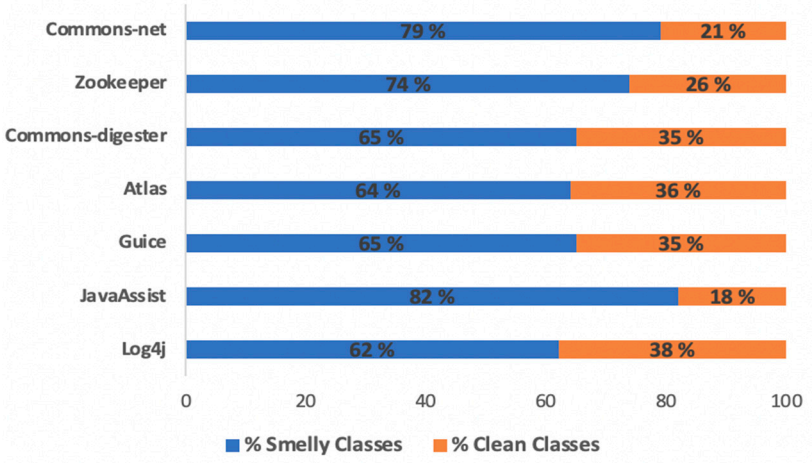

Figure 2. Percentage of smelly and clean files.

Figure 3 depicts the bar chart representing the number changes on classes where design smells are detected for each software system analyzed. In particular, it represents the number of classes with 0, 1, 2, 3 or 4 smells detected. It is possible to observe that the distribution is the same, which is the number of classes decreases when the number of smells increases.

These data confirm the idea that few classes are responsible for the large part of design violations. To investigate more in detail this aspect Figure 3 reports the changes performed on classes that are affected by more than one smell.

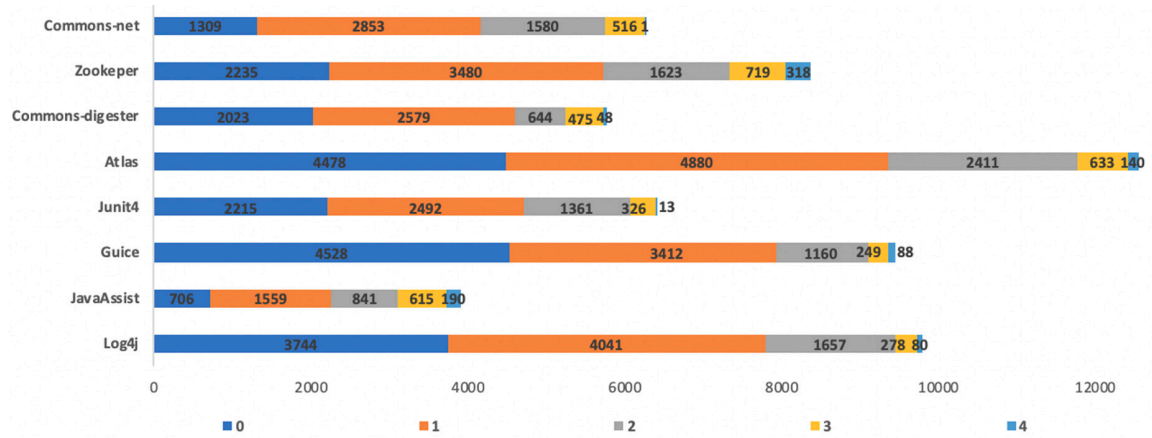

Figure 3. Number of classes that contain one or more smells.

Figure 4 reports, for each system, the percentage of classes affected by changes, in respect to the all classes, distributed among the different types of smells. Specifically, it can be observed that in all the systems, the classes most frequently changed are the ones affected by the following smells: unutilized abstraction, deficient encapsulation, cyclic-dependent modularization, insufficient modularization, and broken hierarchy. This result manifests an improper use of inheritance in the analyzed software and reports the evidence that when it occurs major negative effects can be observed during the maintenance activities.

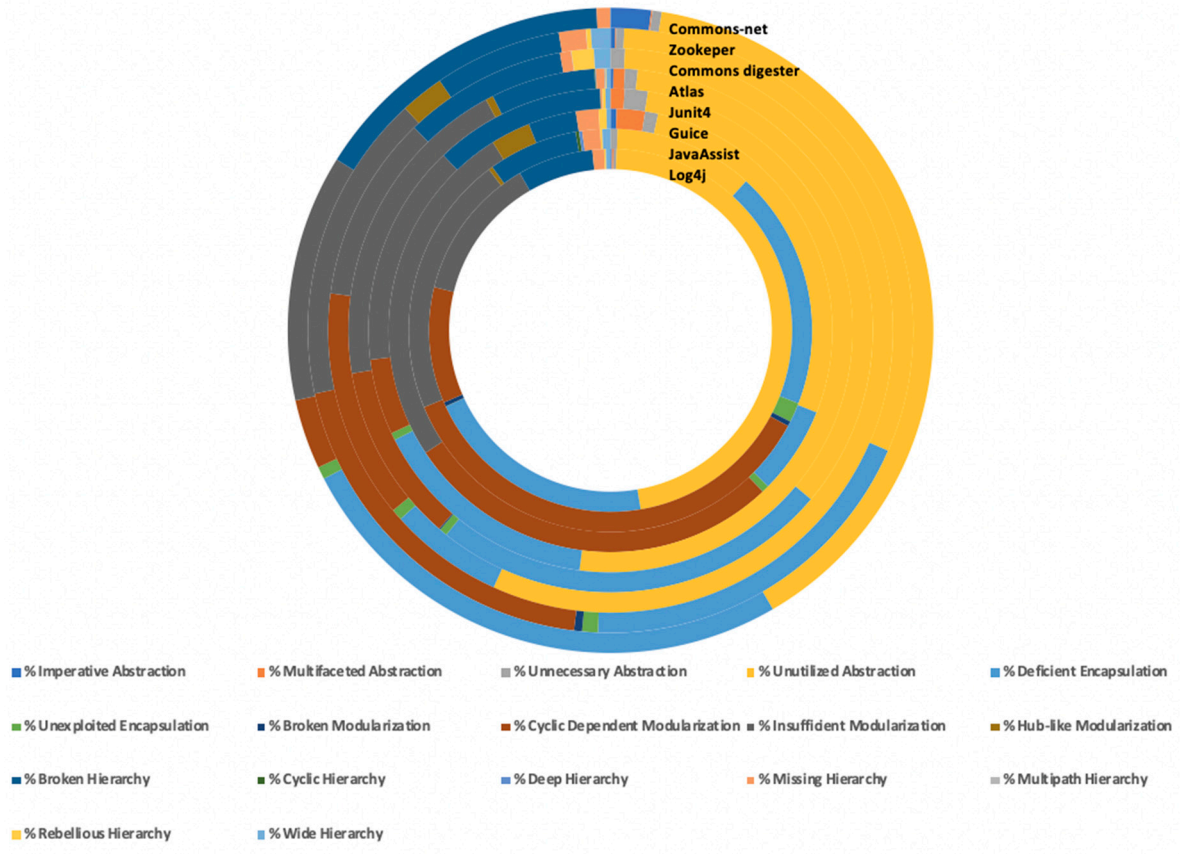

Figure 4. Percentage of changes on smelly files.

Specifically, Figure 3 depicts according to the smelly classes more frequently changed the number of co-occurring smells on those classes, distinguishing the different types.

The analysis of this figure shows that some smells pairs are more frequently co-occurring in the classes subject to changes. The analysis of the figure confirms that the classes mainly subject to changes are affected by the following type of smell: unutilized abstraction, deficient encapsulation, cyclic-dependent modularization, insufficient modularization and broken hierarchy. This reveals that when modularization problems occur than significant difficulties emerge about the architecture of the software system, indeed, cycle dependency modularization smells are also detected. This can lead to an increase of severity in the maintenance tasks. The data have been statistically analyzed using the Spearman rho, which is a rank-based correlation coefficient non-parametric.

In detail, in Log4j there is a high number of changed classes where both the unutilized abstraction e deficient encapsulation smells are detected, 666 classes, (p-value = 0.006). Similarly, the changed classes where the cyclic-dependent modularization and insufficient modularization, and unutilized abstraction and broken hierarchy are respectively 446 and 385 (p-value = 0.000). In the case of JavaAssist, the number of changed classes is higher, in particular, 1028 changes have been performed on classes affected by both the cyclic-dependent modularization and insufficient modularization smells (p-value = 0.000).

Nevertheless, there is a relevant number of changes even in classes where the pair (deficient encapsulation, cyclic-dependent modularization, p-value = 0.000) and (deficient encapsulation, insufficient modularization, p-value = 0.000) smells are detected, respectively 843 and 738. For the Guice system, the classes more changed are the ones with the pair (cyclic-dependent modularization and insufficient modularization, p-value = 0.000) of smells, i.e., 707 changes. This result appears particularly relevant as the successive values are equals to 205 and refer to the pair of smells (cyclic-dependent modularization, deficient encapsulation, p-value = 0.000).

In the case of JUnit4, it is possible to observe that the classes more frequently changed are the ones affected by the unutilized abstraction smell. In particular, classes affected by both unutilized abstraction smell and deficient encapsulation (p-value = 0.000), insufficient modularization (p-value = 0.000), broken hierarchy (p-value = 0.043), smells exhibit a higher number of changes. A very similar case occurs for the Atlas system.

Figure 5 reports that there is a relevant number of classes subject to frequent changes that are affected by more smells, that are: unutilized abstraction, deficient encapsulation, cyclic-dependent modularization, insufficient modularization, and broken hierarchy (p-values = 0.000).

In detail, 987 classes affected by deficient encapsulation and insufficient modularization are subject to change, similarly, 726 and 694 are the classes affected by deficient encapsulation and cyclic-dependent modularization, and, cyclic-dependent modularization and insufficient modularization.

Results for Commons-Digester evidenced that classes with the unutilized abstraction smell detected are frequently committed, and, more often when even the deficient encapsulation and the broken hierarchy smells are detected, involving respectively 221 and 404 classes. Moreover, 226 changes have been performed on classes where both the insufficient modularization and cyclic-dependent modularization smells were detected. In the case of Zookeeper, it emerges a relevant number of changes for classes the following smells are detected: deficient encapsulation, cyclic-dependent modularization, and insufficient modularization, respectively 1077, 1024 and 1126.

Finally, the results obtained for Commons-net point out that a relevant number of changes are performed on classes where the following smells are detected: unutilized abstraction, deficient encapsulation, insufficient modularization and broken hierarchy with 830, 679, 350, and 348 changes.

Summary for RQ1. Overall, it can be observed that in all the systems, the classes most frequently changed are affected by the following smells: unutilized abstraction, deficient encapsulation, cyclic-dependent modularization, insufficient modularization, and broken hierarchy.

These data confirm the idea that few classes are responsible for the large part of design violations.

Specifically, it can be observed that in all the systems, the classes most frequently changed are those where the following smells are co-occurring: unutilized abstraction, deficient encapsulation, cyclic-dependent modularization, insufficient modularization, and broken hierarchy. This suggests

maintainers the need for continuous monitoring of the presence of multiple design smells in a software system in order to avoid a critical increase in change activities.

Figure 5. Number of co-changing smelly files.

5.2. RQ2—To What Extent Are Design Smells Removed during the System's Evolution?

To investigate in more detail, how design smells are removed during the evolution of the system, an analysis has been performed to understand when smells are added and/or removed during the history of a software system, and in case if this addition or removal involves more than one smell at the same time.

Tables 5 and 6 report, that for each type of smell, the percentage of smells added, and the percentage of smells removed in each class, during the evolution of the system, detecting them commit by commit.

Table 5. Smells added and distributed for types.

Smell	Log4j	Java Assist	Guice	Junit4	Atlas	Commos Digester	Zookeeper	Commons Net
Imperative Abstraction	0.00%	0.00%	13.33%	0.00%	22.73%	0.00%	14.81%	6.54%
Multifaceted Abstraction	25.00%	0.00%	13.38%	13.79%	16.47%	0.00%	0.00%	20.00%
Unnecessary Abstraction	41.67%	14.29%	1.54%	2.06%	7.37%	0.00%	0.00%	2.86%
Unutilized Abstraction	5.43%	5.41%	5.87%	1.62%	4.49%	1.22%	0.43%	1.35%
Deficient Encapsulation	5.15%	2.21%	10.48%	3.73%	4.00%	0.61%	1.80%	0.36%
Unexploited Encapsulation	0.00%	8.33%	19.44%	3.23%	8.47%	3.13%	2.27%	2.13%
Broken Modularization	6.25%	4.17%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Cyclic-Dependent Modularization	9.44%	1.91%	7.47%	7.94%	5.80%	2.30%	2.09%	1.52%
Insufficient Modularization	8.60%	1.71%	7.68%	4.65%	7.89%	1.51%	3.13%	0.64%
Hub-like Modularization	0.00%	20.00%	7.58%	0.00%	10.53%	0.00%	3.29%	0.00%
Broken Hierarchy	11.36%	1.59%	9.17%	3.50%	9.45%	1.94%	3.23%	1.29%
Cyclic Hierarchy	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Deep Hierarchy	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Missing Hierarchy	0.00%	7.95%	9.73%	12.50%	9.52%	3.13%	1.33%	0.00%
Multipath Hierarchy	0.00%	0.00%	0.00%	0.00%	44.44%	0.00%	0.00%	0.00%
Rebellious Hierarchy	0.00%	10.00%	17.95%	0.00%	16.67%	4.76%	7.69%	0.00%
Wide Hierarchy	5.88%	2.50%	30.00%	0.00%	3.23%	10.64%	2.75%	0.00%

From the analysis of the data, it is possible to observe that numerous design smells are removed and added during the history of the software system, especially in classes with more frequently detected smells.

In particular, the percentage of smells removed is greater than the reintroduced one, suggesting the hypothesis that the changes in the various classes lead to improvements.

To investigate the relationship among the different design smells, Figure 6 reports the group of smells that are co-removed from a class, i.e., in the same commit. The graphs of commons digester and commons net are not included as for these systems there are not co-removed smells.

Table 6. Smells removed and distributed for types.

Smell	Log4j	JavaAssist	Guice	Junit4	Atlas	Commons Digester	Zookeeper	Commons Net
Imperative Abstraction	0.00%	0.00%	10.00%	0.00%	9.09%	0.00%	7.41%	4.58%
Multifaceted Abstraction	25.00%	0.00%	11.97%	12.07%	14.12%	0.00%	0.00%	0.00%
Unnecessary Abstraction	33.33%	2.86%	13.85%	12.37%	16.84%	2.44%	11.11%	2.86%
Unutilized Abstraction	11.48%	3.95%	13.05%	5.88%	10.37%	3.81%	4.88%	4.53%
Deficient Encapsulation	7.42%	0.85%	9.34%	4.24%	5.10%	3.99%	2.45%	0.62%
Unexploited Encapsulation	50.00%	3.13%	16.67%	3.23%	5.08%	0.00%	0.00%	2.13%
Broken Modularization	9.38%	0.00%	100.00%	0.00%	10.00%	0.00%	8.89%	0.00%
Cyclic-Dependent Modularization	10.01%	0.50%	7.94%	8.25%	7.29%	2.30%	2.18%	0.76%
Insufficient Modularization	8.42%	0.78%	7.22%	2.59%	6.23%	0.75%	1.86%	0.21%
Hub-Like Modularization	0.00%	5.00%	6.57%	0.00%	7.02%	0.00%	1.23%	0.00%
Broken Hierarchy	12.24%	0.91%	10.42%	5.46%	15.45%	7.10%	7.16%	1.37%
Cyclic Hierarchy	0.00%	0.00%	0.00%	100.00%	100.00%	0.00%	0.00%	0.00%
Deep Hierarchy	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%
Missing Hierarchy	0.00%	3.41%	11.50%	12.50%	4.76%	0.00%	0.67%	1.96%
Multipath Hierarchy	0.00%	0.00%	0.00%	0.00%	44.44%	100.00%	50.00%	0.00%
Rebellious Hierarchy	7.69%	10.00%	23.08%	0.00%	16.67%	4.76%	0.00%	0.00%
Wide Hierarchy	5.88%	0.00%	15.00%	4.76%	16.13%	0.00%	1.83%	100.00%

As already emerged in RQ1, the classes more subject to removal are the ones affected by the following smells: unutilized abstraction, deficient encapsulation, cyclic-dependent modularization, insufficient modularization, and broken hierarchy.

In the case of Log4j, it emerges that unutilized abstraction is the smell more frequently co-removed with other smells. Indeed, it is co-removed with the deficient encapsulation, insufficient modularization, and broken hierarchy design smell.

The results of JavaAssist highlight that there is a limited number of classes with smell removal. Indeed, from Figure 6 it is possible to observe that the values of removal for the different types of smells are often closed to zero.

In the case of Guice, there are several smells co-removal, in particular with the insufficient modularization smell. It is often removed with the unutilized abstraction, deficient encapsulation, and cyclic-dependent modularization smell.

Junit4 exhibits pics of co-removed smells in few cases, while the large part of removals is limited and closed to zero. Nevertheless, among the smells co-removed emerge the unutilized abstraction with the deficient encapsulation and broken hierarchy smells.

In the case of Atlas, the most frequently co-removed smell is the insufficient modularization. Indeed, it is often removed with the unutilized abstraction, deficient encapsulation, and cyclic-dependent modularization smell.

Common Digester's results highlight that there are very limited cases of smell removals, in this case, the large part of values are equal to zero.

Figure 6. Number of co-removed smells.

Figure 6 confirms also for Zookeeper that the smell pairs co-removed are: unutilized abstraction, deficient encapsulation, insufficient modularization, and broken hierarchy. In particular, unutilized abstraction and broken hierarchy are the design smells more frequently co-removed.

In Common Net, there are very few smells removed. Indeed, the corresponding values are often equal to zero, except for the unutilized abstraction design smell.

Summary for RQ2. Overall, it can be observed that in all the systems, the most frequently removed smells are unexploited encapsulation, unutilized abstraction, deficient encapsulation, unnecessary abstraction, and broken hierarchy. All these smells deal with class responsibilities.

As explained in the discussion section, it has been observed that design smells are removed due to a restructuring of the architecture of the software project. This means that more activities are contextual, such as code commented, code replaced, new code introduced, and some refactoring performed.

Instead, the smells more frequently introduced are multifaceted abstraction and unnecessary abstraction. Both these smells generally occur when class responsibilities are not adequately designed,

that is a class assumes multiple or not needed responsibilities. Here the use "extract class" refactoring could help developers to improve the current design.

Specifically, it can be observed that in all the systems, the most frequently co-removed smells are the following: unutilized abstraction, deficient encapsulation, insufficient modularization, and broken hierarchy. In particular, unutilized abstraction and broken hierarchy are the design smells more frequently co-removed.

5.3. RQ3—To What Extent Can Smell Removal Be Related to Refactorings or Issues?

While results of RQ2 indicate that smells removals highly co-occur for a different type, such results do not tell yet whether refactorings contribute to these removals. Table 7 reports the percentage of those refactoring that involved classes with smell removals. As the table shows, the number of cases in which refactorings actually occur on the smells-removed source code ranges between 3 of Common Digester and 123 of Atlas. The latter is also the case with the majority of refactorings removals (especially concerning the removal of unutilized abstraction).

Table 7. Refactoring and smells removal co-occurrences.

Smell	Log4j		JavaAssist		Guice		JUnit4		Atlas		Common Digester		Zookeeper		Common Net		All	
Imperative Abstraction	0	0	0	0	0	3	0	0	0	2	0	0	0	2	0	7	0	14
Multifaceted Abstraction	1	2	0	0	1	16	2	5	5	7	0	0	0	0	0	0	9	30
Unnecessary Abstraction	0	8	0	1	1	8	3	9	2	14	0	1	0	5	0	1	6	47
Unutilized Abstraction	**27**	**430**	**0**	**27**	**14**	**233**	**30**	**152**	**43**	**393**	**1**	**105**	**11**	**146**	2	132	128	1618
Deficient Encapsulation	**10**	**121**	1	9	7	34	**7**	**34**	**15**	**138**	**1**	**12**	7	42	1	11	49	401
Unexploited Encapsulation	0	1	1	2	0	6	0	1	0	3	0	0	0	0	0	1	1	14
Broken Modularization	1	2	0	0	0	1	0	0	0	1	0	0	0	4	0	0	1	8
Cyclic-Dependent Modularization	12	76	2	9	**17**	**134**	**3**	**23**	18	80	**0**	**15**	5	40	0	2	57	379
Insufficient Modularization	**9**	**83**	1	9	20	106	**2**	**22**	30	120	**0**	**4**	**2**	**30**	0	2	64	376
Hub-Like Modularization	0	0	0	1	4	9	0	0	1	3	0	0	0	3	0	0	5	16
Broken Hierarchy	4	66	1	3	**2**	**23**	**4**	**35**	**9**	**112**	**0**	**33**	**5**	**46**	4	12	29	330
Cyclic Hierarchy	0	0	0	0	0	0	0	1	0	6	0	0	0	0	0	0	0	7
Deep Hierarchy	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	5
Missing Hierarchy	0	0	1	2	0	13	0	1	0	3	0	0	0	1	0	1	1	21
Multipath Hierarchy	0	0	0	0	0	0	0	0	0	4	0	1	0	1	0	0	0	6
Rebellious Hierarchy	0	1	0	1	0	9	0	0	0	1	1	2	0	0	0	0	1	14
Wide Hierarchy	0	2	0	0	0	3	0	1	0	5	0	0	0	2	0	1	0	14
Overall	64	792	7	64	66	598	51	284	123	897	3	173	30	322	7	170	351	3300

For each project, the first column indicates where there is refactoring, the second indicates where there isn't refactoring.

Overall, it can be observed from the table that the large part of smell removals does not co-occur with refactoring in the same class. More in detail, it can be noted that in Log4j, just the 3% deficient encapsulation smells are removed in co-occurrence with refactoring.

Similarly, 7% of the 10% of unutilized abstraction and insufficient modularization smells were removed in co-occurrence with refactorings. In the case of JavaAssist, a relevant number of unutilized abstraction smells are removed in the presence of refactorings. From the analysis of Guice smells emerged that the 8% and 11% of respectively broken hierarchy and cyclic-dependent modularization smells are removed in the same classes where refactoring were detected.

In JUnit4 it can be noted that the large part of smells removed belongs to unutilized abstraction, deficient encapsulation, cyclic-dependent modularization, insufficient modularization, and broken hierarchy smell types. In particular, the co-occurrence of refactorings is low for all the types of smell removed. In Atlas, the smells mainly removed are as follows: unutilized abstraction, deficient encapsulation and broken hierarchy, with about 10% of co-occurrence of refactorings.

In Common Digester, it is possible to observe that the co-occurrence with refactoring is near to zero. In Zookeeper, smells mainly removed are unutilized abstraction, insufficient modularization, and broken hierarchy with about 10% co-occurrence of refactorings.

From a statistical perspective, we have checked whether, in a commit that involves a smell removal, refactorings are more likely to occur on the smell related code than in another source code.

Results of the Fisher's exact test reported in Table 8, indicate that there is no statistically significant evidence for all the systems analyzed.

Table 8. Proportion of refactorings involving smell-affected code and other code: Fisher's exact test results.

Project	*p*-Value	Odds Ratio
Log4j	0.06901	0.78
Atlas	0.8424	0.97
Commons Digester	0.377	0.54
Commons Net	0.6737	1.16
Guice	0.8391	1.02
Java Assist	0.1969	1.74
Junit4	0.3661	1.16
Zookeeper	0.4728	1.14

To get a further understanding of these data they have also been related to the presence of issues. As reported in Table 9, the occurrences of smell removal have been measured respect the co-occurring issue resolution. It is possible to observe the there are few relevant cases, and concerns the unutilized abstraction, deficient encapsulation, cycle dependent modularization and insufficient modularization smell removal, while in all the other cases the removal of the smell is not related to an issue treatment. This can suggest that the developer in these last cases specifically focused their activities on the smell removal.

Table 9. Issue resolution in presence of smell removal.

Smells	Log4j		JavaAssist		Guice		JUnit4		Atlas		Common Digesters		Zookeeper		Common-Net	
Imperative Abstraction	0	0	0	0	3	0	0	0	2	0	0	0	2	0	7	0
Multifaceted Abstraction	3	0	0	0	17	1	7	1	12	0	0	0	0	0	0	0
Unnecessary Abstraction	8	0	1	0	9	0	12	1	16	0	1	0	5	0	1	0
Unutilized Abstraction	**457**	**4**	**27**	**0**	**247**	**1**	**182**	**18**	**436**	**7**	**106**	**0**	**157**	**2**	**134**	**0**
Deficient Encapsulation	**131**	**0**	**10**	**0**	**41**	**1**	**41**	**3**	**153**	**1**	**13**	**0**	**49**	**0**	**12**	**0**
Unexploited Encapsulation	1	0	3	0	6	1	1	0	3	0	0	0	0	0	1	0
Broken Modularization	3	0	0	0	1	0	0	0	1	0	0	0	4	0	0	0
Cyclic-Dependent Modularization	**88**	**0**	**11**	**1**	**151**	**3**	**26**	**0**	**98**	**1**	**15**	**0**	**45**	**0**	**2**	**0**
Insufficient Modularization	**92**	**0**	**10**	**0**	**126**	**3**	**24**	**3**	**150**	**0**	**4**	**1**	**32**	**0**	**2**	**0**
Hub-like Modularization	0	0	1	0	13	1	0	0	4	0	0	0	3	0	0	0
Broken Hierarchy	70	0	4	0	25	0	39	1	121	0	33	2	51	1	16	0
Cyclic Hierarchy	0	0	0	0	0	0	1	0	6	1	0	0	0	0	0	0
Deep Hierarchy	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0
Missing Hierarchy	0	0	3	0	13	1	1	0	3	0	0	0	1	0	1	0
Multipath Hierarchy	0	0	0	0	0	0	0	0	4	0	1	0	1	0	0	0
Rebellious Hierarchy	1	0	1	1	9	0	0	0	1	0	3	0	0	0	0	0
Wide Hierarchy	2	0	0	0	3	0	1	0	5	0	0	0	2	0	1	0

For each project the first column indicates where the removal of the smell was found, the second indicates the presence of the removal of the smell and the issue.

Summary for RQ3. Overall, it can be observed that there is not co-occurrence of design smells removal and refactorings. This suggests that in all the systems analyzed the large part of smells removal was performed without any relationship with refactorings.

5.4. Qualitative Analysis

To understand what types of changes were made when a particular type of smell has been removed, more specifically when this has occurred with the presence of refactoring, a qualitative analysis was conducted to analyze the commit messages and related changes made in the source code.

This analysis aims to understand if the removal of smells and refactoring are linked by a cause–effect relationship, or if they are simply two activities that occur in a completely random way in the context of improving the quality of the code.

Analyzing the commit messages of the files in which the removal of the smells and the refactoring had been found, it became evident that there is never an explicit reference to the removal of the smell. On the contrary, references have often been found to the refactoring activities carried out, through the use of words, such as “refactoring”, “restructuring”, or “general reorganization”. Following this analysis, it is possible to say that it could be a few cases, in which, refactoring could have contributed to the removal of the smell.

To report some examples where refactoring occurred along with the removal of the smell, there is a commit in Log4j (c6e0193f8a5a9d84030d8e8f65fcf4cb18f5d0e), where the author explicitly

states: "/apache/log4j/test/TP.java is not needed", as confirmation of the removal of the smell unutilized abstraction.

In addition, in Log4j, there is a commit (da37a1de7a89a586c43fe03aaead20642dda8dcf) that reports the removal of a smell deficient encapsulation, and in the commit message, it reads "Updated Category.java to use ClassLoader.getSystemResource in its static initializer. This is much better than the silly stuff we did before."

Looking also at the source code, it would seem that the mentioned class is removed, and its operations are inserted in an already existing class, therefore the accessibility to these operations changes.

In JavaAssist, a commit (42e1dbed4e870e5c452fa5173ac10921c46d06d0) mentioned "implemented javassist. bytecode. stackmap package". Here the removal of an insufficient modularization smell was found, in fact, in the source code there is a decomposition because the methods are redistributed among the classes, it could, therefore, be a refactoring that contributed to the removal of the smell.

In the Commons-Net system, there is a commit (88a631049caa76e8433dd0fcd7f3f97d4c93e383) where the smell unutilized abstraction has been removed and the commit message reads precisely: "removal of recently added files no longer needed because of most recent simplification."

In many cases such as Atlas commit (22624786ee4fe86c94d94fee4bcf4c0855919901), there is the removal of the unutilized abstraction smell, the commit message reports "removed un-used modules" and the class is completely removed as if to confirm the removal of that particular smell.

Considering that few cases have been found in which it could be thought that refactoring contributed to the removal of the smell, it is possible to say that these two activities are carried out together by chance. Finally, we can speak of a simple reorganization of the code architecture and system components.

6. Threats to Validity

In this section, the threats to the validity of the research proposed are discussed.

Construct validity: the main construct validity is due to imprecisions/errors in the measurements we performed. The most significant threats to validity involve the accuracy of design smells detection tools and the refactoring detection tool. To mitigate these threats, Designite and Refactoring Miner have been used, which are widely used tools to detect smells, and mining refactoring respectively.

Internal validity: The threat to internal validity concerns whether the results in our study correctly follow from our data. Particularly, whether the metrics are meaningful to our conclusions and whether the measurements are adequate. To this aim, an accurate process for the data gathering has been performed. Moreover, threats to internal validity concern factors that could influence the observations made. Indeed, the observations may be influenced by the phase of the change history, as well as by the developer itself. To this aim, a qualitative investigation has been conducted.

External validity: Threat to external validity concerns the generalizability of the obtained results, as the size of our selected data set is smaller than the population of all OSS systems which could impact the generalizability of our conclusions. To mitigate this threat, well-known systems have been considered. Additionally, these systems are continuously evolving and different for dimensions, domain, size, timeframe, and the number of commits. However, several limitations to the generalizability of conclusions remain.

7. Conclusions

This document proposes an empirical study aimed at contributing to the relevant research problems related to the presence of design odors within a software system. To this end, an empirical study was conducted considering the complete evolution of a software system, commit by commit, to analyze maintenance-related activities. In particular, eight open-source projects were analyzed.

The relationships between commits detected odors and refactoring were examined in detail.

In particular, the Designite and Refactoring Miner tools were used to identify smells and refactoring activities. For the latter Tsantalis et al. [30] reported high precision and recall values, respectively 98% and 87%.

Preliminary results outline some visible indicators that activities on stinky files have higher frequencies than all other files.

Furthermore, it has been observed that in all systems, the most frequently removed odors are: unexploited encapsulation, unused abstraction, insufficient encapsulation, unnecessary abstraction, and broken hierarchy.

Instead, the most frequently introduced odors are faceted abstraction and unnecessary abstraction.

In particular, it can be observed that in all systems, the most frequently removed odors are unused abstraction and broken hierarchy. As explained in the discussion, this is due to the restructuring of the system architecture, which mainly encourages these removals.

Overall, it can be observed that the removal and refactoring of design odors do not occur simultaneously.

This suggests that in all the systems analyzed, most of the odor removal was performed without any relation to refactoring.

Our long-term goal is to predict architecture decay and potential future problems based on information available at the implementation level.

Author Contributions: Conceptualization, L.A., U.C.; Methodology, L.A., M.I.; Investigation U.C., M.I.; Data curation U.C, M.I.; Writing—original draft preparation, L.A., U.C.; Writing—review and editing, L.A., M.I. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Hochstein, L.; Lindvall, M. Combating architectural degeneration: A survey. *Inf. Softw. Technol.* **2005**, *47*, 643–656. [CrossRef]
2. Garcia, J.; Popescu, D.; Edwards, G.; Medvidovic, N. Toward a Catalogue of Architectural Bad Smells. In *Computer Vision*; Springer Science and Business Media LLC: Berlin/Heidelberg, Germany, 2009; Volume 5581, pp. 146–162.
3. Le, D.; Medvidovic, N. Architectural-based speculative analysis to predict bugs in a software system. In *Proceedings of the Proceedings of the 38th International Conference on Software Engineering Companion*; Association for Computing Machinery (ACM): New York, NY, USA, 2016; pp. 807–810.
4. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Springer Science and Business Media LLC: Berlin/Heidelberg, Germany, 2002; Volume 2418, p. 256.
5. Alkharabsheh, K.; Crespo, Y.; Manso, E.; Taboada, J.A. Software Design Smell Detection: A systematic mapping study. *Softw. Qual. J.* **2018**, *27*, 1069–1148. [CrossRef]
6. Sharma, T.; Mishra, P.; Tiwari, R. Designite. In *Proceedings of the 1st International Workshop on Agents and CyberSecurity-ACySE '14*; Association for Computing Machinery (ACM): New York, NY, USA, 2016; pp. 1–4.
7. Raymond, E. The cathedral and the bazaar. *Knowl. Soc.* **1999**, *12*, 23–49. [CrossRef]
8. Bar, M.; Fogel, K. *Open Source Development with CVS*, 3rd ed.; Paraglyph Press: Norwich, NY, USA, 2003; ISBN 1-932111-81-6.
9. Sharma, S.; Sugumaran, V.; Rajagopalan, B. A framework for creating hybrid-open source software communities. *Inf. Syst. J.* **2002**, *12*, 7–25. [CrossRef]
10. Kumar, L.; Sureka, A. Thirteen Years of Mining Software Repositories (MSR) Conference-What is the Bibliography Data Telling Us? *arXiv* **2016**, arXiv:1609.06266.
11. Sjøberg, D.I.K.; Yamashita, A.; Anda, B.C.; Mockus, A.; Dybå, T. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Trans. Softw. Eng.* **2012**, *39*, 1144–1156. [CrossRef]

12. Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Di Penta, M.; De Lucia, A.; Poshyvanyk, D. *When and Why Your Code Starts to Smell Bad*; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA, 2015; Volume 1, pp. 403–414.
13. Abbes, M.; Khomh, F.; Gueheneuc, Y.-G.; Antoniol, G. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA, 2011; pp. 181–190.
14. Khomh, F.; Di Penta, M.; Guéhéneuc, Y.-G.; Antoniol, G. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empir. Softw. Eng.* **2011**, *17*, 243–275. [CrossRef]
15. Palomba, F.; Bavota, G.; Di Penta, M.; Fasano, F.; Oliveto, R.; De Lucia, A. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. *Empir. Softw. Eng.* **2017**, *23*, 1188–1221. [CrossRef]
16. Moha, N.; Gueheneuc, Y.-G.; Duchien, L.; Le Meur, A.-F. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Softw. Eng.* **2009**, *36*, 20–36. [CrossRef]
17. Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; Poshyvanyk, D.; De Lucia, A. Mining Version Histories for Detecting Code Smells. *IEEE Trans. Softw. Eng.* **2015**, *41*, 462–489. [CrossRef]
18. Tsantalis, N.; Chatzigeorgiou, A. Identification of Move Method Refactoring Opportunities. *IEEE Trans. Softw. Eng.* **2009**, *35*, 347–367. [CrossRef]
19. Suryanarayana, G.; Samarthyam, G.; Sharma, T. *Refactoring for Software Design Smells: Managing Technical Debt*, 1th ed.; Morgan Kaufmann: Burlington, MA, USA, 2014.
20. Le, D.M.; Link, D.; Shahbazian, A.; Medvidovic, N. An Empirical Study of Architectural Decay in Open-Source Software. In *Proceedings of the 2018 IEEE International Conference on Software Architecture (ICSA)*; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA, 2018; pp. 176–17609.
21. Fontana, F.A.; Lenarduzzi, V.; Roveda, R.; Taibi, D. Are architectural smells independent from code smells? An empirical study. *J. Syst. Softw.* **2019**, *154*, 139–156. [CrossRef]
22. Wang, Y.; Yu, H.; Zhu, Z.-L.; Zhang, W.; Zhao, Y.-L. Automatic Software Refactoring via Weighted Clustering in Method-Level Networks. *IEEE Trans. Softw. Eng.* **2018**, *44*, 202–236. [CrossRef]
23. Brunet, J.; Bittencourt, R.; Serey, D.; Figueiredo, J. On the Evolutionary Nature of Architectural Violations. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA, 2012; pp. 257–266.
24. Fontana, F.A.; Pigazzini, I.; Roveda, R.; Tamburri, D.; Zanoni, M.; Di Nitto, E. Arcan: A Tool for Architectural Smells Detection. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA, 2017; pp. 282–285.
25. Designite Tools. Available online: http://www.designite-tools.com (accessed on 2 July 2020).
26. Pietrzak, B.; Walter, B. Leveraging Code Smell Detection with Inter-smell Relations. In Proceedings of the International Conference on Extreme Programming and Agile Processes in Software Engineering, XP 2006, Oulu, Finland, 17–22 June 2016; pp. 75–84.
27. Mo, R.; Cai, Y.; Kazman, R.; Xiao, L. Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. In *Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells*; Defense Technical Information Center (DTIC): Fort Belvoir, VA, USA, 2015; pp. 51–60.
28. Walter, B.; Fontana, F.A.; Ferme, V. Code smells and their collocations: A large-scale experiment on open-source systems. *J. Syst. Softw.* **2018**, *144*, 1–21. [CrossRef]
29. Yamashita, A.; Zanoni, M.; Fontana, F.A.; Walter, B. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA, 2015; pp. 121–130.
30. Tsantalis, N.; Mansouri, M.; Eshkevari, L.M.; Mazinanian, D.; Dig, D. Accurate and efficient refactoring detection in commit history. In *Proceedings of the Proceedings of the 40th International Conference on Software Engineering-ICSE '18*; Association for Computing Machinery (ACM): New York, NY, USA, 2018; pp. 483–494.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).