# A Large-Scale Empirical Study on the Lifecycle of Code Smell Co-occurrences

Fabio Palomba[a], Gabriele Bavota[b], Massimiliano Di Penta[c], Fausto Fasano[d], Rocco Oliveto[d], Andrea De Lucia[e]

[a] *University of Zurich - Binzmuhlestrasse 14, CH-8050 Zurich, Switzerland*
[b] *Università della Svizzera italiana - Via Giuseppe Buffi 13, 6900 Lugano, Switzerland*
[c] *University of Sannio - Via Traiano, 3, 82100 Benevento, Italy*
[d] *University of Molise - Via Hertz, 1, 86090 Pesche, Italy*
[e] *University of Salerno - Via Giovanni Paolo II, 132, 84084 Fisciano, Italy*

**Abstract**

**Context.** Code smells are suboptimal design or implementation choices made by programmers during the development of a software system that possibly lead to low code maintainability and higher maintenance costs.

**Objective.** Previous research mainly studied the characteristics of code smell instances affecting a source code file, while only few studies analyzed the magnitude and effects of smell co-occurrence, *i.e.,* the co-occurrence of different types of smells on the same code component. This paper aims at studying in details this phenomenon.

**Method.** We analyzed 13 code smell types detected in 395 releases of 30 software systems to firstly assess the extent to which code smells co-occur, and then we analyze (i) which code smells co-occur together, and (ii) how and why they are introduced and removed by developers.

**Results.** 59% of smelly classes are affected by more than one smell, and in particular there are six pairs of smell types (*e.g., Message Chains and Spaghetti Code*) that frequently co-occur. Furthermore, we observed that method-level code smells may be the root cause for the introduction of class-level smells. Finally, code smell co-occurrences are generally removed together as a consequence of other maintenance activities causing the deletion of the affected code components (with a consequent removal of the code smell instances) as well as the result of a major restructuring or scheduled refactoring actions.

**Conclusions.** Based on our findings, we argue that more research aimed at designing co-occurrence-aware code smell detectors and refactoring approaches is needed.

*Keywords:* Code Smells Co-Occurrences, Empirical Study, Mining Software Repositories

## 1. Introduction

Software systems are continuously evolved to introduce new features, to adapt them to varying execution and usage contexts, or to correct defects [1]. In such a scenario, and especially when time-to-market is crucial or when there are other stringent deadlines, developers are required to make design decisions and to implement changes in the shortest time possible [2, 3, 4]. As a direct consequence, developers do not always have the chance to apply changes that preserve the system design quality, possibly leading to the introduction of design defects, also known as *technical debt* [5, 6, 7]. Among the different types of technical debt, one of the many issues involving the source code is the presence of *code smells*, *i.e.,* symptoms of poor design and implementation choices [8, 9].

The negative impact of code smells on non-functional attributes has been widely analyzed by previous work. For instance, Khomh *et al.* [10] empirically investigated the extent to which the presence of code smells makes classes more change- and fault-prone, finding statistically significant correlations between these phenomena. Other studies investigated the relevance of the problem for developers [11], when and why code smells are introduced [12, 13], their longevity [14, 15, 16, 17, 18], and their impact on maintainability properties [19, 20, 21, 22].

Also, the research community devised automatic detectors able to identify instances of code smells in the source code. Most of such techniques rely on the analysis of the structural properties of the source code (*e.g.,* method calls) to define heuristics aimed at alerting developers of the presence of a code smell [23, 24, 25, 26]. More recently, the use of historical and textual information has also been proposed as an alternative to standard detectors [27, 28].

Most of the research summarized above focused on studying the effects of a single code smell instance affecting a code component, while very few studies investigated problems occurring when several different types of smell instances affect the same component. Indeed, the interaction of code smells has been only recently the object of empirical studies that demonstrated some of the potential risks of code smells co-occurrence. In particular, Yamashita and Moonen [22] showed that the interaction of code smells consistently inhibits the ability of developers to maintain code. More importantly, Palomba *et al.* [29] showed that classes affected by more than one smell are up to 350% more change-prone and 100% more fault-prone than classes affected by a single smell instance. A possible reason—as reported by Abbes *et al.* [30]—is that the co-occurrence of code smells strongly hinder the developers' ability to understand the source code.

Although these studies empirically showed the harmfulness of code smell interactions, there is a noticeable lack of knowledge about (i) the magnitude of the observed phenomenon, *i.e.,* how frequently code smells co-occur; (ii) how the phenomenon manifests, *i.e.,* which code smells tend to co-occur more frequently; and (iii) the likely reasons of the co-occurrences, *i.e.,* how and why developers introduce and remove code smell co-occurrences. These pieces of information are of a paramount importance to correctly understand the phenomenon and devise methodologies aimed to effectively deal with the problem.

To this aim, this paper presents a large-scale empirical study aimed at quantifying the diffuseness of the problem in terms of how frequently code smells occur together, and the underlying dynamics leading developers to the introduction and removal of multiple types of code smell instances in the same code component.

The study considers 13 code smell types from the catalogs by Fowler [9] and Brown *et al.* [8], and has been conducted across 395 releases belonging to 30 open source software systems. To the best of our knowledge, this is the largest study aimed at characterizing the phenomenon of code smell co-occurrences.

Our results provide evidence on the high diffuseness of code smell co-occurrences; we observed that almost 59% of smelly classes contain more than one type of code smell instances. Specifically, we identified frequent co-occurrences between six code smell pairs, *i.e., Message Chains–Spaghetti Code*, *Message Chains–Complex Class*, *Message Chains–Blob*, *Message Chains–Refused Bequest*, *Long Method–Spaghetti Code*, and *Long Method–Feature Envy.*

Moreover, when analyzing how code smell co-occurrences are introduced, we found that often code smells affecting a single method (*e.g., Message Chains*) represent the trigger for class-level code smells introduction, thus being the triggering event for the "infection" of the entire class. Finally, we characterize how co-occurrences are removed, finding that in most cases the code smells affecting the same code component disappear all together. We complement our quantitative analysis with qualitative examples aimed at providing the motivations behind the introduction and removal of code smell co-occurrences. From this analysis, we learned that code smells are generally removed together as a consequence of maintenance and evolution activities.

Our findings have implications for both researchers and practitioners. For researchers, the observed co-occurrences of code smells and the temporal sequence in which they appear in the affected class represent information potentially useful to build co-occurrence-aware code smell detectors and refactoring recommenders. Such tools could be able to proactively recommend refactorings not only related to the smell being detected, but also warn for consequences of other smells that can co-occur with it. For example, a class affected by the Message Chains smell is more likely to also become a Complex Class in the future. For practitioners, knowing how code smells co-occur can help in reasoning about code design principles that, once violated, can lead to the introduction of several code smells, all triggered by the same violation. Also, being aware that (i) the introduction of a code smell instance is likely to trigger the introduction of other smells and (ii) classes affected by several code smells have a higher change- and fault-proneness as compared to classes affected by a single instance [29], highlights the risk to pay very high technical-debt interests as a consequence of an apparently minor compromise in terms of good design practices, thus stressing even more the need for high-quality code.

**Structure of the paper.** Section 2 describes the study objectives and design, while Section 3 reports the achieved results. Section 4 discusses the threats possibly affecting the validity of our experiment. In Section 5 we summarize the related literature, while Section 6 concludes the paper.

## 2. Empirical Study Definition and Design

Table 1: Systems involved in the study.

| System | Description | #Releases | Classes | Methods | KLOCs |
|---|---|---|---|---|---|
| ArgoUML | UML Modeling Tool | 16 | 777-1,415 | 6,618-10,450 | 147-249 |
| Ant | Build System | 22 | 83-813 | 769-8,540 | 20-204 |
| aTunes | Player and Audio Manager | 31 | 141-655 | 1,175-5,109 | 20-106 |
| Cassandra | Database Management System | 13 | 305-586 | 1,857-5,730 | 70-111 |
| Derby | Relational Database Management System | 9 | 1,440-1,929 | 20,517-28,119 | 558-734 |
| Eclipse Core | Integrated Development Environment | 29 | 744-1,181 | 9,006-18,234 | 167-441 |
| Elastic Search | RESTful Search and Analytics Engine | 8 | 1,651-2,265 | 10,944-17,095 | 192-316 |
| FreeMind | Mind-mapping Tool | 16 | 25-509 | 341-4,499 | 4-103 |
| Hadoop | Tool for Distributed Computing | 9 | 129-278 | 1,089-2,595 | 23-57 |
| HSQLDB | HyperSQL Database Engine | 17 | 54-444 | 876-8,808 | 26-260 |
| Hbase | Distributed Database System | 8 | 160-699 | 1,523-8148 | 49-271 |
| Hibernate | Java Persistence Framework | 11 | 5-5 | 15-18 | 0.4-0.5 |
| Hive | Data Warehouse Software Facilitates | 8 | 407-1,115 | 3,725-9,572 | 64-204 |
| Incubating | Codebase | 6 | 249-317 | 2,529-3,312 | 117-136 |
| Ivy | Dependency Manager | 11 | 278-349 | 2,816-3,775 | 43-58 |
| Lucene | Search Manager | 6 | 1,762-2,246 | 13,487-17,021 | 333-466 |
| JEdit | Text Editor | 23 | 228-520 | 1,073-5,411 | 39-166 |
| JHotDraw | Java GUI Framework | 16 | 159-679 | 1,473-6,687 | 18-135 |
| JFreeChart | Java Chart Library | 23 | 86-775 | 703-8,746 | 15-231 |
| JBoss | Java Webserver | 18 | 2,313-4,809 | 19,901-37,835 | 434-868 |
| JVlt | Vocabulary Learning Tool | 15 | 164-221 | 1,358-1,714 | 18-29 |
| jSL | Java Service Launcher | 15 | 5-10 | 26-43 | 0.5-1 |
| Karaf | Standalone Container | 5 | 247-470 | 1,371-2,678 | 30-56 |
| Nutch | Web-search Software | 7 | 183-259 | 1,131-1,937 | 33-51 |
| Pig | Large Dataset Analyzer | 8 | 258-922 | 1,755-7,619 | 34-184 |
| Qpid | Messaging Tool | 5 | 966-922 | 9,048-9,777 | 89-193 |
| Sax | XML Parser | 6 | 19-38 | 119-374 | 3-11 |
| Struts | MVC Framework | 7 | 619-1,002 | 4,059-7,506 | 69-152 |
| Wicket | Java Application Framework | 9 | 794-825 | 6,693-6,900 | 174-179 |
| Xerces | XML Parser | 16 | 162-736 | 1,790-7,342 | 62-201 |
| **Total** | - | **395** | **5-4,809** | **15-37,835** | **0.4-868** |

The *goal* of the study is to analyze (i) to what extent code smells co-occur in software systems, (ii) which types of code smells tend to co-occur more frequently, and (iii) how such co-occurrences are introduced and removed by the developers. It is worth noting that with "how" we refer to the temporal relationships between the introduction and the removal of the different smell instances affecting the same code component, *e.g.,* the introduction of a smell instance of type $A$ triggers a subsequent introduction of a smell instance of type $B$. The study *perspective* is of researchers interested in understanding how code smell co-occurrences appear and disappear in large software projects, as well as of practitioners, who are interested in knowing the magnitude of the phenomenon.

More specifically, the study aims at answering the following research questions:

- $\mathbf{RQ_0}$**:** *To what extent do code smells co-occur?* This research question aimed at assessing the extent to which software systems contain classes affected by more types of code smell instances. Such a quantification is needed to obtain an indication of the magnitude of the phenomenon we are studying. For example, if just few classes are affected by more than one type of code smell instances, answering the other research questions might be of little interest.

- **RQ$_1$:** *How often a code smell type co-occurs with another code smell type?* In this research question we aimed at analyzing the types of code smells that co-occur more frequently. The results of such a research question can be used, for example, to suggest appropriate refactoring operations when multiple types of smell occur in the same code component.

- **RQ$_2$:** *How are code smell co-occurrences introduced?* The research question aimed at investigating the way new types of code smells are introduced in classes already affected by at least one code smell instance. We answered this research question by mining the *common introduction patterns*, *i.e.,* analyzing the sequence of code smells introduction. This will aid to understand whether one smell type *temporally causes* the other, or *vice versa.*

- **RQ$_3$:** *How are code smell co-occurrences removed?* With this research question we were interested in the analysis of how smell co-occurrences are removed from a software system. Specifically, we mined the frequent *common removal patterns*, *i.e.,* the sequence where code smells are removed from an affected class.

Table 2: Code smells considered in the context of the study.

| Name | Description |
|---|---|
| Blob (BL) | A large class implementing different responsibilities and centralizing most of the system processing. |
| Class Data Should Be Private (CDSBP) | A class exposing its fields, violating the principle of data hiding. |
| Complex Class (CC) | A class having at least one method having a high cyclomatic complexity. |
| Feature Envy (FE) | A method is more interested in a class other than the one it actually is in. |
| Inappropriate Intimacy (II) | Two classes exhibiting a very high coupling between them. |
| Lazy Class (LC) | A class having very small dimension, few methods and low complexity. |
| Long Method (LM) | A method that is unduly long in terms of lines of code. |
| Long Parameter List (LPL) | A method having a long list of parameters, some of which avoidable. |
| Message Chain (MC) | A long chain of method invocations is performed to implement a class functionality. |
| Middle Man (MM) | A class delegates to other classes most of the methods it implements. |
| Refused Bequest (RB) | A class redefining most of the inherited methods, thus signaling a wrong hierarchy. |
| Spaghetti Code (SC) | A class implementing complex methods interacting between them, with no parameters, using global variables. |
| Speculative Generality (SG) | A class declared as abstract having very few children classes using its methods. |

## *2.1. Context Selection*

The *context* of the study consisted of (i) software systems and (ii) code smells. As for the former, we considered 395 releases of 30 open source software systems belonging to two different ecosystems, *i.e.,* the Apache Software Foundation and the Eclipse Development Framework. Table 1 summarizes the scope and the characteristics of the subject systems in terms of number of public releases and size (*i.e.,* number of classes, methods, and KLOC).

As for the code smell types, we took into account 13 design flaws from the catalogs by Fowler [9] and Brown *et al.* [8]. In Table 2 we report the set of code smells analyzed together with a short description. We focused our attention on a mixed set of code smells. For instance, we considered *Blob* classes, *i.e.,* low-cohesive classes having a large number of methods and dependencies with data classes [9], but also other smells indicating violations of the Object-Oriented programming principles such as the *Feature Envy*, a method having more dependencies with another class with respect to the one it is actually in.

Although several other smells have been defined in the literature [8, 9], we selected this subset of smells because (i) they relate to different types of design issues, and (ii) they are the smells that have been more extensively studied in the past [11, 22, 31].

It is important to note that both the source code and the information about code smell instances of the software projects considered in this study come from a publicly available dataset that we built in a previous work [29][1]. Specifically, the dataset contains 40,888 **manually validated** instances of the 13 code smells analyzed in the study across the 395 releases of the 30 subject systems. More in detail, the construction of the dataset consisted of three steps.

In the first place, a simple tool that discarded the classes/methods that surely do not contain code smells was developed with the aim of easing the manual validation phase by excluding the analysis of those code elements that are clearly non-smelly. Specifically, for each code smell considered, the tool analyzed the metric profile of classes/methods and outputs a list of code elements to further analyze manually. For instance, when analyzing the *Class Data Should Be Private*, we filter out all the classes having no public attributes because they cannot be affected by the considered smell. For sake of completeness, we report the list of rules adopted in the filtering phase in our online appendix [33]. Once concluded the first phase, two Master students (*i.e.,* the *inspectors*) individually analyzed and classified the code elements of each system as true positive or false positive for a given smell. The output consisted of a list of smells identified by each inspector. In the second step, the produced oracles were compared, and the inspectors discussed the differences, *i.e.,* smell instances identified by one inspector but not by the other. All the instances positively classified by both the inspectors have been considered as actual smells. As for the others, the inspectors opened a discussion to resolve the disagreement and taking a

[1]The dataset used in this paper is **not** the same as the one we built when introducing the Landfill platform [32].

shared decision. The final output consisted of a unique list of smells that we used to answer our research questions. It is important to note that we relied on a manually-built dataset, because existing code smell detectors generally try to find a good compromise between the precision and the completeness of the recommendations, which lead to output a number of false positive instances as well as to miss some true negatives. A manual validation, instead, is supposed to be more accurate. The list of code smell instances affecting each release of the analyzed systems is available in our online appendix [33].

### *2.2. Data Analysis*

To answer our preliminary research question ($\mathbf{RQ}_0$) we computed the number of code smells affecting each class. Then, we reported the percentage of classes affected by one or more types of code smell instances.

As for $\mathbf{RQ}_1$, we investigated how often the presence of a code smell of a given type (*e.g.,* a *Feature Envy*) in a source code class implies the presence of another code smell of a different type (*e.g.,* a *Blob*). Specifically, for each code smell type $cs_i$ we computed the percentage of times its presence in a class/method co-occurs with the presence of another code smell type $cs_j$. Formally, for each pair of code smell types $cs_i$ and $cs_j$ we computed the percentage of co-occurrences of $cs_i$ and $cs_j$ using the following formula:

$$\textit{co-occurrences}_{cs_{i,j}} = \frac{|cs_i \wedge cs_j|}{|cs_i|}, \text{ with } i \neq j$$

where $|cs_i \wedge cs_j|$ is the number of co-occurrences of $cs_i$ and $cs_j$ and $|cs_i|$ is the number of occurrences of $cs_i$. Note that $\textit{co-occurrences}_{cs_{i,j}}$ differs from $\textit{co-occurrences}_{cs_{j,i}}$ since the formula's denominator changes from $|cs_i|$ to $|cs_j|$.

To answer $\mathbf{RQ}_2$, we mined the releases of the subject systems looking for temporal relationships between the introduction of two or more instances of different code smell types in subsequent versions of the same class. In particular, we analyzed the percentage of times a code smell type (*e.g., Long Method*) is introduced temporally after another code smell type (*e.g., Spaghetti Code*) using the following formula:

$$\textit{introduction\_pattern}_{cs_{i,j}} = \frac{|cs_i \rightarrow^{intro} cs_j|}{|cs_i \wedge cs_j|}, \text{ with } i \neq j$$

where $|cs_i \rightarrow^{intro} cs_j|$ is the number of times a smell instance of type $cs_i$ was introduced in a previous release with respect to a smell instance of type $cs_j$ and $|cs_i \wedge cs_j|$ is the total number of times instances of types $cs_i$ and $cs_j$ appeared in the same class over the analyzed releases. Note that also in this case $\textit{introduction\_pattern}_{cs_{i,j}}$ differs from $\textit{introduction\_pattern}_{cs_{j,i}}$ since the formula's numerator changes from $|cs_i \rightarrow^{intro} cs_j|$ to $|cs_j \rightarrow^{intro} cs_i|$. Following this procedure, we were able to track the history of each code smell co-occurrence in our dataset and describe the *common introduction patterns.*

Finally, to answer **RQ**$_3$ we conducted a complementary analysis to the one presented in **RQ**$_2$. In particular, we analyzed the percentage of times a code smell type (*e.g., Long Method*) is removed before another code smell type (*e.g., Blob*) from a class in which they co-occur. In this case, we used the following formula:

$$\textit{removal\_pattern}_{cs_{i,j}} = \frac{|cs_i \rightarrow^{rem} cs_j|}{|cs_i \wedge cs_j|}, \text{ with } i \neq j$$

where $|cs_i \rightarrow^{rem} cs_j|$ is the number of times a smell instance of type $cs_i$ was removed in a previous release with respect to a smell instance of type $cs_j$ and $|cs_i \wedge cs_j|$ is the total number of times instances of types $cs_i$ and $cs_j$ appeared in the same class over the considered releases. So, we analyzed whether there exists a sequence of changes aimed at removing code smell co-occurrences. We call such sequences *common removal patterns*. It is worth noting that we considered a code smell instance as removed if the dataset reports the presence of the smell in a release $r_i$ and its absence in a subsequent release $r_{i+1}$. For this reason, we only observed whether a code smell was not present anymore in the release $r_{i+1}$, *i.e.,* we did not know if that code smell instance was removed because of a specific refactoring operation performed by developers.

Finally, to statistically investigate the significance of the identified introduction (**RQ**$_2$) and removal (**RQ**$_3$) patterns, we adopted the *Granger causality test* [34] to determine whether one time series is useful in forecasting another. In other words, we tested whether the presence of a code smell type $cs_i$ can be used to "predict" the future introduction of another code smell type $cs_j$. Note that in this study we used the *Granger* test instead of association rule discovery [35], because we are interested in assessing the statistical significance of the temporal consequence of different kinds of smells, rather than of just smell co-occurrences. Since we performed multiple Granger tests between different smell types, we adjusted $p$-values using the Holm's correction procedure [36]. In particular, the more hypotheses we check the higher the probability of a Type I error (*i.e.,* false positive). The Holm's method aims at controlling the probability that one or more Type I errors will occur by adjusting the rejection criteria of each of the individual hypotheses. The procedure firstly sorts the $p$-values resulting from $n$ tests in ascending order of values, multiplying the smallest $p$-value by $n$, the next by $n - 1$, and so on. Then, each resulting $p$-value is then compared with the desired significance level (*e.g.,* 0.05) to determine whether or not it is statistically significant.

## 3. Analysis of the Results

This section discusses the achieved results with the goal of answering the four formulated research questions. Specifically, we will present the findings for **RQ**$_0$ in Section 3.1, while the results for **RQ**$_1$, **RQ**$_2$, and **RQ**$_3$ are presented together in Section 3.2 to facilitate discussion of the results and avoid redundancies.

Table 3: **RQ$_0$**: Percentage of smelly classes affected by one, two, and three code smell instances simultaneously.

| Category | # | % |
|---|---|---|
| Classes affected by one smell | 16,765 | 41% |
| Classes affected by two smells | 12,675 | 31% |
| Classes affected by three smells | 11,448 | 28% |

## *3.1. RQ$_0$: Diffuseness of code smell co-occurrences*

Table 3 reports the code smell co-occurrences we found in our dataset. Note that we found only classes affected by one to three different types of code smells simultaneously.

Overall, the results in Table 3 highlight that the phenomenon is not negligible. Indeed, while 41% of the smelly classes are affected by a single code smell instance, the remaining 59% of the smelly classes in our dataset are affected by two or three different types of code smell instances. Specifically, 31% of the smelly classes are affected by two smells, while a co-occurrence of three smells was found in 28% of the smelly classes. The results are consistent among all the studied systems. Specifically, we did not find a high variability in the percentages reported above: the standard deviation is ≈5%, ≈3%, and ≈3% for classes affected by one, two, and three smells, respectively.

As a more general consideration, if we consider recent findings on the harmfulness of code smell co-occurrences [29], *i.e.,* classes affected by more than one smell are up to 350% more change-prone and 100% more fault-prone than classes affected by a single smell instance, our results reveal that there is a high risk of appearance of code smell co-occurrences that might considerably worsen software systems maintainability. One interesting example of smells co-occurrence was found in the Apache Struts project, an open source framework for the development of Java EE web applications. Specifically, in version 2.2.3 a *Long Method* instance was introduced in the method `parseStandardAction` belonging to the class `jasper.compiler.JspDocumentParser`. Such a method was introduced in that version and its main responsibility is the parsing of `HTTP` requests made on the Web server. In the subsequent version (*i.e.,* 2.2.3.1) two other smells appeared in the same class. In particular, the class became a *Blob* due to the implementation of new methods aimed at deploying a Web application and resulting in a reduction of the class cohesion (one of the characteristics of *Blob* classes). In addition, the number of dependencies with other classes of the system increased sensibly, also because of the introduction of a *Feature Envy* instance affecting the method `startElement`.

Interestingly, during the history of the system this class was involved in several bug-fixing activities, as reported in the project issue tracker[2]. For instance, when committing a change to this class in the repository, a developer posted the commit message shown below:

[2]`http://tinyurl.com/hcjqygc`

*"Class impossible to touch!*
*This code will never be fine!"*

This example seems to confirm that developers experience some difficulties when modifying classes affected by several code smells, as highlighted in previous work by Yamashita and Moonen [22].

On the one hand, our results reinforce the need for a deep investigation into the factors leading to the introduction and removal of co-occurring smells. On the other hand, they highlight the importance of (i) empirical studies aimed at analyzing the effects of the interaction of smells [21, 22, 30] and (ii) devising *interaction-aware* code smell detectors and prioritizers: in particular, knowing which code smells tend to co-occur can be a further piece of information (*e.g.,* besides the classic code quality metrics used by existing detection tools [25, 37]) that can be used not only to identify code smells (*e.g.,* knowing that a class is affected by the Message Chain smell increases its likelihood of being affected by other smells), but also to predict and warn developers about the future introduction of other smell types, before they even appear in the system.

**Finding 1.** The phenomenon of code smell co-occurrence is highly diffused. In a dataset containing 40,888 instances of 13 different code smells identified across the 395 releases of 30 software systems, we observed that 24,124 of the smelly classes (59%) are affected by more than one type of code smell.

### *3.2. $RQ_{1-3}$: Understanding the lifecycle of code smell co-occurrences*

Tables 4, 5 and 6 report the results of the analysis carried out to answer $\mathbf{RQ}_1$, $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$, respectively. Specifically, Table 4 reports the co-occurrences of the 13 analyzed code smell types in the 395 subject software releases. To facilitate the reading of the table, we reported in bold face the code smell type pairs co-occurring in at least 10% of cases (percentages in Table 4 are computed following the process described in Section 2).

Table 5 reports, for each pair of frequently co-occurring smell types found in the context of $\mathbf{RQ}_1$ (columns "A" and "B" in Table 5), (i) the percentage of times the code smell type "A" has been introduced before the code smell type "B" (column "A $\rightarrow$ B"), (ii) the percentage of times the code smell type "B" has been introduced before the code smell types "A" (column "B $\rightarrow$ A"), and (iii) the percentage of times the code smells have been introduced in the same release.

Finally, Table 6 reports, for each pair of frequently co-occurring smell types, (i) the percentage of times the code smell type "A" has been removed before the code smell type "B" (column "A $\rightarrow$ B"), (ii) the percentage of times the code smell type "B" has been removed before the code smell types "A" (column "B

Table 4: **RQ**$_1$: Co-occurrences of code smells in the 395 analyzed releases of the studied projects. Absolute numbers reported in parenthesis.

| i/j | CDSBP (3,526) | CC (2,073) | FE (2,409) | BL (1,663) | II (1,220) | LC (1,061) | LM (14,802) | LPL (4,100) | MC (130) | MM (250) | RB (2,479) | SC (3,875) | SG (3,330) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CDSBP (3,526) | | 0% | **14% (493)** | 0% | 1% (48) | 0% | 5% (176) | 3% (106) | 0% | 0% | 0% | 0% | 0% |
| CC (2,073) | 0% | | 3% (62) | 2% (41) | 1% (21) | 0% | 7% (145) | 3% (62) | 2% (41) | 0% | 0% | 1% (21) | 0% |
| FE (2,409) | **20% (493)** | 3% (62) | | 6% (149) | 2% (49) | 0% | **18% (433)** | 1% (24) | 0% | 0% | 3% (72) | 3% (72) | 0% |
| BL (1,663) | 0% | 3% (41) | 9% (149) | | 0% | 0% | 5% (83) | 3% (50) | 1% (17) | 0% | 0% | 1% (17) | 0% |
| II (1,220) | 4% (48) | 2% (21) | 4% (49) | 0% | | 0% | 8% (98) | 3% (37) | 1% (12) | 0% | 1% (12) | 1% (12) | 0% |
| LC (1,061) | 0% | 0% | 0% | 0% | 0% | | 1% (11) | 1% (11) | 0% | 0% | 0% | 0% | 0% |
| LM (14,802) | 1% (176) | 1% (145) | 3% (433) | 1% (83) | 0% | 0% | | 0% | 0% | 0% | 0% | 1% (148) | 0% |
| LPL (4,100) | 3% (106) | 2% (62) | 0% | 1% (50) | 1% (37) | 0% | 0% | | 0% | 0% | 1% (41) | 1% (41) | 0% |
| MC (130) | 0% | **32% (41)** | 0% | **13% (17)** | 9% (12) | 0% | 0% | 0% | | 0% | **35% (46)** | **17% (22)** | 0% |
| MM (250) | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | | 0% | 0% | 0% |
| RB (2,479) | 0% | 0% | 3% (72) | 0% | 0% | 0% | 0% | 1% (41) | 2% (46) | 0% | | 0% | 0% |
| SC (3,875) | 0% | 1% (21) | 2% (72) | 0% | 0% | 0% | **38% (148)** | 1% (41) | 0% | 0% | 0% | | 0% |
| SG (3,330) | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | |

In **bold** the co-occurrences > 10%

Table 5: **RQ**$_2$: Common patterns in the introduction of code smell co-occurrences.

| A | B | A $\rightarrow$ B | B $\rightarrow$ A | A == B |
|---|---|---|---|---|
| Message Chains | Spaghetti Code | 69% | 11% | 20% |
| Message Chains | Complex Class | 64% | 26% | 10% |
| Message Chains | Blob | 53% | 22% | 25% |
| Message Chains | Refused Bequest | 12% | 10% | 78% |
| Long Method | Spaghetti Code | 82% | 4% | 14% |
| Long Method | Feature Envy | 34% | 26% | 40% |

$\rightarrow$ A"), and (iii) the percentage of times the smells were removed in the same version.

As it is possible to see in Table 4, there exist few pairs of code smell types frequently co-occurring. Among these, the *Message Chains* smell is the one that tend to co-occur more with other code smell types, and in particular with *Complex Class* (*i.e.,* 32% of classes affected by *Message Chains* are also affected by *Complex Class*), *Refused Bequest* (35%), *Spaghetti Code* (17%), and *Blob* (13%). The *Message Chains* smell appears when the invocation of a method results in a subsequent long chain of other method invocations. In the context of our study, we found several cases in which such smelly methods were part of complex/long classes.

Concerning the relationship between *Message Chains* and *Complex Class*, we observed that complex methods belonging to the *Complex Class* are often the ones responsible for triggering long chains of method calls.

Table 6: **RQ**$_3$: Common patterns in the removal code smell co-occurrences.

| A | B | A → B | B → A | A == B |
|---|---|---|---|---|
| Message Chains | Spaghetti Code | 7% | 10% | 83% |
| Message Chains | Complex Class | 11% | 14% | 75% |
| Message Chains | Blob | 5% | 6% | 89% |
| Message Chains | Refused Bequest | 14% | 20% | 66% |
| Long Method | Spaghetti Code | 23% | 4% | 73% |
| Long Method | Feature Envy | 33% | 6% | 61% |

A clear example involves the Apache Ant 1.8 project. The class `FTP` belonging to the package `ant.taskdefs.optional` implements a basic FTP client that can send, receive, list, and delete files, and create directories. The class is affected by the *Complex Class* smell, and indeed its McCabe's cyclomatic complexity [38] is 64. At the same time, the method `accountForIncludedDir` is affected by a *Message Chain* smell because it recursively invokes four different methods belonging to different classes, thus triggering a long chain of calls.

Looking at Table 5, we can observe that in 69% of the cases a *Message Chains* smell is introduced before a *Spaghetti Code* instance. This may indicate that the introduction of a *Message Chains* increases the complexity of a class, which will be subsequently affected by a *Spaghetti Code* smell. To verify this conjecture, we firstly analyzed the number of releases occurring between the introduction of the first and of the second smell, finding that the *Spaghetti Code* instance is generally introduced in the subsequent release after *Message Chains* smell is introduced. For example, the class `ant.taskdefs.optional.FTP` mentioned above was affected by the *Message Chains* smell in the version 1.7, while in the version 1.8 the class was affected by the *Spaghetti Code* smell. During these two versions, the McCabe metric increased from 33 to 64, and the method that experienced more changes was `accountForIncludedDir` (*i.e.,* the method affected by the *Message Chains*). This result seems to confirm previous finding by D'Ambros *et al.* [31] about the harmfulness of the *Message Chains* smell.

The results of the Granger causality test are shown in Table 7, which reports the adjusted $p$-values obtained by the test when testing the hypotheses: (i) code smell type "A" triggers the introduction of code smell "B" (column "A → B"), and (ii) code smell type "B" triggers the introduction of code smell "A" (column "B → A"). As it is possible to see, the causality between the introduction of *Message Chains* and *Complex Class* instances is confirmed also by the statistical test. Other cases confirmed by statistical tests are *Message Chains → Spaghetti Code* and *Message Chains → Blob.*

The existence of relationships between *Message Chains* and *Blob*, and between *Message Chains* and *Spaghetti Code* seems to delineate a clear trend in the results. Indeed, similarly to what observed for the *Complex Class*, in most of the cases a *Message Chains* instance is introduced before a smell involving the entire class (see Table 5). This may indicate that design issues occurring in methods could degenerate in code smells involving the whole class. A relevant

Table 7: **RQ**$_2$: Granger Causality Test Results (adjusted $p$-values). Statistically significant $p$-values are reported in **bold** face.

| A | B | A → B | B → A |
|---|---|---|---|
| Message Chains | Spaghetti Code | <**0.001** | 1.00 |
| Message Chains | Complex Class | <**0.001** | 1.00 |
| Message Chains | Blob | **0.009** | 1.00 |
| Message Chains | Refused Bequest | 1.00 | 1.00 |
| Long Method | Spaghetti Code | <**0.001** | 1.00 |
| Long Method | Feature Envy | 1.00 | 1.00 |

Table 8: **RQ**$_3$: Granger Causality Test Results (adjusted $p$-values). Statistically significant $p$-values are reported in **bold** face.

| A | B | A → B | B → A |
|---|---|---|---|
| Message Chains | Spaghetti Code | 1.00 | 1.00 |
| Message Chains | Complex Class | 1.00 | 1.00 |
| Message Chains | Blob | 1.00 | 1.00 |
| Message Chains | Refused Bequest | 1.00 | 1.00 |
| Long Method | Spaghetti Code | 1.00 | 1.00 |
| Long Method | Feature Envy | 1.00 | 1.00 |

example concerns the class `utils.regex.RegexParser` of the APACHE XERCES project. In the version 1.4.1 a *Message Chains* instance was introduced in the method `processStar`. From that moment, the class underwent 14 changes that lead to the introduction of a *Blob* instance.

The discussion is different when analyzing the relationship between *Message Chain* and the *Refused Bequest* code smells. Classes affected by a *Refused Bequest* override most of the methods they inherit from their superclass(es). By analyzing some of the co-occurrences present in our dataset, we found that classes affected by *Refused Bequest* implement a higher number of methods with respect to non-smelly classes (14 *vs* 6). As a consequence such classes have a higher probability to contain methods affected by *Message Chains.* Basically, it seems that this result does not imply a causality relationship. This is somehow confirmed by the results achieved when analyzing the introduction and removal patterns (see Tables 5 and 6): *Message Chains* and *Refused Bequest* instances are generally introduced and removed together.

Indeed, the Granger test (Tables 7 and 8) does not show a temporal relationship between these smells (*i.e.,* one does not temporally correlate with the introduction/removal of the other). An example is represented by the class `bsh.BshClassManager` of the JEDIT project. In version 4.4.1, 14 of the 17 inherited methods of this class have been overridden, thus refusing the bequest of the parent `ClassManagerImpl`.

In the same version, in the (overridden) method `loadSourceClass` a *Message Chains* instance was introduced because of a long chain of method invocations needed to monitor and handle certain Java version-dependent classes.

Other co-occurrences we found were quite expected. For instance, the one related to the *Spaghetti Code* and the *Long Method* code smells. By definition, a *Spaghetti Code* is a class implementing several complex methods interacting between them, with no parameters and using global variables [9]. Given the well-known relationship between size (method length) and complexity, it is reasonable to think that the complex methods present in a class affected by the *Spaghetti Code* smell are *Long Method* instances too. Analyzing the way the two smells are introduced (see Table 5), we found that in most cases *Long Method* instances precede the introduction of *Spaghetti Code* instances. This is perfectly in line with the definition of the *Spaghetti Code* smell. The likely introduction of a *Spaghetti Code* as a consequence of the introduction of a *Long Method* instance is confirmed by the Granger causality test.

Finally, another interesting case of co-occurrence is the one between the *Feature Envy* and the *Long Method* code smells. *Long Method* instances are generally composed of several code statements. In such methods, the likelihood to have dependencies toward other classes may be higher than the one of other non-smelly methods. Therefore, *Long Method* instances may also be involved in the *Feature Envy* smell. In this case, Table 5 shows that 34% of the co-occurrences started with the introduction of a *Long Method* instance. This seems to indicate that developers working on long methods are more likely to add statements in which external classes are invoked, thus increasing the chances of introducing a *Feature Envy*. However, the results of the Granger test do not support such a conclusion (see Table 7).

When considering the way the smells are removed (Table 6), we observed that in almost all the cases (83%) the code smells co-occurrence was resolved by removing all the smells at the same time. This frequent pattern may indicate that the code smells removal could be a consequence of other maintenance activities causing the deletion of the affected code components (with a consequent removal of the code smell instances), as well as the result of a major restructuring or scheduled refactoring actions [17, 15].

For example, the `ant.taskdefs.optional.FTP` class was completely restructured in the version 1.8.3 because of several changes to the way the FTP client was implemented, as documented in the release notes[3]. This resulted in the removal of all its smell instances.

The miss of significant temporal relationships for what concerns the removal of these co-occurring code smell types is also confirmed when analyzing the statistical causality test (see Table 8).

[3]http://tinyurl.com/hdej8xx

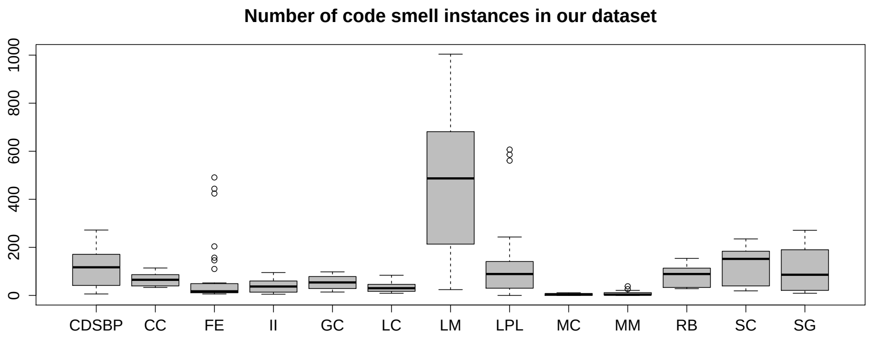

Figure 1: Absolute number of code smell instances in the analyzed systems.

**Finding 2.** We found six code smell pairs frequently co-occurring. In most cases, code smells affecting source code methods correlate with the introduction of other smells affecting the entire class. Finally, we found that code smell co-occurrences are generally removed together as a consequence of maintenance activities causing the deletion of the affected code components (with a consequent removal of the code smell instances) as well as the result of a major restructuring or scheduled refactoring actions.

## 4. Threats to Validity

Threats to *construct validity* are related to the relationship between theory and observation. In the context of this study, they are mainly due to imprecisions/errors in the performed measurements. The code smell instances considered in this study have been previously manually validated, thus providing a high confidence in the reported findings [29]. However, we cannot exclude that the oracle we used misses some smells, or else includes some false positives.

Threats to *internal validity* concern factors internal to our study design that could have affected our conclusions. In particular, while we can claim—using Granger's causality test—a statistical causality between the introduction of a smell and another on the same code component, we cannot tell whether, indeed, one smell caused the other or, instead, other phenomena related to the project evolution caused the introduction of both smells. We partially back-up such a threat by conducting some qualitative analysis.

Threats related to the relationship between the treatment and the outcome (*conclusion validity*) are related to our analysis method.

As also done in other papers related to code smells co-occurrence [39, 40], the analysis has been conducted at release level. Another threat in this category might be related to the high diffuseness of instances of a certain code smell type. Indeed, frequent co-occurrences between the considered smells might simply be the results of the high diffuseness of one smell type (*e.g., Message Chains*), possibly indicating no causation in the observed relationships. Fig. 1 shows box plots depicting the distribution of code smell instances for each type present in the dataset [29]. Due to space limitations, we only report the overall diffuseness of code smells in the 30 studied systems. As it is possible to notice, all the code smells are almost equally distributed and, therefore, the observed co-occurrences should not be just the result of the high diffuseness of single code smell types. Moreover, one of the smells more involved in co-occurrences (*i.e.,* the *Message Chains*) is also one of the less widespread. To further strengthen our conclusion validity, we have complemented the reported quantitative analysis with a qualitative discussion of some interesting cases.

Finally, regarding the generalization of our findings (*external validity*) this is, to the best of our knowledge, the largest study—in terms of number of software releases (395) and considered code smell types (13)—concerning the analysis of code smell co-occurrence. However, we are aware that we limited our attention only to Java systems. Also, although the 13 code smell types are well representative of the types of smells investigated in past literature on smell evolution (and as said this set is larger than any previous study in this area), we cannot exclude co-occurrence phenomena with smells that were not considered in our work. For this reason, the generalizability of our results is clearly limited to the 13 smell types we considered. Further studies aiming at replicating our study on systems written in other programming languages are therefore highly desirable.

## 5. Related Work

The research community has extensively studied the code smell phenomenon under different perspectives. Several papers focused on how to detect and prioritize code smells [25, 37, 28, 23, 26], while others analyzed how code smells are diffused [17, 15], how they are introduced [13, 12], what is their impact on non-functional attributes of source code [31, 30, 10, 41], and how developers perceive and work on classes affected by code smells [11, 22, 21, 20].

Due to the nature of this study, in the following we focus on the works investigating the characteristics of code smells. A complete overview of automated techniques for smell detection is available in a recent survey by Fernandes *et al.* [42].

Several studies considered the way the code smells evolve during the evolution of a system. The study by Chatzigeorgiou and Manakos [15] showed that (i) the number of instances of code smells increases during time; and (ii) developers are reluctant to perform refactoring operations in order to remove them. On the same line, the results reported by Peters and Zaidman [17] show that developers are often aware of the presence of code smells in the source code, but they do not invest time in performing refactoring activities aimed at removing them. A partial reason to this behavior is given by Arcoverde *et al.* [14], who studied the longevity of code smells showing as they often survive for a long time in the source code. The authors point to the will of avoiding changes to API as one of the main reason behind this result [14]. Palomba *et al.* [43] studied the evolution of code smells detected using different types of information, *i.e.,* structural and textual, finding the design problems detected using textual-based tools tend to be more maintained and refactored than the ones detected using structural-based tools.

Recently, Tufano *et al.* [13] investigated when code smells are introduced by developers, and the circumstances and reasons behind their introduction. They showed that most of the times code artifacts are affected by smells since their creation, and developers introduce them not only when they are implementing new features or enhancing existing ones, but also, in some cases, during refactoring operations. To the best of our knowledge the empirical study reported in this paper is the first one investigating the diffuseness and the lifecycle of code smell co-occurrences.

The research community has been also active in investigating the impact of code smells on maintenance activities. Sjoberg *et al.* [20] investigated the impact of twelve code smells on the maintainability of software systems, finding that smells do not always constitute a problem, and that often class size impacts maintainability more than the presence of smells.

Lozano *et al.* [16] proposed the use of change history information to better understand the relationship between code smells and design principle violations, in order to assess the causes for which a code smell appears. Deligiannis *et al.* [44] also performed a controlled experiment showing that the presence of *Blob* smell negatively affects the maintainability of source code. Also, the authors highlight an influence played by these smells in the way developers apply the inheritance mechanism.

Recently, Palomba *et al.* [11] investigated how the developers perceive code smells, showing that smells characterized by long and complex code are those perceived more by developers as relevant problems.

A consistent body of research on code smells is represented by studies analyzing the relationships between design flaws and code maintainability. In this context, Khomh *et al.* [10] showed that the presence of code smells increases the code change proneness. Also, they showed that the code components affected by code smells have a higher fault-proneness with respect to components not affected by any smell [45]. These results were confirmed by Palomba *et al.* [29], however they found that refactoring code smells does not always help in reducing the change- and fault-proneness of classes. Gatrell and Counsell [41] conducted an empirical study aimed at quantifying the effect of refactoring on change- and fault-proneness of classes. Their study revealed that classes subject to refactoring have a lower change- and fault-proneness, independently of whether during the observation period classes were subject to refactoring or not. Li *et al.* [46] empirically evaluated the correlation between the presence of code smells and the probability that the class exhibits faults. They studied the post-release evolution process showing that many code smells are positively correlated with the class faults-proneness. Olbrich *et al.* [47] conducted a study on the *Blob* and *Brain Class* code smells, reporting that these code smells were changed less frequently and had a fewer number of defects with respect to the other classes.

D'Ambros *et al.* [31] also studied the correlation between the *Feature Envy* and *Shotgun Surgery* smells and the defects in a system, reporting no consistent correlation between them.

All the studies mentioned above have been performed by considering single smell instances occurring in a code component, without explicitly considering code smell co-occurrences.

To overcome this limitation, Abbes *et al.* [30] studied the impact of two types of code smells, namely *Blob* and *Spaghetti Code*, on program comprehension. Their results show that the presence of a code smell in a class does not have an important impact on the developers' ability to comprehend the code. Instead, a combination of more code smells affecting the same code components strongly decreases the developers' ability to deal with comprehension tasks. The interaction between different smell instances affecting the same code components has also been studied by Yamashita *et al.* [21, 22], who confirmed that developers experience more difficulties in working on classes affected by more than one code smell. Yamashita *et al.* [48] also presented a replicated study where they analyzed the problem of code smell interaction in both open and industrial systems, finding that the relation between smells vary depending on the type of system taken into account. Our study is complementary to the ones above, since it investigates how code smell co-occurrences are introduced and removed by developers.

As for studies investigating which types of code smells tend to co-occur together in production code, Anubhuti *et al.* [40] studied the co-occurrences of 7 code smell types in two large open source projects, Chromium and Mozilla. They evaluated the percentage of smells co-occurring over the change history of such projects, finding that often traditional code smells (*i.e.,* Feature Envy and Data Clumps) co-exist together with code duplication. Unlike this work, we studied a larger number of systems (*i.e.,* 30 instead of 2) and a larger number of code smells types (*i.e.,* 13 instead of 7).

Arcelli Fontana *et al.* [39] studied the phenomenon of code smell co-occurrence by counting the percentage of smells appearing in the same class during the history of the software projects. By relying on the Qualitas Corpus [49] dataset, Arcelli Fontana *et al.* found that only in a small percentage of cases (*i.e.,* 3% of average) a *Brain Method* co-occur with other smells as *Dispersed Coupling* and *Message Chains*. Our study has been carried out considering a larger number of smells, and highlight several other co-occurrences between code smells.

Palomba *et al.* [50] applied association rule discovery to identify frequent co-occurrences between 13 smell types, finding six code smell pairs that frequently appear together in open-source software systems. Besides analyzing the co-occurrences between smells, the proposed study provides insights on (i) the extent of the code smell co-occurrence phenomenon and (ii) how such co-occurrences are introduced and removed by developers. Therefore, our study is complementary to the work by Palomba *et al.* [50], as it provides a deeper investigation into the phenomenon.

Lozano *et al.* [51] reported a preliminary analysis of the co-occurrences between four code smells in three systems. They evaluated whether specific smells tend to co-occur together and whether they are removed in the same moment. While this investigation represents the closest study with respect to the work reported in this paper, we believe that our analysis goes beyond previous research for two main reasons: in the first place, we consider a large number of code smells having different characteristics on a much larger set of software systems and evolution history (30 vs 3).

On the other hand, our study provides a comprehensive overview of the co-occurrence phenomenon, by providing hints about (i) the diffuseness of the phenomenon, (ii) the co-introduction of smells, (iii) the co-removal, and (iv) more importantly, studying the Granger's causality between the introduction and removal events.

In a closely related area of research, Tufano *et al.* [12] conducted an empirical investigation of the co-occurrence between test [52] and code smells in the context of a more general investigation into the nature of test smells. The results showed that some test and production smells are generally related, as in the case of *Assertion Roulette* and *Spaghetti Code* [12].

## 6. Conclusion

Code smells are symptoms of poor design or implementation choices. As a form of technical debt, code smells can negatively affect code maintainability [10]. Despite the effort devoted by the research community in investigating the code smells phenomenon, few studies targeted the problem of code smell co-occurrences.

In this paper we presented a large-scale empirical investigation into the nature of code smell co-occurrences, by providing evidence on (i) the extent to which code smells co-occur, (ii) the types of code smells that often co-occur, and (iii) how code smells co-occurrences are introduced and removed by developers.

Our study provided four main findings:

- **The phenomenon of code smell co-occurrences is highly spread.** We observed that 59% of the smelly classes are affected by more than one code smell. Therefore, we can claim that the phenomenon is worth of studying and needs to be further investigated in future research.

- **There are six code smell types frequently co-occurring together.** We discovered six strong relationships between the considered smells. In particular, we observed that the *Message Chains* is the smell more frequently involved in classes affected by multiple design issues. Indeed, the smell is frequently associated with *Spaghetti Code*, *Complex Class*, *Blob*, and *Refused Bequest*, highlighting the likelihood of the smell to appear together with smells characterized by complex and long code. Moreover, our study revealed other two frequent co-occurrences, *i.e., Long Method–Spaghetti Code* and *Long Method–Feature Envy.* These relationships were quite expected because of the nature of the involved smells.

- **Method-level code smells may be the cause of class-level code smells.** Analyzing the patterns which introduced code smells, we found that often the presence of method-level smells (*e.g., Message Chains*) *"induces"* code smells affecting the entire class. This may be due to the fact that method-level smells increase the complexity of a class, making developers more prone to apply sub-optimal design choices in the whole class.

- **Co-occurring code smells tend to disappear together.** When mining the *frequent removal patterns*, we observed that smells co-occurrences are generally removed together as a consequence of other maintenance activities causing the deletion of the affected code components (with a consequent removal of the code smell instances) as well as the result of a major restructuring or scheduled refactoring actions [17].

Our findings have several implications for the research community:

1. **Co-occurrence-aware code smell detector.** The frequent co-occurrence between smells might represent an important source of information in the context of code smell detection. Indeed, co-occurrence information (*e.g.,* monitoring together the symptoms characterizing a frequent code smell pair) might be exploited to build more accurate code smell detectors able to identify the location and/or the severity of design problems affecting a class.
2. **Leveraging temporal analysis to predict code smell introduction.** The use of co-occurrence information can provide important benefits when predicting the future appearance of code smells, as the symptoms of a certain smell can effectively forecast the introduction of another smell. Thus, the research community might investigate the extent to which the introduction of code smells can be predicted using co-occurrence information.
3. **More attention to method-level code smells.** As highlighted in our study, design problems occurring at method-level can be the cause of infection for an entire class. As a consequence, we believe that the research community should investigate more the definition of appropriate method-level monitoring tools as well as method-level code smell detectors. Furthermore, our results might possibly indicate that refactoring operations performed on methods can avoid the introduction of coarse-grained code smells. Thus, the research community should pay more attention to refactoring techniques and tools able to work at method-level: indeed,

while some method-level techniques have been proposed over the years [53], they mainly focus on the identification of refactoring opportunities for code smells like *Feature Envy* [37, 54, 55], thus not specifically targeting code smells found as more dangerous in the context of code smell co-occurrences (*e.g., Message Chains*).

These findings represent the main input for our future research agenda. Specifically, we plan to design new code smell detectors able to suggest appropriate refactoring operations in presence of smell co-occurrences. Furthermore, we plan to further understand the phenomenon of code smell co-occurrences by analyzing the impact on maintainability given by the introduction of a second smell in a class that is already smelly.

## References

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

[2] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. L. Nord, I. Ozkaya, R. S. Sangwan, C. B. Seaman, K. J. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the Workshop on Future of Software Engineering Research, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE).* ACM, 2010, pp. 47–52.

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

[4] F. Shull, D. Falessi, C. Seaman, M. Diep, and L. Layman, *Perspectives on the Future of Software Engineering.* Springer, 2013, ch. Technical Debt: Showing the Way for Better Transfer of Empirical Results, pp. 179–190.

[5] W. Cunningham, "The WyCash portfolio management system," *OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[6] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)," in *Dagstuhl Reports*, vol. 6, no. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[7] C. Izurieta, I. Ozkaya, C. B. Seaman, P. Kruchten, R. L. Nord, W. Snipes, and P. Avgeriou, "Perspectives on managing technical debt: A transition point and roadmap from dagstuhl." in *QuASoQ/TDA@ APSEC*, 2016, pp. 84–87.

[8] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1$^{st}$ ed. John Wiley and Sons, March 1998.

[9] M. Fowler, *Refactoring: improving the design of existing code.* Addison-Wesley, 1999.

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

[11] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *In Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14), Victoria, Canada*, 2014, pp. 101–110.

[12] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 4–15.

[13] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[14] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in *Proceedings of the International Workshop on Refactoring Tools.* ACM, 2011, pp. 33–36.

[15] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology*, ser. QUATIC '10. IEEE Computer Society, 2010, pp. 106–115.

[16] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, ser. IWPSE '07. New York, NY, USA: ACM, 2007, pp. 31–34.

[17] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *European Conference on Software Maintenance and ReEngineering.* IEEE, 2012, pp. 411–416.

[18] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *8th European Conference on Software Maintenance and Reengineering (CSMR 2004), 24-26 March*

*2004, Tampere, Finland, Proceeding.* IEEE Computer Society, 2004, pp. 223–232.

[19] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs," *Commun. ACM*, vol. 36, no. 11, pp. 81–94, Nov. 1993.

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

[21] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *International Conference on Software Maintenance (ICSM).* IEEE, 2012, pp. 306–315.

[22] ——, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *International Conference on Software Engineering (ICSE).* IEEE, 2013, pp. 682–691.

[23] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings of the 9th International Conference on Quality Software.* Hong Kong, China: IEEE CS Press, 2009, pp. 305–314.

[24] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA.* IEEE Computer Society, 2004, pp. 350–359.

[25] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[26] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in *Proceedings of the* $14^{th}$ *Conference on Software Maintenance and Reengineering*, R. Capilla, R. Ferenc, and J. C. Dueas, Eds. IEEE Computer Society Press, March 2010.

[27] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *Software Engineering, IEEE Transactions on*, vol. 41, no. 5, pp. 462–489, May 2015.

[28] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[29] 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 study,” *Empirical Software Engineering*, p. to appear, 2017.

[30] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, “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*, ser. CSMR ’11. IEEE Computer Society, 2011, pp. 181–190.

[31] M. D’Ambros, A. Bacchelli, and M. Lanza, “On the impact of design flaws on software defects,” in *Quality Software (QSIC), 2010 10th International Conference on*, July 2010, pp. 23–31.

[32] F. Palomba, D. D. Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia, “Landfill: An open dataset of code smells with public evaluation,” in *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, 2015, pp. 482–485.

[33] 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 - online appendix,” 2017. [Online]. Available: http://www.mediafire.com/file/mzyr95cgmrbym19/dataset.zip

[34] C. W. J. Granger, “Investigating causal relations by econometric models and cross-spectral methods,” *Econometrica*, vol. 37, no. 3, pp. 424–438, 1969.

[35] R. Agrawal and R. Srikant, “Mining sequential patterns,” in *Data Engineering, 1995. Proceedings of the Eleventh International Conference on.* IEEE, 1995, pp. 3–14.

[36] S. Holm, “A simple sequentially rejective Bonferroni test procedure,” *Scandinavian Journal on Statistics*, vol. 6, pp. 65–70, 1979.

[37] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[38] T. McCabe, “A complexity measure,” *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[39] F. A. Fontana, V. Ferme, and M. Zanoni, “Towards assessing software architecture quality by exploiting code smell relations,” in *Software Architecture and Metrics (SAM), 2015 IEEE/ACM 2nd International Workshop on*, May 2015, pp. 1–7.

[40] A. Garg, M. Gupta, G. Bansal, B. Mishra, and V. Bajpai, *Do Bad Smells Follow Some Pattern?* Singapore: Springer Singapore, 2016, pp. 39–46.

[41] M. Gatrell and S. Counsell, "The effect of refactoring on change and fault-proneness in commercial c# software," *Science of Computer Programming*, vol. 102, no. 0, pp. 44 – 56, 2015.

[42] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '16. New York, NY, USA: ACM, 2016, pp. 18:1–18:12. [Online]. Available: http://doi.acm.org/10.1145/2915970.2915984

[43] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *Transactions on Software Engineering*, 2017.

[44] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, no. 2, pp. 129 – 143, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121203002401

[45] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France.* IEEE Computer Society, 2009, pp. 75–84.

[46] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, pp. 1120–1128, 2007.

[47] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *in: Intl Conf. Softw. Maint*, 2010, pp. 1–10.

[48] 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 *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 121–130. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2015.7332458

[49] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *Proc. 17th Asia Pacific Software Eng. Conf.* Sydney, Australia: IEEE, December 2010, pp. 336–345.

[50] F. Palomba, R. Oliveto, and A. De Lucia, "Investigating code smell co-occcurrence using association rule learning: A replicated study," in *Proceedings of the 2017 1st International Workshop on Machine Learning for Software Quality Evaluation*, ser. MaLTeSQuE'17, 2017.

[51] A. Lozano, K. Mens, and J. Portugal, "Analyzing code evolution to uncover relations," in *2015 IEEE 2nd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP)*, March 2015, pp. 1–4.

[52] A. van Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, 2001, pp. 92–95.

[53] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Recommending refactoring operations in large software systems," in *Recommendation Systems in Software Engineering.* Springer, 2014, pp. 387–419.

[54] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, 2014.

[55] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, "Recommending move method refactorings using dependency sets," in *Reverse Engineering (WCRE), 2013 20th Working Conference on.* IEEE, 2013, pp. 232–241.