Prevalence of Bad Smells in C# Projects

Amanda Lima Sabóia[1], Antônio Diogo Forte Martins[2], Cristiano Sousa Melo[2], José Maria Monteiro[2], Cidcley Teixeira de Souza[1] and Javam de Castro Machado[2]

[1]*Department of Computing, Federal Institute of Ceará, Fortaleza - Ceará, Brazil*

[2]*Department of Computing, Federal University of Ceará, Fortaleza - Ceará, Brazil*

Keywords: Bad Smell, Prevalence, C#.

Abstract: Bad smell can be defined as structures in code that suggest the possibility of refactoring. In object-oriented languages such as C# and Java, Bad Smells are heavily exploited as a way to avoid potential software failures. The presence of a high number of bad smells in a software project makes the system maintenance and evolution hard. So, identifying smells in code and refactoring them helps to improve and maintain software quality. Anti-patterns are considered inadequate programming practices, but not an error, they are bad solutions to recurring software problems. In this work, we propose an exploratory study on open source projects written in C# and published in GitHub. We empirically analyzed a total of 25 projects, studying the prevalence of Bad Smells, in a quantitatively and qualitatively manner, and their relationship in order to identify possible anti-patterns. Our results showed that implementation smells are the most common. Besides, some smells occur together, such as Missing Default and Unutilized Abstraction that are perfectly correlated, and ILS and IMN detected by association rules. Thus, the proposed study aims to assist software developers in avoiding future problems during the development of C# projects.

1 INTRODUCTION

In object-oriented languages such as C# and Java, software metrics have been used to provide developers with additional information about the software quality (Singh and Kahlon, 2011). According to (Pressman and Maxim, 2016), a metric lists the individual measurements found in the software. For example, the average number of errors found per unit test, providing quantitative measures to evaluate the quality of software projects.

On the other hand, bad smells have been used as a means to identify problematic classes in object-oriented systems. In recent decades, bad smells have received massive attention among software engineering researchers, as they can pinpoint symptoms that may, in the future, become significant problems (Fowler, 2018). According to (Sharma et al., 2017), the term bad smell indicates the presence of quality issues, primarily impacting the maintenance of a software system. Bad smell can degrade aspects of code quality, such as readability and mutability, and may lead to the introduction of software flaws. However, even if a bad smell does not directly represent a defect in the source code, as they are not technically incorrect and do not interfere with execution, they should not be ignored as they may cause future problems and compromise software quality.

In (Fowler, 2018), the authors have defined different types of bad smells that can be refactored. They defined refactoring as a process of changing the internal structure of a software system without changing functionality. Thus, bad smells are used as a means to identify problematic classes for refactoring in object-oriented systems (Li and Shatnawi, 2007).

The presence of a high number of bad smells in a software project makes the system maintenance and evolution hard. So, identifying smells in code and refactoring them helps to improve and maintain software quality (Sharma, 2017). Some bad smells indicate real code problems (for example, using a long parameter list makes it difficult to invoke methods), while other smells are possible symptoms of a problem. For instance, when a method has feature envy, it may indicate that the method is misplaced or that some pattern like Visitor is being applied (Fontana et al., 2012). Besides, in (Suryanarayana et al., 2014), the authors had noticed that some smells amplify the effect of other smells.

In this work, we describe an exploratory study on

Sabóia, A., Martins, A., Melo, C., Monteiro, J., Teixeira de Souza, C. and Machado, J.
Prevalence of Bad Smells in C Projects.
DOI: 10.5220/0009580204240431
In *Proceedings of the 22nd International Conference on Enterprise Information Systems (ICEIS 2020) - Volume 2*, pages 424-431
ISBN: 978-989-758-423-7

25 open source projects written in C# and published in GitHub. First, We have investigated the prevalence of Bad Smells through a quantitative analysis. Next, we have studied the correlation between Bad Smells. Then, we have analyzed the degree of co-occurrence between these smells. Finally, we tried to identify possible anti-patterns. We have noted that some smells occur more often than others and, based on this observation, some of them have been indicated as strong candidates for possible anti-patterns. Anti-patterns are bad solutions to recurring software problems. Importantly, this work is not intended to identify and catalog anti-patterns, but only to analyze some frequently occurring smells that are strong candidates for anti-patterns. Moreover, in later studies verify whether they really can be classified or not as anti-patterns. The main idea of this paper is to get useful information, such as correlations, association rules between C# Bad Smells, in order to assist software developers in avoiding future problems during the development of C# projects.

2 RELATED WORKS

Some authors have already studied the presence of bad smells in C# projects. In (Sharma et al., 2017), the authors studied the relationship between the occurrence of design smells and implementation smells, as well as the distribution of these smells in C# codes. Their studies showed that the density of smells and lines of code in the analyzed C# projects have no strong correlation, and the most frequently occurring smells are unutilized abstraction and magic number, i. e., it's an unexplained number used in an expression. Besides, they observed a high degree of correlation between the number of detected instances of design and implementation smells. In (Alenezi and Zarour, 2018), the authors analyzed six open source systems written in C# and checked whether bad smells are resolved while the software is evolving. They focused on monitoring the evolution of bad smells. They also noted that in most cases, bad smells persisted over several successive versions of the software and realized that smells are introduced into maintenance activities.

In addition, some authors have studied the presence of bad smells in databases and mobile applications. In (de Almeida Filho et al., 2019), they have studied the occurrence of bad smells in PL/SQL projects and the degree of co-occurrence between them. They observed that many smells have high correlation coefficients, above 0.9, and found bad smells in SQL that occur together. In (Mannan et al., 2016), the authors analyzed 500 open source applications developed on Android and 750 desktop applications written in Java, and later compared the instances of smells present in the android application code with the smells present in the java application code. They noted that in java applications, the code is dominated by two smells: external duplication and internal duplication. Meanwhile, android apps display a more diverse set of bad smells. In (Habchi et al., 2017), the authors performed a study related to the presence of bad smells in the IOS mobile application. They proposed a catalog of 6 IOS-specific smells identified in the official platform documentation and feedback from the developers. In addition, the authors also presented an adaptation of the PAPRIKA tool, a tool initially designed to detect smells in android applications, to detect smells in IOS applications.

Other authors have studied the impact of smells on dimensions such as bug-proneness and change-proneness. In (Palomba et al., 2016), the authors used earlier findings on bug-proneness to create a specialized bug prediction model for smelling classes. Results indicated that the accuracy of a bug prediction model increases by adding the intensity of bad smells as a predictor. In (Kaur et al., 2016), the authors extracted bad smells from a mobile application written in Java. They used machine learning techniques to predict software change-proneness using bad smells as predictor variables. They noted that bad smells are better predictors of change-proneness problems compared to object-oriented software metrics for balanced and unbalanced learning methods. In (Liu et al., 2018), the authors studied six open source projects written in Java with up to 60 versions and noted that: in most cases, smell-based metrics outperform CK baseline metrics in the change-proneness study. Furthermore, when used together, smell-based metrics are most effective at predicting change-proneness files.

This paper proposes an empirical analysis of the prevalence of bad smells in C# projects. Thus, the main idea of this research is to obtain useful information, such as correlations and rules of association between these smells. Afterward, analyzing which smells are candidates for possible anti-patterns, deserving future works special attention.

3 METHODOLOGY

We performed all experiments using only the back-end code from the latest releases of some open source projects written in C# public available on GitHub.

The methodology used in this article consists of

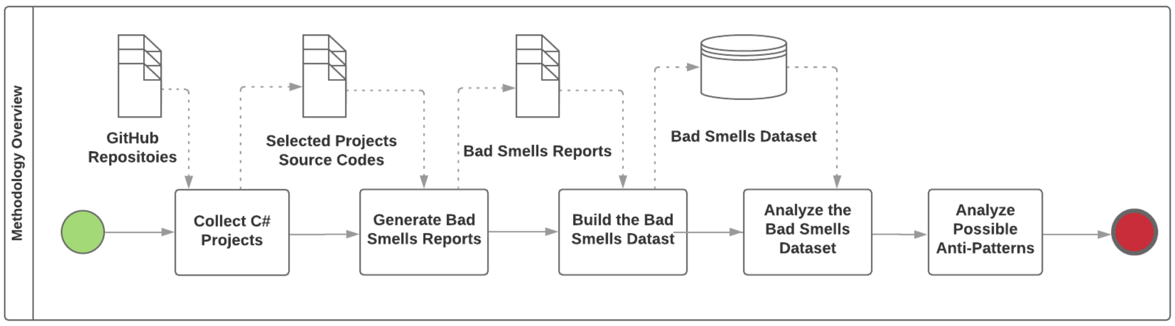
Methodology Overview
GitHub Repositoies
Selected Projects Source Codes
Bad Smells Reports
Bad Smells Dataset
Collect C# Projects
Generate Bad Smells Reports
Build the Bad Smells Datast
Analyze the Bad Smells Dataset
Analyze Possible Anti-Patterns

Figure 1: Diagram with the Methodology Overview.

five steps. The first step is to select some projects developed in C# and published in public repositories on GitHub. Subsequently, we applied a code analysis tool for C# projects called Designite to the selected projects. Designite generates a CSV file containing information about the bad smells found for each project. In the third phase, we developed Python scripts to process the CSV files generated by the tool and, later, we generated the datasets with the smells (a spreadsheet composed by a summary and aggregated information about the smells found in the projects). In the fourth phase, we applied a set of data analysis techniques to the dataset with the bad smells. In using these techniques, we use scripts in Python and Jupyter Notebook. Finally, we studied bad smells in isolation to analyze possible anti-patterns. Figure 1 contains an overview of the methodology used.

3.1 Collect C# Projects

First, we researched many projects developed in C# and published in public repositories on GitHub. These projects were selected based on the following criteria:

- The number of stars on GitHub, which represents the sum of developers who appreciate the repository;
- The number of forks in the repository, which shows the number of times the repository has been copied to private developer repositories;
- The percentage (%) of the project written in the C# language, which shows the percentage of the code implemented in C#. In this selection, only projects whose implementation in C# was greater than 80% were considered.

We have chosen twenty-five projects selected on GitHub. All information about these projects, such as name, description, the number of stars, the number of forks. and the percentage (%) of the code written in C# can be seen in our public repository[1].

[1]https://github.com/amandalsaboia/Bad-Smells-Projects

3.2 Report Generation with Bad Smells

We have used Designite (version 3.2.0.0) to perform the collection of Bad smells from C# projects, a tool that analyzes C# code and identifies quality problems for software. Based on granularity and scope, bad smells can be classified into architectural smells (creation nature), smells from design (structural nature), and smells implementation (behavioral nature) (Ganesh et al., 2013). Designite detects seven smells of architecture, eleven of implementation, and twenty smells of design, totaling thirty-eight bad smells. Also, this tool detects some object-oriented design (OO) metrics, such as the number of classes and the number of namespaces, in addition to detecting code clones. Some of these metrics were collected by the tool for the twenty-five projects analyzed in this article.

For each analyzed project, Designite generates three CSV files containing the bad smells found in each class. The first file contains the architectural smells (at the namespace level), the second contains the design smells (at the class level), and the last, the implementation smells (at the method level). Each file contains the location where a bad smell was found, and the cause of smell occurring. Although there is a difference in granularity between the types of bad smells, the tool associates the occurrence of smell with a class.

3.3 Construction of Datasets with Bad Smells

Each previously generated CSV file contains a considerable amount of bad smells in C#. For example, the Newtonsoft.Json project contains a total of 15534 bad smells. Since it is not recommended to extract smells manually, for the generation of datasets with the smells found in each project, we developed some Python scripts. In total, there were four scripts. In the first and second script, the data contained in the

CSV files were prepared with the smells of design and implementation, in which a column called Path containing the junction of Namespace and the Class where smell took place. The files with the architecture Smells used as path the value displayed in the column Namespace, and it is not necessary to create a new one. In the third script, the bad smells were added, which were only of architecture, then those that were only of the design and finally, those that were only of implementation. Finally, in the last script, we have constructed the dataset joining all bad smells (architectural, design, and implementation) found in the project. Each line of this dataset represents a class present in the analyzed project, and each column represents a bad smell. So, the value of a given cell means the number of occurrences of a given smell in a given class. We use these datasets in the data analysis performed by the project.

The creation of a dataset with the smells found in all projects happened similarly. It is only necessary to create one more Python script. This script is responsible for merging all datasets per project created previously. In this case, each row of the dataset with all smells represents a C# project, and each column represents a smell, in which the value of a given cell contains the number of occurrences of the smell in a given project. We used this dataset mainly in the quantitative analysis of the data.

3.4 Analysis of Datasets with C# Bad Smells

We performed both quantitatively and qualitative data analysis. In quantitative data, research generally includes descriptive statistics and correlation analysis, for example. Descriptive statistics, such as mean values, standard deviation, histograms, and scatterplots, are used to help understand the data. Correlation analysis is used to describe how one smell is related to another. For qualitative data, the main objective of the analysis is to draw conclusions from the data, maintaining a clear chain of evidence. Besides, advances in Machine Learning techniques and data analysis have provided powerful means for extracting useful information and data knowledge (de Almeida Filho et al., 2019). In this study, we used some data analysis techniques, such as correlation analysis, association rule, and descriptive statistics.

In this stage of the work, we apply a set of data analysis techniques to the dataset formed with the bad smells. For this, we use scripts in python and the Jupyter notebook for the execution of scripts. The objective of this step was to understand the smells in C# and the relationship between them.

1. *Quantitative Analysis*: In this analysis, we used some methods of descriptive statistics, such as bar graphs, to understand the prevalence of smells in C#. More specifically, the most common smells and the most common types of smells.
2. *Correlation between Bad Smells*: An effective correlation coefficient measures the extent to which two variables tend to change together. Thus, it describes the strength and direction (positive or negative) of the relationship between these two variables. Correlations between variables can be measured using different coefficients (de Almeida Filho et al., 2019). The most common correlation coefficients are Pearson, Spearman, and Kendall.

   The Pearson correlation coefficient measures the strength of the linear relationship between continuous variables. It is the value that indicates how much a line can describe the relationship between variables. The Spearman correlation coefficient assesses the monotonic relationship between two continuous or ordinal variables. In a monotonic relationship, the variables tend to change together, but not necessarily at a constant rate. Finally, we have the correlation coefficient of Kendall, which is a measure of rank correlation, i.e., it verifies the similarity between the orders of the data when classified by one of the quantities. This correlation takes into account the directional agreement of the so-called concordant and discordant pairs. In this work, we use the Spearman correlation coefficient, because it is the correlation that does not need an assumption of normality of the distribution.
3. *Association Rules*: The purpose of the association rule is to identify associations between data records (items) that are related. For this, the main idea is to find subsets of items whose presence is correlated with the presence of some other item in the same transaction (de Almeida Filho et al., 2019). Smells usually do not exist in isolation and are often accompanied by others smells within the same class or in related classes (Walter et al., 2018). In the association rule, there are two components, one called antecedent and the other called consequent. Therefore, we observe the set of bad smells (antecedent) that implies the presence of other smells (consequent) in the same classes. We use the Apriori algorithm, which results in a set of various items that should be used to create the association rules that represent trends found in the dataset. Then we use the association rule to identify associations between the bad smells in C#.

3.5 Analysis of Possible Anti-patterns

For the analysis of possible anti-patterns in C#, in the Results and Discussion section, we analyzed and discussed three research questions. Through these analyses, it was possible to select some bad smells present in the twenty-five projects analyzed and indicate them as strong candidates for possible anti-patterns.

4 RESULTS AND DISCUSSION

In this section, the results collected through analysis and observations are shown through the following research questions (RQ):

- RQ1: How often do bad smells appear in C# projects?
- RQ2: Are there significant correlations between C# bad smells?
- RQ3: Are there smells that occur together in C# projects?

4.1 Quantitative Analysis

To answer the first question (RQ1), we performed quantitative analysis on the occurrence of bad smells in the twenty-five projects collected in C#. For this, we built a dataset per project containing all bad smells collected in Designite. Subsequently, we created a dataset joining the smells of all chosen projects individually.

Table 1: Architecture Smells.

Acronym	Architecture Smell
ACD	Cyclic Dependency
AUD	Unstable Dependency
AAI	Ambiguous Interface
AGC	God Component
AFC	Feature Concentration
ASF	Scattered Function
ADS	Dense Structure

Tables 1, 2, and 3 contain the bad smells detected by Desiginite, together with the acronym for each smell (Garcia et al., 2009), (de Andrade et al., 2014), (Suryanarayana et al., 2014). An acronym was assigned to each smell, which the smells of design start with the letter “D”, those for implementation with the letter “I” and those for architecture with the letter “A”. These acronyms were created in order to facilitate later references in the article.

First, we observed that smell DFE was not found in the studied projects. We also noted that some

Table 2: Implementation Smells.

Acronym	Implementation Smells
ICC	Complex Conditional
ICM	Complex Method
IDC	Duplicate Code
IEC	Empty Catch Block
ILI	Long Identifier
ILM	Long Method
ILP	Long Parameter List
ILS	Long Statement
IMN	Magic Number
IMD	Missing Default
IVC	Virtual Method Call from Constructor

Table 3: Design Smells.

Acronym	Smell Design
DBH	Broken Hierarchy
DBM	Broken Modularization
DCD	Cyclically-Dependent Modularization
DCH	Cyclic Hierarchy
DDH	Deep Hierarchy
DDE	Deficient Encapsulation
DDA	Duplicate Abstraction
DHM	Hub-like Modularization
DIA	Imperative Abstraction
DIM	Insufficient Modularization
DMH	Missing Hierarchy
DMA	Multifaceted Abstraction
DMU	Multipath Hierarchy
DRH	Rebellious Hierarchy
DUE	Unexploited Encapsulation
DUH	Unfactored Hierarchy
DUA	Unnecessary Abstraction
DUN	Unutilized Abstraction
DWH	Wide Hierarchy
DFE	Feature Envy

smells are rare to happen, such as ACD, AUD, AAI, ACG, AFC, ASF, and ADS. Besides, the two most frequent smells are the IMN and the ILS, both of which are smells of implementation and with a 50.1 % and 16.74 % occurrence percentage, respectively. In addiction, the Newtonsoft.Json project is the largest in the number of lines of code (1594611) and the number of classes (14195). IMN and ILS are the bad smells that occur most frequently in this project.

4.2 Correlation between the Bad Smells

We obtained the correlation between the bad smells of the C# code to assess how closely they are related. We performed an analysis of the strongly correlated characteristics to identify whether they are potentially related. Initially, we had to perform an analysis of the smells correlation in each project using the correlation coefficient of Spearman and a lower limit of 0.5. As shown in Figure 2, we noted that as the number of classes in a project increased, the values of the correlation coefficients tended to decrease. It occurs since the number of classes in a project increases. So, the smells tend to dissolve between them, thus decreasing the correlation between smells.

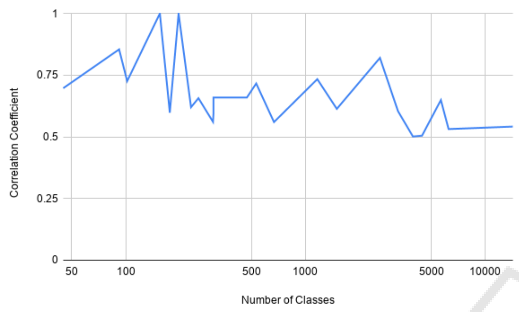

Figure 2: Graph with the Relationship between Correlation Coefficient and Number of Classes of the Project.

However, when analyzing the correlation between bad smells at the project level, we observed that there were many weakly correlated smells. So to answer the research question RQ2, we listed only smells with a correlation coefficient greater than 0.675.

The DUE and DIM smells have a correlation coefficient equal to 1 in the MaterialSkin project. The smell 1 addresses the fact that it has a series of if-else within a class by doing an explicit type check. The smell 2 addresses the fact that a very large class can be reduced to reduce complexity. So, it makes sense that the two smells occur together, as there is a large class where there are several if-else doing type checking. However, this class could be reduced using the concept of polymorphism. The code snippet in Figure 3 contains a real example of the joint occurrence of smells ILS and IMN in the MaterialSkin project.

The smells DUA and DBM are strongly correlated since the smell 1 occurs when a class exists, but it is not necessary. The smell 2 occurs when methods that should be present in a single class are found in several classes. Thus, the strong correlation of these 2 smells makes sense because the classes instantiated unnecessarily may be using methods that should be exclusive to a single class. The smells ICM and ICC are also strongly correlated. This is justified because smell 1 when it occurs generates very high cyclomatic complexity, and smell 2 occurs when a very complex decision structure is used. Thus, if there is a complex decision structure within a method, the method will also be complex, increasing its cyclomatic complexity.

The smells ACD and DMA are strongly correlated since smell 1 occurs when a class A method depends on a class B method and vice versa. The smell 2 occurs when there is a class with many methods. Thus, a class with a large number of methods has a good chance of being called and going into cycles.

4.3 Association Rule

In order to answer the RQ3 research question, we have developed a script in Python that implements the Apriori algorithm to find the association rules. We generated a data collection in table format as follows: if the smell occurs in a given project, the value TRUE for that cell, and the value FALSE will be associated, otherwise.

With datasets in the proper format, the next step in finding the association rules based on confidence is to set the confidence parameter to 50 %. With this lower limit, it is possible to identify the smells that occur together in some projects. We also identified which of the smells occur in most projects. In total, we found 56 different rules that happened in at least two projects. However, we have considered only the rules that happened from three projects onwards in this study.

Thus, we used three metrics: Support, Confidence, Lift. According to (Cardoso and Figueiredo, 2015), the Support metric calculates the proportion of transactions that contain the set of items, showing their importance and significance. Confidence is the probability of seeing the consequent of the rule on the condition that the transactions contain the antecedent. Moreover, the Lift metric measures how many times more than expected the antecedent smell, and the consequent smell occur together if they are statistically independent.

From the results of the association rule, we observed that:

- IMN is the smell that most occurs together with others smells.
- The values displayed in the columns that start with the word min., inform the lowest value of the parameter found in one of the projects in which the rule occurred. Furthermore, the opposite is repeated for the values shown in the columns that start with the word max.

The rule that occurs in the most significant number of projects, in which the antecedent is smell ILS, and the consequent is IMN, also appears in the correlation

```
private static readonly Color FLAT_BUTTON_BACKGROUND_HOVER_LIGHT = Color.FromArgb(20.PercentageToColorComponent(), 0x999999.ToColor());
```

Figure 3: Source code with the joint occurrence of the ILS and IMN smells.

analysis, as previously studied. The second rule that happens in the highest number of projects has as its antecedent the smell ICM and the consequent smell IMN. The source code snippet in Figure 4 contains an example from the SharpZipLib project of the occurrence of this rule.

```
public void SetData(byte[] data, int index, int count)
        {
            using (MemoryStream ms = new MemoryStream(data, index, count, false))
            using (ZipHelperStream helperStream = new ZipHelperStream(ms))
            {
                // bit 0       if set, modification time is present
                // bit 1       if set, access time is present
                // bit 2       if set, creation time is present
                _flags = (Flags)helperStream.ReadByte();
                if (((_flags & Flags.ModificationTime) != 0))
                {
                    int iTime = helperStream.ReadLEInt();
                    _modificationTime = new DateTime(1970, 1, 1, 0, 0, 0, 0, DateTimeKind.Utc) +
                        new TimeSpan(0, 0, 0, iTime, 0);
                    if (count <= 5) return;
                }
                if ((_flags & Flags.AccessTime) != 0)
                {
                    int iTime = helperStream.ReadLEInt();
                    _lastAccessTime = new DateTime(1970, 1, 1, 0, 0, 0, 0, DateTimeKind.Utc) +
                        new TimeSpan(0, 0, 0, iTime, 0);
                }
                if ((_flags & Flags.CreateTime) != 0)
                {
                    int iTime = helperStream.ReadLEInt();
                    _createTime = new DateTime(1970, 1, 1, 0, 0, 0, 0, DateTimeKind.Utc) +
                        new TimeSpan(0, 0, 0, iTime, 0);
                }
            }
        }
```

Figure 4: Source code with the joint occurrence of ICM and IMN smells.

4.4 Analysis of Possible Anti-patterns

Bad Smells and anti-patterns are recurring software problems. There is a minimal difference between the two terms, in which anti-pattern is considered a inadequate programming practice, but not an error. In (Singh and Kaur, 2017), the author defined bad smell as a warning for the presence of an anti-pattern.

Bad smells warn the developers of software that the source code has some problems, while anti-patterns provide software engineers, architects, designers, and developers a common vocabulary to recognize possible sources of problems in advance. However, refactoring is a solution to remove both anti-patterns and bad smells from the code. According to (Luo et al., 2010), the refactoring methods for bad smells are more technical and programming-based, whereas for anti-patterns are ”approaches to involving the solution in a better one.”

During the quantitative analysis of datasets with bad smells, we noted that the smells IMN and ILS are the ones that occur in higher numbers both in the joint analysis of the 25 projects and in the individual analysis per project. Moreover, in the qualitative analysis, we also noted that the IMN and ILS smells have had a high correlation and occurred together in 14 projects. The frequency of these two smells can be caused by some common reasons in development environments, such as the time when the functionality must be developed and delivered, and the lack of knowledge and understanding of good programming practices. Thus, these two smells are strong candidates to be classified as possible anti-patterns.

5 CONCLUSIONS

In this paper, we presented an exploratory study on the occurrence of bad smells in C# projects. We analyzed 25 open-source projects, published on GitHub, in order to study the prevalence of bad smells in C# code. From this exploratory study, we answered the three key questions of the article:

- RQ1: How often do bad smells appear in C# projects? Some smells are more frequent than others. Implementation smells are the most common, with IMN and ILS being the most common.
- RQ2: Are there significant correlations between bad smells C#? The results showed that some smells have high correlation coefficients, with some having a coefficient equal to 1, such as the smells IMD and DUN.
- RQ3: Are there smells that occur together in C# projects? We used the Apriori algorithm and found some interesting association rules, in which we verified the occurrence of some smells that occurred together. Also, we identified that the rule that occurred in most projects was the one that had the smell ILS as an antecedent and IMN as a consequent.

Finally, we analyzed that the smells IMN and ILS can be considered possible anti-patterns, this being a future work. So, this work has the potential to help professionals in the software development field in order to avoid future problems during the development of C# projects.

ACKNOWLEDGEMENTS

This research was funded by LSBD/UFC.

REFERENCES

Alenezi, M. and Zarour, M. (2018). An empirical study of bad smells during software evolution using designite tool. *i-Manager's Journal on Software Engineering*, 12(4):12.

Cardoso, B. and Figueiredo, E. (2015). Co-occurrence of design patterns and bad smells in software systems: An exploratory study. In *Anais do XI Simpósio Brasileiro de Sistemas de Informação*, pages 347–354. SBC.

de Almeida Filho, F. G., Martins, A. D. F., Vinuto, T. d. S., Monteiro, J. M., de Sousa, Í. P., de Castro Machado, J., and Rocha, L. S. (2019). Prevalence of bad smells in pl/sql projects. In *Proceedings of the 27th International Conference on Program Comprehension*, pages 116–121. IEEE Press.

de Andrade, H. S., Almeida, E., and Crnkovic, I. (2014). Architectural bad smells in software product lines: An exploratory study. In *Proceedings of the WICSA 2014 Companion Volume*, page 12. ACM.

Fontana, F. A., Braione, P., and Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5–1.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Ganesh, S., Sharma, T., and Suryanarayana, G. (2013). Towards a principle-based classification of structural design smells. *Journal of Object Technology*, 12(2):1–1.

Garcia, J., Popescu, D., Edwards, G., and Medvidović, N. (2009). Toward a Catalogue of Architectural Bad Smells. In *Proceedings of the Fifth International Conference on Quality of Software Architectures: Architectures for Adaptive Software Systems*, volume 5581 of *Lecture Notes in Computer Science*, pages 146–162. Springer International Publishing.

Habchi, S., Hecht, G., Rouvoy, R., and Moha, N. (2017). Code smells in ios apps: How do they compare to android? In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 110–121. IEEE.

Kaur, A., Kaur, K., and Jain, S. (2016). Predicting software change-proneness with code smells and class imbalance learning. In *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 746–754. IEEE.

Li, W. and Shatnawi, R. (2007). An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software*, 80(7):1120–1128.

Liu, H., Yu, Y., Li, B., Yang, Y., and Jia, R. (2018). Are smell-based metrics actually useful in effort-aware structural change-proneness prediction? an empirical study. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 315–324. IEEE.

Luo, Y., Hoss, A., and Carver, D. L. (2010). An ontological identification of relationships between anti-patterns and code smells. In *2010 IEEE Aerospace Conference*, pages 1–10. IEEE.

Mannan, U. A., Ahmed, I., Almurshed, R. A. M., Dig, D., and Jensen, C. (2016). Understanding code smells in android applications. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 225–236. IEEE.

Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A., and Oliveto, R. (2016). Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 244–255. IEEE.

Pressman, R. and Maxim, B. (2016). *Engenharia de Software-8^a Edição*. McGraw Hill Brasil.

Sharma, T. (2017). Designite: A customizable tool for smell mining in c# repositories. In *10th Seminar on Advanced Techniques and Tools for Software Evolution, Madrid, Spain*.

Sharma, T., Fragkoulis, M., and Spinellis, D. (2017). House of cards: code smells in open-source c# repositories. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 424–429. IEEE Press.

Singh, S. and Kahlon, K. (2011). Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells. *ACM SIGSOFT Software Engineering Notes*, 36(5):1–10.

Singh, S. and Kaur, S. (2017). A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*.

Suryanarayana, G., Samarthyam, G., and Sharma, T. (2014). *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann.

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