The Relationship between Traceable Code Patterns and Code Smells

Zadia Codabux*[1], Kazi Zakia Sultana†[2] and Byron J. Williams‡[2]

[1]Department of Computer Science, Colby College, USA
[2]Department of Computer Science and Engineering, Mississippi State University, USA

Abstract

Context: *It is important to maintain software quality as a software system evolves. Managing code smells in source code contributes towards quality software. While metrics have been used to pinpoint code smells in source code, we present an empirical study on the correlation of code smells with class-level (micro pattern) and method-level (nano-pattern) traceable patterns of code.* ***Objective****: This study explores the relationship between code smells and class-level and method-level structural code constructs.* ***Method****: We extracted micro patterns at the class level and nano-patterns at the method level from three versions of Apache Tomcat and PersonalBlog and Roller from Standford SecuriBench and compared their distributions in code smell versus non-code smell classes and methods.****Result****: We found that $DataManager$, $Record$ and $Outline$ micro patterns are more frequent in classes having code smell compared to non-code smell classes in the applications we analyzed. $localReader$, $localWriter$, $Switcher$, and $ArrReader$ nano-patterns are more frequent in code smell methods compared to the non-code smell methods.* ***Conclusion****: We conclude that code smells are correlated with both micro and nano-patterns.*

1. Introduction

Software quality is crucial. Detecting and managing code smells improves the quality and maintainability of software systems [10, 12]. A code smell is a surface indication that usually corresponds to a deeper problem in the system [7]. This surface indication may be related to a code or design problem that increases the difficulty of maintenance. [6]. Various raw measures have been derived to assess system characteristics such as lines of code, method length, number of methods etc [11]. These types of measures have been evaluated to establish relative threshold values that are used with other metrics to define code smells. Code smell detection tools are based on the evaluated metrics and established thresholds. A small change in threshold values impacts code smell detection accuracy [5]. Code smells can also be transitively related to each other [6]. Fontana et al. [4] also identified other relations among code smells useful for detecting architectural degradation.

Research has been conducted on code smell detection mechanisms, correlations among code smells, and code smell fault proneness [9, 10, 12, 17, 18]. Fontana et al. analyzed the dependencies between code smells and micro patterns [5]. Micro patterns are class-level traceable patterns that are defined using some formal conditions of the structure of a Java class [8]. Gil et al. [8] developed the concept of traceable patterns and defined 27 micro patterns organized into 8 categories based on the structure of Java classes. These studies were focused on improving software quality by reducing the use of bug-prone micro patterns. Fontana found that dependencies between code smells and micro patterns can be used to improve code quality by detecting code smells using micro pattern data [5]. Method-level traceable patterns are called nano-patterns. Singer et al. [16] listed 17 fundamental nano-patterns organized into 4 groups. They applied their work to clustering and categorizing Java methods based on the associated nano-patterns. Deo et al. [2] found that some nano-patterns such as $localReader$ are highly present in defective methods.

Previous studies focused on finding correlations between code smells and micro patterns. There are certain code smells that are defined based on the methods in a class (e.g., feature envy, long parameter list). In this study, our goal is to determine if correlations exist between code smells with

*zadiacodabux@ieee.org
†ks2190@msstate.edu
‡williams@cse.msstate.edu

DOI reference number: 10.18293/SEKE2017-121

both micro and nano-patterns. These patterns are defined on class or method behavior and code smell occurs due to the violation of fundamental design principles. Therefore, correlating class and method-level structural and behavioral information with code smell will strengthen code smell detection and augment existing smells with pattern-based information. The motivation of this study is to pinpoint code smells in Java classes and methods using their structural information. The patterns having high correlation with code smells can later be used for code smell detection. Moreover, developers will be guided about the use of patterns in code for ensuring better code quality. Detecting code smells in code provides a good indicator of the system's design quality.

The contributions of this study are as follows:

- The results serve to create awareness among developers that using certain patterns can result in particular code smells.

- This study present alternative approaches to identifying code smells and other problematic areas in software.

The paper is organized as follows: Section 2 discusses the related work. Section 3 presents the methodology of our work. Sections 4 presents the results from the experimental analysis and Section 5 discusses the results. Section 6 concludes the paper.

2. Related Work

Prior research focused on finding code smells using various threshold values for related metrics. The problems associated with interpreting these thresholds and setting values applicable to the system under evaluation greatly impact code smell detection accuracy [1, 14]. In order to avoid these problems, researchers examined ways to detect code smells more accurately by correlating smells with other constructs (including other code smells). Fontana [4] extracted relationships among code smells such as determining whether a code smell contains another code smell or if a smelly method calls another smelly method. The authors exploited these relations to tease out architectural anomalies in a system. Other studies described different relationships among code smells [6, 15]. Plain Support and Transitive Support are two other types of code smell relationships [6]. Plain Support denotes the likelihood of the existence of one code smell with the presence of another code smell. On the other hand, Transitive Support is concerned with transitive dependencies among three code smells [6].

There has also been research conducted on the relationships between micro patterns and defects. Destefanis et al. identified certain micro patterns that were more error-prone than others and observed correlations between the patterns and defects [3]. They also showed that the classes that do not contain any micro pattern are more fault prone than classes with micro patterns. In a study by [5], the authors correlated code smells with micro patterns. They explored how structural information can be exploited to detect code smells and bad programming practices. They identified associations among code smells as well as between code smell and micro patterns. Their study is helpful to expedite the code smell detection mechanism as well as to remove discrepancies or contradictions among different code smell detection tools [5].

Although prior studies concentrated on the associations among code smells and micro patterns, in this study we also identify how code smells are correlated to method-level patterns known as nano-patterns. This study results in a new approach for code smell detection using nano-patterns. We also examine correlations between class-level patterns and code smells and compare our results with the previous work.

3. Methodology

This Section elaborates the goal of the study, the research questions, the research approach, and study design.

3.1. Research Goal

The goal of the study is to determine if there exists a correlation between code smells with micro patterns and nano-patterns. This goal is addressed by the research questions presented in the Section 3.2.

3.2. Research Questions

Research Question 1 (RQ1): *How are code smells related to micro patterns?*

The rationale behind this question is to understand if micro patterns are distributed differently in classes having code smells versus classes without code smells. Moreover, we aim to find if there is a correlation between micro patterns with different types of code smells. More specifically:

RQ1.1: *How are micro patterns distributed in classes containing code smells?*

RQ1.2: *Are micro patterns distributed differently in classes with different types of code smells?*

Research Question 2 (RQ2): *How are code smells related to nano-patterns?*

This question is similar to RQ1 except that this analysis is at the method-level as nano-patterns are method-level traceable patterns. More specifically:

RQ2.1: *How are nano-patterns distributed in methods having code smells?*

RQ2.2: *Are the nano-patterns distributed differently in methods with different types of code smells?*

3.3. Study Design

The study was conducted using 3 versions of Apache Tomcat along with PersonalBlog and Roller from Stanford SecuriBench[1]. Apache Tomcat is a web application server developed by the Apache Software Foundation[2]. The software has about a half million lines of code with more than 3000 classes and 12,000 methods in each version. For our study, we considered versions 6.0.45, 7.0.69 and 8.0.33 from the Apache Tomcat Archives[3] (the last released version for each major release during this study). Stanford SecuriBench is a set of Java open source real-life programs. PersonalBlog is a personal blogging application. The software has 38 classes and 275 methods. Roller is a blog server consisting of 226 classes and 2136 methods. In this study, we use the term "code smell classes" to describe the classes where code smells are found and "non-code smell classes" for the classes where no code smell is found.

3.4. Data Extraction

Step 1: Extract code smells: Next, we used Intooitus inCode [4] to extract the code smells from the source code. inCode follows Marinescu's detection strategies to quantify design problems [11] [13]. inCode tests for code smells that are most commonly encountered in software projects. inCode extracts the following specific class-level code smells: god class, data class and schizophrenic class and method-level code smells: Data Clump, Duplication, Feature Envy. We ran inCode on three versions of Apache Tomcat and the two SecuriBench software.

Step 2: Extract micro patterns: The micro pattern tool as described in [8] is interfaced via the command line and accepts a class name or .jar file as input and extracts all micro patterns identified in that class or classes in the jar file. If a particular micro pattern exists in that class, the respective entry is '1' otherwise, it is '0'. We evaluated the unique classes across three versions of Tomcat. For Tomcat, we collected the micro pattern data for a total of 117 "code smell classes" and 7848 "non-code smell classes". For PersonalBlog, we collected data for 9 "code smell classes" and 28 "non-code smell classes". For Roller, we collected data for 33 "code smell classes" and 238 "non-code smell classes".

Step 3: Extract nano-patterns: The nano-pattern detector detects nano-patterns in Java bytecode [16]. We modified the original version of the tool to provide input via a '.properties' file instead of command line arguments and stored the results in a database [2]. For Tomcat, we collected the nano-pattern data for a total 473 "code smell methods" and 37226 "non-code smell methods". For Roller, we collected the nano-pattern data for a total 36 "code smell methods" and 76218 "non-code smell methods".

[1]https://suif.stanford.edu/ livshits/securibench/intro.html

[2]http://tomcat.apache.org/

[3]http://archive.apache.org/dist/tomcat/

[4]https://www.intooitus.com,its evolution at http://www.aireviewer.com

3.5. Data Analysis

Research Question 1 (RQ1): *How are code smells related to micro patterns?*

For RQ1.1, we calculated the percentage of each micro pattern in code smell classes and non-code smell classes. For example, in Tomcat there are 10 classes with the $Record$ micro pattern of the 117 total code smell classes. This micro pattern exists in 8.55% of code smell classes. For RQ1.2, we separated the classes affected by each type of code smell: data class, schizophrenic class and god class. We then computed the percentage of each micro pattern in those classes. For example, in Tomcat we have 18 classes with the $Sink$ micro pattern out of total 35 classes reported to contain the data class type of code smell. Therefore, we can say the percentage of $Sink$ micro pattern in data class code smell classes is 51.4%.

Research Question 2 (RQ2): *How are code smells related to nano-patterns?*

For RQ2.1, we calculated the percentage of each nano-pattern extracted from code smell methods as well as non-code smell methods. For example, in Tomcat there were 192 methods having $ObjCreator$ nano-pattern out of 473 methods having code smells. So 40.6% of the code smell methods have the $ObjCreator$ nano-pattern. Regarding RQ2.2, we separated the methods affected by the following types of code smell: data clumps, duplication and feature envy. The message chain code smell was not a very prominent smell in the system methods and we did not perform any further analysis on it. We then computed the percentage of each nano-pattern in those methods. For example, in Tomcat we have 242 methods having $Void$ nano-patterns out of 300 methods reported to have data clump code smell. Therefore, we can deduce that the percentage of $Void$ nano-pattern in data clump code smell is 80.7%.

For RQ1.1 and RQ2.1, we also performed a Chi-Square Test of Independence to compare the difference in distribution of micro patterns and nano-patterns across the code smell classes and methods and non-code smell classes and methods respectively. We formulated the following null hypotheses with the significance level α = 0.05.

$H0$ *The distribution of micro patterns is independent of code smell and non-code smell classes.*

$H0$ *The distribution of nano-patterns is independent of code smell and non-code smell methods.*

We performed separate Chi Square Tests for Independence for the different types of micro patterns and nano-patterns.

4. Results

4.1. Research Question 1 (RQ1)

The test statistic for the chi-Square test of independence involves comparing observed (sample data) and expected frequencies. If the null hypothesis is confirmed, the observed and expected frequencies will be close in value and the chi-Square statistic will be close to zero. If the null hypothesis is false, then the chi-Square statistic will be large. For degrees of freedom of 1 and at 5% levels of significance, the appropriate critical value is 3.84 and the decision rule is as follows: Reject $H0$ if $\tilde{\chi}^2 \geq 3.84$. We computed the chi-square test statistic for each micro pattern and nano-pattern. We reject the null hypothesis when the test statistics for the micro patterns are greater than 3.84. We have statistically significant evidence at $\alpha = 0.05$ to show the distribution of these following micro patterns is not independent of the code smell and non-code smell classes. We have summarized our finding as follows:

- For Apache Tomcat, the micro patterns that are frequent in code smell classes compared to the non-code smell classes are $DataManager$, $Record$ and $Outline$.
- For PersonalBlog, the micro patterns that are frequent in code smell classes compared to the non-code smell classes are $Immutable$ and $Sink$.

Our results on the relations between the different micro patterns and the types of code smells as addressed by RQ1.2 are summarized as follows:
For Apache Tomcat,

- The micro patterns that are more frequent in the classes having data class code smell are $CompoundBox$, $RestrictedCreation$, $CommonState$, $Sink$, $Record$, $FunctionObject$, $Immutable$, $Extender$, $DataManager$, $LimitedSelf$.
- The micro patterns more frequent in the classes having schizophrenic class code smell are $CompoundBox$, $Outline$, $PureType$, $StateMachine$.
- The micro patterns more frequent in the classes having god class code smell are $Implementor$, $Overrider$, $Stateless$, $Recursive$, $LimitedSelf$.

For PersonalBlog,

- The micro patterns that are more frequent in the classes having data class code smell are $Sink$ and $Immutable$.

The Roller application did not contain a significant number of nano-patterns and was excluded from these results.

4.2. Research Question 2 (RQ2)

To address RQ2.1, we separated all the methods containing code smells and then found the nano-pattern distribution in those methods. We calculated the chi-square test statistics for nano-patterns. We reject the null hypothesis when the test statistics for the nano-patterns were greater than 3.84. We have statistically significant evidence at $\alpha = 0.05$ to show the distribution of the nano-patterns is not independent of the code smell and non-code smell methods. We have summarized our findings as follows:

- For Apache Tomcat, the nano-patterns that are more frequent in code smell methods compared to the non-code smell methods are $localWriter$, $Switcher$ and $ArrReader$.
- For Roller, the nano-patterns that are more frequent in code smell methods compared to the non-code smell methods are $localReader$ and $localWriter$.

Our results on the relation between different nano-patterns and the three code smells as addressed by RQ2.2 are summarized as follows:
For Apache Tomcat,

- The nano-patterns more frequent in the methods having data clump code smell are $Void$, $LocalReader$, $JdkClient$ and $TailCaller$.
- The nano-patterns more frequent in the methods having duplication code smell are $LocalReader$, $LocalWriter$, $JdkClient$ and $TailCaller$.
- The nano-patterns more frequent in the methods having feature envy code smell are $objCreator$, $LocalReader$, $LocalWriter$ and $thisInstanceFieldReader$.

For Roller,

- The nano-patterns more frequent in the methods having data clump code smell are $LocalReader$ and $exceptions$.
- The nano-patterns more frequent in the methods having feature envy code smell are $objCreator$ and$LocalWriter$.

The PersonalBlog application did not contain a significant number of nano-patterns and was excluded from these results.

5. Discussion

5.1. Code Smell and Micro Patterns

From Section 4.1, the micro patterns that are frequent in code smell classes compared to the non-code smell classes are $DataManager$, $Record$ and $Outline$. Fontana [5] found that the $Outline$ micro pattern often results in $SignificantDuplication$ code smell (Confidence level=76%). We also found association of the set of $DataManager$, $Immutable$ and $Extender$ with different code smells at more than 60% confidence.

Data classes are classes which only contain fields, getters / setters, or only public fields. $DataManager$ is a class where all methods are either setters or getters [8]. $Record$ is a class in which all fields are public, no declared methods. $Sink$ is a class whose methods do not propagate calls to any other class. An $Immutable$ class is class whose instance fields are only changed by its constructors. According to 4.1, $DataManager$, $Record$, $Sink$, and $Immutable$ are among the micro patterns that are more frequent in the classes having data class code smell. The definition of these micro patterns support our finding relating to their association with the data class code smell. For example, $Record$ classes contain public fields as data classes do in some cases. As data classes contain only getter or setter methods for finding or setting values to their fields, they do not call other methods to serve any other purpose. Similarly, $Sink$ classes also do not allow its methods to call methods from other classes. Schizophrenic class describes a class with a large and non-cohesive interface. The lack of cohesion is revealed by several disjoint sets of public methods that are used by disjoint sets of client classes. The class with $PureType$ pattern has nothing more than four abstract methods which concrete subclasses must override [8]. $StateMachine$ pattern is an interface to define only parameter-less methods. Such an interface allows client code to either query the state of the object or request the object to change its state in some predefined manner [8]. Therefore, all these types of micro patterns can result in non-cohesive interfaces. $Implementor$ and $Overrider$ micro patterns are among the most frequent micro patterns in the classes containing the god class code smell. God class is an excessively complex class with non-cohesive functionality and heavily manipulates data members from other classes. $Implementor$ is a concrete class, where all the methods override inherited abstract methods. $Overrider$ is a class where all methods override inherited, non-abstract methods [8]. The use of these patterns increases the possibility of having non-cohesive environment.

5.2. Code Smells and nano-patterns

From Section 4.2, the nano-patterns that are more frequent in code smell methods compared to the non-code smell methods are $localWriter$, $Switcher$ and $ArrReader$. $localWriter$ writes values of local variables on the stack frame. $ArrReader$ reads values from an array and $Switcher$ patterns are methods that contain switch statements [16]. We found $ArrReader$ to be a problematic nano-pattern, and this result is in line with the findings of Deo [2] which reported that methods with the $ArrReader$ nano-pattern are highly defect prone.

Data Clumps are large groups of parameters that appear together in the signature of many operations. The nano-patterns that are more frequent in the methods containing the data clump code smell are $Void$, $LocalReader$, $JdkClient$ and $TailCaller$. $Void$ patterns are methods that do not return a value. $LocalReader$ reads values of local variables on stack frame. $JdkClient$ calls methods from the JDK standard library (java.*). $TailCaller$ contains method call followed immediately by return statement [16]. Based on these observations, we can deduce that methods with data clump code smells may not return any value or return control to the calling function after execution. These methods interact and store local variables on the stack. Code Duplication refers to groups of operations which contain identical or slightly adapted code fragments. By breaking the Don't Repeat Yourself (DRY) rule, duplicated code multiplies the maintenance effort, including the management of changes and bug-fixes. Moreover, the code base gets bloated. $LocalReader$, and $LocalWriter$ nano-patterns are among the more frequent patterns in the methods having the duplicated code smell. $LocalWriter$ writes values of local variables on stack frame [16]. $LocalWriter$ has been associated with high defect density [2] and an undesirable effect of code duplication is the introduction of defects. $JdkClient$ may call the same method multiple times and this contributes to code duplication. Feature Envy refers to an operation that is manipulating a lot of data external to its definition scope. In object-oriented code this is a method that uses many data members from a few other classes instead of using the data members of its definition class. $objCreator$ nano-pattern has been found as more frequent in the methods having feature envy code smell. $objCreator$ creates new objects in a method and therefore, its association with a method makes the method more associated with other classes. $objCreator$ has been associated with high defect density as well [2].

6. Conclusion

This study represents preliminary data collection and analysis to determine the relationship of code smells with micro and nano-patterns. We analyzed how the micro patterns and nano-patterns are distributed in code smell and non-code smell classes. Our results can be used for identifying micro patterns and nano-patterns causing code smells. It can also help developers in avoiding the use of particular micro and nano-patterns that often result in code smells. We provided an indication of the most pertinent patterns to code smells by doing a comparative analysis between code smell and non-code smell classes and methods. This study provides a basis for future work to determine the underlying reason behind the significantly different distribution of patterns in code smell vs non-code smell code. We also plan to extend the study to other systems as well as use these findings to create a prediction model for code smells using traceable patterns.

References

[1] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10. IEEE Computer Society, 2010.

[2] A. Deo and B. J. Williams. Preliminary study on assessing software defects using nano-pattern detection. In *Proceedings of the 24th International Conference on Software Engineering and Data Engineering (SEDE)*, 2015.

[3] G. Destefanis, R. Tonelli, E. Tempero, G. Concas, and M. Marchesi. Micro pattern fault-proneness. In *Proceedings of the 2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA '12, pages 302–306. IEEE Computer Society, 2012.

[4] F. A. Fontana, V. Ferme, and M. Zanoni. Towards assessing software architecture quality by exploiting code smell relations. In *Proceedings of the Second International Workshop on Software Architecture and Metrics*, SAM '15, pages 1–7. IEEE Press, 2015.

[5] F. A. Fontana, B. Walter, and M. Zanoni. Code smells and micro patterns correlations. In *RefTest 2013 Workshop, co-located event with XP 2013 Conference*, 2013.

[6] F. A. Fontana and M. Zanoni. On investigating code smells correlations. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 474–475. IEEE, 2011.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, July 1999.

[8] J. Y. Gil and I. Maman. Micro patterns in java code. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 97–116. ACM, 2005.

[9] T. Hall, M. Zhang, D. Bowes, and Y. Sun. Some code smells have a significant but small effect on faults. *ACM Trans. Softw. Eng. Methodol.*, 23(4):33:1–33:39, Sept. 2014.

[10] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 75–84. IEEE Computer Society, 2009.

[11] M. Lanza and R. Marinescu. *Object-oriented metrics in practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Publishing Company, Inc., 2010.

[12] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *J. Syst. Softw.*, 80(7):1120–1128, July 2007.

[13] R. Marinescu, G. Ganea, and I. Verebi. Incode: Continuous quality assessment and improvement. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 274–275, March 2010.

[14] P. Oliveira, M. T. Valente, and F. P. Lima. Extracting relative thresholds for source code metrics. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 254–263, Feb 2014.

[15] B. Pietrzak and B. Walter. Leveraging code smell detection with inter-smell relations. In *Proceedings of the 7th International Conference on Extreme Programming and Agile Processes in Software Engineering*, XP'06, pages 75–84. Springer-Verlag, 2006.

[16] J. Singer, G. Brown, M. Lujn, A. Pocock, and P. Yiapanis. Fundamental nano-patterns to characterize and classify java methods. *Electronic Notes in Theoretical Computer Science*, 253(7):191 – 204, 2010. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[17] A. Yamashita. Assessing the capability of code smells to explain maintenance problems: An empirical study combining quantitative and qualitative data. *Empirical Softw. Engg.*, 19(4):1111–1143, Aug. 2014.

[18] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, pages 17–23. ACM, 2011.