International Journal of Software Engineering and Knowledge Engineering
© World Scientific Publishing Company

The Relationship between Code Smells and Traceable Patterns - Are They Measuring the Same Thing?

ZADIA CODABUX
Department of Computer Science
Colby College, USA.
zadiacodabux@ieee.org

KAZI ZAKIA SULTANA
Department of Computer Science and Engineering
Mississippi State University, USA.
ks2190@msstate.edu

BYRON J. WILLIAMS
Department of Computer Science and Engineering
Mississippi State University, USA.
williams@cse.msstate.edu

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Context: It is important to maintain software quality as a software system evolves. Managing code smells in source code contributes to the quality of a 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 code patterns. **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, three versions of Apache CXF, two J2EE web applications: PersonalBlog and Roller from Stanford SecuriBench and then compared their distributions in code smell versus non-code smell classes and methods. **Results**: We found that *Immutable* and *Sink* micro patterns are more frequent in classes having code smells compared to the non-code smell classes in the applications we analyzed. On the other hand, *LocalReader* and *LocalWriter* 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.

Keywords: Code Smell; Micro Pattern; Nano-Pattern; Traceable Pattern

1. Introduction

Software quality is crucial. Detecting and managing code smells improves the quality and maintainability of software systems [17], [19]. A code smell is a surface indication that usually corresponds to a deeper problem in the system [13]. This surface indication may be related to a code or design problem that increases the difficulty of maintenance [12]. Various raw measures including lines of code, method length and number of methods have been derived to assess system characteristics [18]. 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 [11]. Code smells can be transitively related to each other [12]. Fontana et al. [9] identified 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 [16], [17], [19], [27], [28]. Fontana et al. analyzed the dependencies between code smells and micro patterns [11]. Micro patterns are class-level traceable patterns that are defined using the formal conditions of the structure of a Java class [15]. Gil et al. [15] developed the concept of traceable patterns and defined 27 micro patterns organized into eight categories based on the structure of Java classes. Researchers focused on improving software quality by reducing the use of bug-prone micro patterns [6]. 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 [11]. Method-level traceable patterns are called nano-patterns. Batarseh first introduced the idea of nano-patterns in [3]. Singer et al. [24] listed 17 fundamental nano-patterns organized into 4 groups. They extended the catalog of nano-patterns and developed a tool to extract nano-patterns from Java source code. They applied their work to clustering and categorizing Java methods based on the associated nano-patterns. Deo et al. [5] found that some nano-patterns, such as *LocalReader*, are highly present in defective methods. Sultana et al. found potentially vulnerable areas in code using nano-patterns [29], [30]. Micro and nano-patterns are related to particular granularity level of code. Micro patterns are defined based on object-oriented features of a class, and they can better capture the code defects that result from object-oriented characteristics such as encapsulation and inheritance. On the other hand, nano-patterns are more granular than micro patterns. They are method-based patterns and capture the properties of a function inside a class. Therefore, relating these granular patterns with code smells can serve in a better indication of code smells and improved code quality.

Previous research has focused on finding correlations between code smells and micro patterns [11]. In this paper, the authors investigated the association rules among code smells and micro patterns to find their co-existence in code, but they did not specify the minimum support and confidence they considered for pruning the rules. Moreover, they found the existence of code smells in a particular micro pattern class and ignored the existence of code smells in the classes without the micro pattern. As a result, the work lacks in the comparative analysis of the presence of code smells in classes having a particular micro pattern and classes without the pattern. In addition, there are certain code smells that are defined based on the methods in a class (e.g. feature envy, long parameter list, and so on).

In this study, our goal is to determine whether there exists any correlation between code smells and micro and nano-patterns by analyzing both code smell and non-code smell classes and methods respectively. 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 smells 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 source code provides a good indicator of the system's design quality.

This study is an extension of our earlier research performed on two software systems to analyze the relationship between code smells and micro and nano-patterns [4].

The contributions of this study are as follows:

- The correlation between code smells and traceable patterns serve to create awareness among developers that using certain patterns can result in particular code smells.
- This study presents alternative approaches to identifying code smells and other problematic areas in software.

The paper is organized as follows: Section 2 describes the terminologies used in this study. Section 3 discusses the related work. Section 4 presents the methodology of this study. Section 5 presents the results from the experimental analysis, and section 6 discusses the results. Section 7 presents the limitations of this work. Finally, section 8 concludes the paper.

2. Background

In this section, we introduce and describe some of the terminologies that we will use throughout this study.

2.1. *Micro Pattern*

Micro patterns are mechanically traceable patterns based on the formal conditions of the structure of a Java class. Different class characteristics, such as use of inheritance, immutability, data management and wrapping, restricted creation etc., are considered for defining micro patterns. They are similar to design patterns except

Table 1. Micro patterns in the catalog

Category	Patterns	Description
Degenerate State and Behavior	Designator	An interface with absolutely no members.
	Taxonomy	An empty interface extending another interface.
	Joiner	An empty interface joining two or more superinterfaces.
	Pool	A class which declares only static final fields, but no methods.
Degenerate Behavior	Function Pointer	A class with a single public instance method, but with no fields.
	Function Object	A class with a single public instance method, and at least one instance field.
	Cobol Like	A class with a single static method, but no instance members
Degenerate State	Stateless	A class with no fields, other than static final ones.
	Common State	A class in which all fields are static.
	Immutable	A class with several instance fields, which are assigned exactly once, during instance construction.
Controlled Creation	Restricted Creation	A class with no public constructors, and at least one static field of the same type as the class
	Sampler	A class with one or more public constructors, and at least one static field of the same type as the class
Wrappers	Box	A class which has exactly one, mutable, instance field.
	Compound Box	A class with exactly one non primitive instance field.
	Canopy	A class with exactly one instance field that it assigned exactly once, during instance construction.
Data Managers	Record	A class in which all fields are public, no declared methods.
	Data Manager	A class where all methods are either setters or getters.
	Sink	A class whose methods do not propagate calls to any other class.
Base Classes	Outline	A class where at least two methods invoke an abstract method on"this"
	Trait	An abstract class which has no state.
	State Machine	An interface whose methods accept no parameters.
	Pure Type	A class with only abstract methods, and no static members, and no fields
	Augmented Type	Only abstract methods and three or more static final fields of the same type
	Pseudo Class	A class which can be rewritten as an interface: no concrete methods, only static fields
Inheritors	Implementor	A concrete class, where all the methods override inherited abstract methods.
	Overrider	A class in which all methods override inherited, non-abstract methods.
	Extender	A class which extends the inherited protocol, without overriding any methods.

that they are defined at a lower level of abstraction and have a particular granularity level [15]. Design patterns are not directly traceable from code. On the other hand, micro patterns are defined on class-level, and each single class may contain a set of micro patterns. The combination of these micro patterns represent that class. These micro patterns are traceable and can be extracted from source code. Table 1 presents the detailed description of the 27 micro patterns defined in [15].

2.2. *Nano-pattern*

Nano-patterns are method-level traceable patterns that capture the formal properties of Java methods. For example, a method having no branching or looping maintains a simple coding structure. This type of method is termed as a *StraightLine* pattern whereas a method containing loops is known as a *Looping*. Table 2 presents the 17 fundamental nano-patterns [24]. Singer et al. defined some additional nano-patterns in the catalog of nano-patterns. They developed a tool to extract nano-patterns including the fundamental nano-patterns from source code. As we used the same tool for our study, our results show all of the nano-patterns as defined in [24].

2.3. *Code Smell*

Code smells are design flaws in a system. Code smells usually lead to software systems that are difficult to maintain and potentially more fault-prone. There are different flavors of code smells, both at class (referred to as code smell classes henceforth) and method (referred to as code smell methods henceforth) levels. Classes and methods with no occurrence of code smells are referred to as non-code smell classes and non-code smell methods respectively in this study. Table 3 describes the code smells used in this study.

Table 2. Catalog of Fundamental Nano-patterns

Category	Nano-patterns
Calling	*NoParams*—takes no arguments *NoReturn* — returns void *Recursive* — calls itself recursively *SameName* — calls another method with the same name *Leaf* — does not issue any method calls
Object-Oriented	*ObjectCreator* — creates new objects *FieldReader* — reads (static or instance) field values from an object *FieldWriter* — writes values to (static or instance) field of an object *TypeManipulator* — uses type casts or instanceof operations
Control Flow	*StraightLine* — no branches in method body *Looping* — one or more control flow loops in method body *Exceptions* — may throw an unhandled exception
Data Flow	*LocalReader* — reads values of local variables on stack frame *LocalWriter* — writes values of local variables on stack frame *ArrayCreator* — creates a new array *ArrayReader* — reads values from an array *ArrayWriter* — writes values to an array

Table 3. Code Smells Description

Category	**Code Smell**	**Description**
Class-Level	God Class	An excessively complex class which gathers too much non-cohesive functionality and heavily manipulates data members from other classes.
	Data Class	Classes which only contain fields, getters / setters, or only public fields.
	Schizophrenic Class	Classes with a large and non-cohesive interface.
Method-Level	Data Clump	Refers to large groups of parameters that appear together in the signature of many operations.
	Duplication	Refers to groups of operations which contain identical or slightly adapted code fragments.
	Feature Envy	Refers to a method that accesses the data of another object more than its own data.

3. 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], [21]. 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 [9] 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 [12], [22]. Plain support and transitive support are two other types of code smell relationships [12]. 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 [12].

Researchers also conducted studies 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 [6]. They also showed that the classes that do not contain any micro pattern are more fault-prone than classes with micro patterns. In another study, Singer et al. extracted the association between micro patterns with class name suffixes which might be useful for run-time bug detection by the developers [25]. Sultana et al. analyzed nano-patterns to explore the relationship between nano-patterns and vulnerabilities [29], [30]. Deo et al. also found certain nano-patterns to be more fault-prone than others [5]. Fontana et al. 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 [11].

Research relating code smells, vulnerabilities and traceable patterns are also prominent. According to Islam et al. [36], "A particular piece of code is considered vulnerable if it contains code smells or bad coding patterns, and the severity of the vulnerability is dictated by the severity of existing code smells." Software vulnerabilities are types of software defects for which researchers developed a number of metrics and patterns to ensure software security and quality [37], [38], [39]. All the metrics have high false negative rates, and they have no particular granularity level. Researchers also introduced security patterns in different phases of software development [31]. Yoder et al. presented the first seven security patterns in [34]. In [35], the authors showed that security patterns are not as good as design patterns, and only one-fourth of the software security patterns provide code examples. Researchers developed some security patterns and guidelines for the developers to use in the implementation phase [32], [33]. All these patterns and guidelines have been defined based on the experience of the developers. Yoshioka in [31] showed that there is no consistent set of patterns that are easily traceable during software development. Therefore, our research on traceable patterns and code smells will play an important role in ensuring software security.

Although prior studies focused on the associations between code smells and micro patterns, they lack in presenting the comparative analysis of classes with code smells and classes without code smells. We examine the correlations between micro patterns and code smells and compare our results with previous studies. 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. In addition, as traceable patterns have been investigated for vulnerability proneness which may be caused by code smells, this study between traceable patterns and code smells may emerge as a new research area in software security.

4. Methodology

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

4.1. *Research Goal*

The goal of the study is to determine if there exists a correlation between code smells, micro patterns, and nano-patterns. This goal is addressed by the research questions presented in Section 4.2.

4.2. *Research Questions*

4.2.1. *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?

4.2.2. *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?

4.3. *Systems*

The study was conducted using three versions of Apache Tomcat[a], three versions of Apache CXF[b], and two stand-alone Java web applications: Personalblog 1.2.6 and Roller 0.9.9 from Stanford SecuriBench[c]. Apache Tomcat is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and

[a]http://tomcat.apache.org/
[b]http://cxf.apache.org/
[c]https://suif.stanford.edu/ livshits/securibench/intro.html

Table 4. Statistics of the Systems

Projects	Systems	Classes		Methods	
		Code Smell	**Non-Code Smell**	**Code Smell**	**Non-Code Smell**
Apache	Tomcat	117	7848	473	37226
	CXF	78	14683	1171	86569
Stanford	PersonalBlog	9	28	36	76218
	Roller	33	238	0	33

Java WebSocket technologies and powers numerous large-scale, mission-critical web applications. Apache CXF is an open source services framework which helps in developing services. The source codes for Apache Tomcat and Apache CXF are located in Apache Archives of Tomcat[d] and CXF[e], respectively. Stanford SecuriBench is a set of open source programs to be used as a testing ground for static and dynamic security tools [40]. We conducted the experiment for two J2EE web applications: Personalblog 1.2.6 and Roller 0.9.9 in Stanford dataset. Their source code is located at Stanford website[f]. The statistics of all the systems under study are presented in Table 4. 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.

4.4. *Data Extraction*

Step 1: Extract code smells: We used Intooitus inCode[g] to extract the code smells from the source code. inCode follows Marinescu's detection strategies to quantify design problems [18], [20]. The detection strategies implemented by inCode were comprehensively described by [18], [20], and this was one of the decisive factors for selecting the tool. inCode has been used for detecting code smells in many studies [10], [14], [2], [27],[26],[23]. In addition, inCode tests for code smells that are most commonly encountered in software projects. It extracts the following specific class-level code smells: God class, Data class and Schizophrenic class and method-level code smells: Data Clump, Duplication and Feature Envy.

Step 2: Extract micro patterns: The micro pattern tool as described in [15] is interfaced via the command line. It accepts a class name or .jar file as input and extracts all micro patterns identified in that class or classes in the jar file. The command line tool is available at Maman's webpage[h]. If a particular micro pattern exists in that class, the respective entry is '1' otherwise, it is '0'. For Apache Tomcat, we collected the micro pattern data for a total of 117 "code smell classes" and 7848 "non-code smell classes." For Apache CXF, we collected micro pattern data for

[d]http://archive.apache.org/dist/tomcat/
[e]http://archive.apache.org/dist/cxf/
[f]https://suif.stanford.edu/ livshits/securibench/download.html
[g]https://www.intooitus.com,its evolution at http://www.aireviewer.com
[h]http://www.cs.technion.ac.il/ imaman/mp/download.html

78 “code smell classes” and 14683 “non-code smell classes.” For PersonalBlog, we collected micro pattern data for nine “code smell classes” and 28 “non-code smell classes.” For Roller, we collected micro pattern 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 [24]. Deo modified the original version of the tool by providing input via a .properties file instead of command line arguments and stored the results in a database [5]. For Apache Tomcat, we collected the nano-pattern data for a total 473 “code smell methods” and 37226 “non-code smell methods.” For Apache CXF, we collected nano pattern data for 1171 “code smell methods” and 86569 “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.”

4.5. *Data Analysis*

4.5.1. *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 Apache Tomcat, there are 10 classes with the *Record* micro pattern of the 117 total code smell classes. This micro pattern exists in 8.55 percent of code smell classes. For illustration purposes, we show the distributions of micro patterns in the code smell and non-code smell classes for Apache Tomcat in Figure 1.

For RQ1.2, we separated the classes affected by each type of code smells: data class, schizophrenic class and god class. We then computed the percentage of each micro pattern in those classes. For example, in Apache 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 percent. For illustration purposes, we show the relations between different micro patterns and three types of code smells for Apache Tomcat in Figure 2.

4.5.2. *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 Apache Tomcat, there are 192 methods having *ObjCreator* nano-pattern out of 473 methods having code smells. So 40.6 percent of the code smell methods have the *ObjCreator* nano-pattern. For illustration purposes, we show the distributions of nano patterns in the code smell and non-code smell methods for Apache Tomcat in Figure 3.

Regarding RQ2.2, we separated the methods affected by the following types of code smells: data clumps, duplication and feature envy. We then computed the percentage of each nano-pattern in those methods. For example, in Apache 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-patterns in data clump code smell is 80.7. For illustration purposes, we show the relations between different nano-patterns and three types of code smells for Apache Tomcat in Figure 4.

For RQ1.1 and RQ2.1, we also performed a chi-squared test of independence to compare the difference in the distributions 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 $\alpha = 0.05$.

$H0_1$ *The distribution of micro patterns is independent of code smell and non-code smell classes.*

$H0_2$ *The distribution of nano-patterns is independent of code smell and non-code smell methods.*

We performed separate chi-squared tests for independence for the different types of micro patterns and nano-patterns.

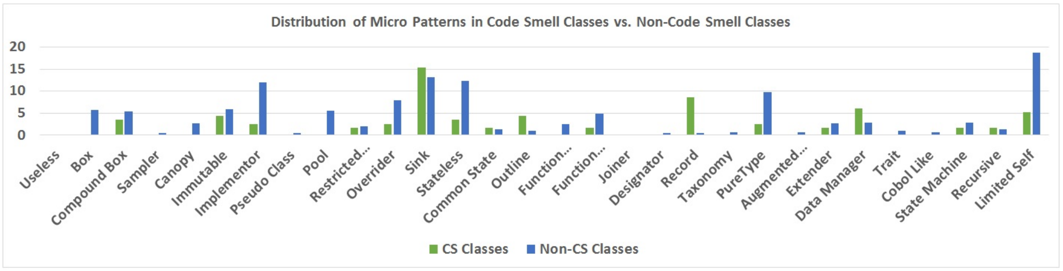

Fig. 1. Micro patterns in Code Smell vs. Non-Code Smell Classes for Apache Tomcat

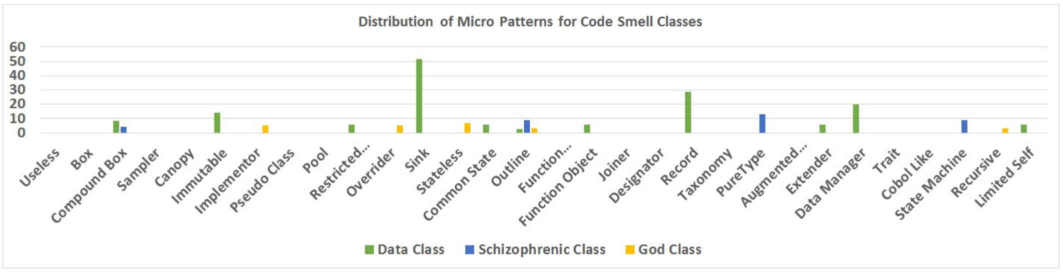

Fig. 2. Micro Patterns in Different Types of Code Smell Classes for Apache Tomcat

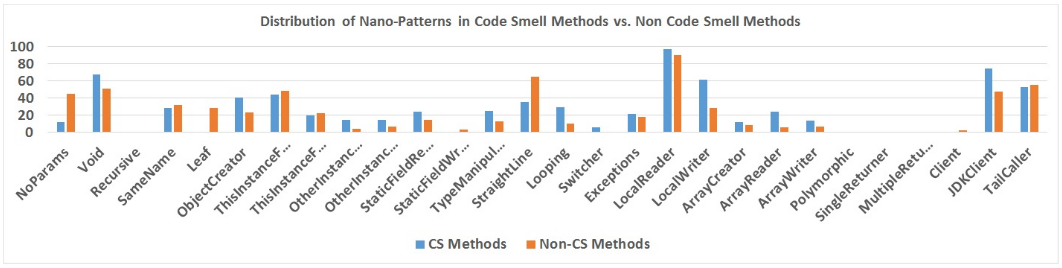

Fig. 3. Nano-Patterns in Code Smell vs. Non-Code Smell Methods for Apache Tomcat

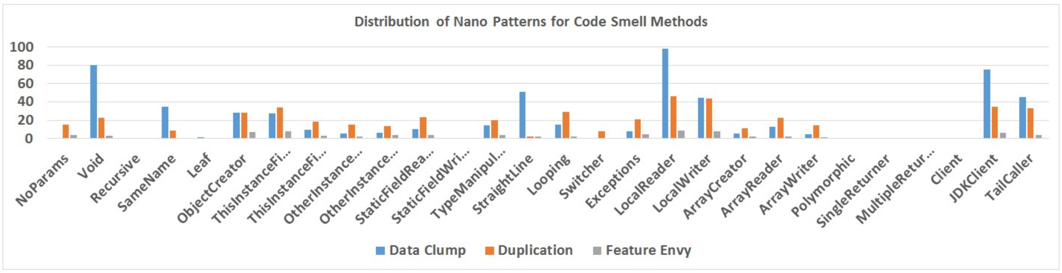

Fig. 4. Nano-Patterns in Different Types of Code Smell Methods for Apache Tomcat

5. Results

5.1. *Research Question 1 (RQ1)*

The test statistic for the chi-squared 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-squared statistic will be close to zero. If the null hypothesis is false, then the chi-squared statistic will be large. For degree of freedom of 1 and at 5 percent level of significance, the appropriate critical value is 3.84 [41], and the decision rule is as follows: Reject H_0 if $\tilde{\chi}^2 \geq 3.84$. We computed the chi-squared 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 findings as follows:

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

- For Apache CXF, the micro patterns that are frequent in the code smell classes compared to the non-code smell classes are *Box*, *Immutable*, *Implementor*, *Sink*, *PureType* and *Extender*.

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* and *DataManager*.
- The micro patterns more frequent in the classes having schizophrenic class code smell are *CompoundBox*, *Outline*, *PureType* and *StateMachine*.
- The micro patterns more frequent in the classes having god class code smell are *Implementor*, *Overrider* and *Stateless*.

For PersonalBlog,

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

For Apache CXF,

- The micro patterns that are more frequent in the classes having data class code smell are *Sink*, *DataManager*, *PureType* and *CompoundBox*.
- The micro pattern more frequent in the classes having schizophrenic class code smell is *DataManager*.
- The micro patterns more frequent in the classes having god class code smell are *Canopy*, *Stateless* and *Outline*.

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

5.2. *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-squared test statistics for nano-patterns. We rejected 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 the 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 the code smell methods compared to the non-code smell methods are *LocalReader* and *LocalWriter*.
- For Apache CXF, all the nano-patterns except *Polymorphic*, *SingleReturner* and *MultipleReturner* are more frequent in the code smell methods compared to the non-code smell methods.

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*.

For Apache CXF,

- The nano-patterns more frequent in the methods having data clump code smell are *Void*, *StraightLine*, *LocalReader*, *LocalWriter*, *JdkClient* and *TailCaller*.
- The nano-patterns that are more frequent in the methods having duplication code smell are *Void*, *SameName*, *StraightLine*, *LocalReader*, *LocalWriter* and *TailCaller*.
- The nano-patterns more frequent in the methods having feature envy code smell are *Void*, *SameName*, *ObjCreator*, *StaticFieldReader*, *TypeManipulator*, *StraightLine*, *Looping*, *LocalReader*, *LocalWriter*, *JdkClient* and *TailCaller*.

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

6. Discussion

6.1. *Code Smell and Micro Patterns*

According to Section 5.1, the micro patterns that are frequent in the code smell classes compared to the non-code smell classes are *Sink*, *Record*, *Immutable*, *Outline*, *Implementor* and *Extender*. Fontana [11] found that the *Outline* micro pattern often results in *SignificantDuplication* code smell (Confidence level=76%). Gil et al. [15] defined an *Outline* pattern as an abstract class where two or more declared methods invoke at least one abstract method of the current ("this") object. Kim et al. [8] also observed the *Outline* micro pattern as a defective pattern as they found high defect rates in classes having this pattern. On the other hand, *Record* micro pattern is declared as an anti-pattern by Destefanis in [7] as it is associated to bad programming practices.

Data classes are classes which contain only fields, getters / setters or public fields. *DataManager* is a class where all methods are either setters or getters [15]. *Record* is a class in which all fields are public with no declared methods. *Sink* is a class whose methods do not propagate calls to any other class. An *Immutable* class is a class whose instance fields can only be changed by its constructors. According to Section 5.1, *DataManager*, *Record*, *Sink*, and *Immutable* are among the micro patterns that are more frequent in the classes having data class code smell. The definitions of these micro patterns support our findings relating 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 getting 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. We also found *CompundBox*, *FunctionObject*, *RestrictedCreation*, *Extender* and *CommonState* as frequent in classes having data class code smell. Other researchers [6], [8] also detected them as fault-prone micro patterns. Fontana in [11] showed *Sink*, *Immutable*, *CompundBox*, *Extender* and *FunctionObject* to be associated with data class code smells.

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. According to Section 5.1, *CompoundBox*, *Outline*, *PureType*, *StateMachine* and *DataManager* are among the micro patterns that are more frequent in the classes having schizophrenic class code smell. The class with *PureType* pattern has nothing more than four abstract methods which concrete subclasses must override [15]. *StateMachine* pattern is an interface to define only parameter-less methods. Such an interface allows client code to either query the state of an object or request the object to change its state in some predefined manner [15]. Therefore, all these types of micro patterns can result in non-cohesive interfaces.

Canopy, *Stateless*, *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 [15]. The use of these patterns increases the possibility of having non-cohesive environment. *Canopy* and *Stateless* patterns have been presented a fault-prone patterns in [6].

6.2. *Code Smells and Nano-patterns*

According to Section 5.2, 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 [24]. Our results are in line with the findings of Deo [5] and Sultana et al. [30] which reported that methods with the *ArrReader* and *LocalWriter* nano-patterns are highly defect-prone.

Data Clumps refer to the large groups of parameters that appear together in the signatures 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 a stack frame. *JdkClient* calls methods from the JDK standard library (java.*). *TailCaller* contains a method call followed immediately by a return statement [24]. 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 a stack frame [24]. *LocalWriter* has been associated with a high defect density [5] 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 manipulates a lot of data external to its definition scope. In object-oriented code, this is a method that uses many data members from 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 a high defect density as well in other related research [30], [5]. Sultana et al. [30] showed that defective methods contain *LocalWriter*, *ObjCreator*, *JdkClient* together. Those methods were marked as having SQL Injection vulnerability as string concatenation is not allowed in queries. Therefore, as in [30], we can also conclude that methods where a local variable is assigned a string value or a newly-created object should be rigorously tested for code smells.

7. Threats to validity

Construct Validity: Construct validity refers to the degree to which a test measures what it claims. In this study, we used micro and nano-patterns to find their relation with code smells. Although micro patterns are defined on the formal conditions of the structure of a Java class, they may not capture all types of class characteristics. Moreover, the micro and nano-patterns detection tools mine only the presence and absence of patterns in source code. The binary method of detection instead of numerical scale can be considered as threats to validity for this study. In addition, the extraction process we used for code smells was based on the documented coding standards and code smell detection strategies included in the software tools. We understand that these strategies are limited to the process defined by the tool and consider this a threat to construct validity.

External Validity: External Validity refers to the ability to generalize results. The experiment was conducted on six versions of two Apache projects and two vulnerable J2EE web applications. Micro patterns are defined only for Java classes. We considered Apache projects because they are Java-based systems, and their vulnerability data are available along with class details. Therefore, it cannot be concluded that the results can be generalized for other systems written in different programming languages or different frameworks.

Internal Validity: This threat refers to the possibility of having unwanted or unanticipated relationships. We are not claiming causation, rather relating traceable patterns with the presence of code smells. In section 6, we explain why some smells are proportionally related to certain traceable patterns. Therefore, it can be assumed that these classes and methods contain some structural properties that are related to defective code. This analysis will help testers and developers to provide more effort on the patterns related with code smells.

8. Conclusion

This study represents 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 and methods, respectively. Our results can be used for identifying micro patterns and nano-patterns that may be related with the 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.

In addition, 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 *Proc. of the 2010 IEEE International Conference on Software Maintenance* (*ICSM '10*), pages 1–10. IEEE Computer Society, 2010.

[2] L. Amorim, E. Costa, N. Antunes, B. Fonseca and M. Ribeiro, Experience report: Evaluating the effectiveness of decision trees for detecting code smells, *IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Gaithersbury, MD, 2015, pp. 261-269.

[3] F. Batarseh, Java nano patterns: a set of reusable objects, in *Proc. of the 48th Annual Southeast Regional Conference*, New York, NY, USA, 2010.

[4] Z. Codabux, K.Z. Sultana, and B.J. Williams. The Relationship between Traceable Code Patterns and Code Smells, *29th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, PA, 2017.

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

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

[7] G. Destefanis, *Assessing software quality by micro patterns detection*, PhD Thesis, University of Cagliari, 2012.

[8] S. Kim, K. Pan, and E. Whitehead Jr., *Micro pattern evolution*, Proceedings of the International Workshop on Mining Software Repositories, pages 4046, 2006.

[9] 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.

[10] F. A. Fontana, et al. On experimenting refactoring tools to remove code smells. *Scientific Workshop Proceedings of the XP2015*. ACM, 2015.

[11] 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.

[12] 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.

[13] 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.

[14] G. Ganea, I. Verebi, and R. Marinescu. Continuous quality assessment with inCode. *Science of Computer Programming*, 134 (2017): 19-36.

[15] J. Y. Gil and I. Maman, Micro patterns in java code, in *Proc. of the 20th Annual*

ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (*OOPSLA '05*), pages 97–116. ACM, 2005.

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

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

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

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

[20] 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.

[21] 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.

[22] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus and T. Dyb, Quantifying the Effect of Code Smells on Maintenance Effort," *in IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144-1156, Aug. 2013.

[23] J. Singer, G. Brown, M. Lujn, A. Pocock, and P. Yiapanis, Fundamental nanopatterns to characterize and classify java methods, *Electronic Notes in Theoretical Computer Science*, 253(7):191–204, 2010.

[24] J. Singer, and C. Kirkham, Exploiting the correspondence between micro patterns and class names, in *Proc. of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 67–76, 2008.

[25] A. Yamashita and M. Leon. To what extent can maintenance problems be predicted by code smell detection?An empirical study. *Information and Software Technology* 55.12 (2013): 2223-2242.

[26] 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.

[27] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, Investigating the impact of design debt on software quality, in *Proc. of the 2nd Workshop on Managing Technical Debt* (*MTD '11*), pages 17–23. ACM, 2011.

[28] K. Z. Sultana, A. Deo, and B. J. Williams, A Preliminary Study Examining Relationships between Nano-Patterns and Software Security Vulnerabilities, in *Proc. of the 40th IEEE Computer Society International Conference on Computers, Software and Applications*, Atlanta, Georgia, USA, June 10-14, 2016.

[29] K. Z. Sultana, A. Deo, and B. J. Williams, Correlation Analysis among Java Nanopatterns and Software Vulnerabilities, in *Proc. of the 18th IEEE International Symposium on High Assurance Systems Engineering*, Singapore, January 12-14, 2017.

[30] N. Yoshioka, H. Washizaki, and K. Maruyama, A survey on security patterns, in *Progress in Informatics, Special issue: The future of software engineering for security and privacy*, vol. 5, pp. 35-47, October, 2008.

[31] M. G. Graff and K. R. Wyk, Secure Coding: Principles and Practices, in *Chapter 4: Implementation*, pp. 99–123, OReilly, 2003.

[32] R. C. Seacord, Secure Coding in C and C++, Addison Wesley, 2006.

[33] J, Yoder and J. Barcalow, Architectural Patterns for Enabling Application Security, in *PLoP*, 1997.

[34] M. Bunke, Software-security patterns: degree of maturity, in *Proc. of the 20th European Conference on Pattern Languages of Programs*, Kaufbeuren, Germany, July 08-12, 2015.

[35] Md. R. Islam, and M. F. Zibran, A Comparative Study on Vulnerabilities in Categories of Clones and Non-cloned Code, in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 3, pages 8–14, 2016.

[36] Y. Shin, A. Meneely, L. Williams, and J. Osborne, Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities, in *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, November/December, 2011.

[37] I. Chowdhury, B. Chan, and Mohammad Zulkernine, Security metrics for source code structures, in *Proc. of the fourth international workshop on Software engineering for secure systems*, pp. 57–64, Leipzig, Germany, May 17-18, 2008.

[38] I. Chowdhury and M. Zulkernine, Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities, in *Journal of Systems Architecture*, Volume 57, Issue 3, March 2011, Pages 294–313.

[39] V. B. Livshits and M. S. Lam, Finding security errors in Java programs with static analysis, in *Proc. of the 14th Usenix Security Symposium*, pp. 271–286, Aug, 2005.

[40] A. Field, Z. F., and J. Miles, Discovering statistics using R, 2012 (2012).