IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 23, 2020, accepted December 20, 2020, date of publication December 28, 2020,
date of current version January 7, 2021.

Digital Object Identifier 10.1109/ACCESS.2020.3047870

The Effect of Code Smells on the Relationship Between Design Patterns and Defects

TAREK ALKHAEIR AND BARTOSZ WALTER
Faculty of Computing and Telecommunications, Poznań University of Technology, 60-965 Poznań, Poland
Corresponding author: Tarek Alkhaeir (tarek.y.alkhaeir@doctorate.put.poznan.pl)

This work was supported by Poznań University of Technology under Grant 0311/SBAD/0702.

ABSTRACT The relationship between design patterns and defects was investigated in the literature, but with mixed results. While the majority of studies found the presence of patterns to be positively correlated with defects, other works reported the opposite conclusions. This may suggest that contextual factors affect this relationship. In this study we analyze the role of code smells as a confounding variable in the relationship between design patterns and defects in Java classes. To investigate this, we applied statistical tests to capture the difference in the impact on defects between pattern classes with/without code smells in 10 Java systems from the PROMISE dataset, with respect to 13 design patterns and 10 code smells. The presence of code smells in patterns appears to be a valid factor affecting defect-proneness. Pattern classes with smells attract more defects than non-smelly pattern classes, and in most systems smelly design patterns are positively correlated with the presence of defects, while non-smelly patterns tend to have no impact, or a slightly negative impact on the presence of defects. As a result, the presence of code smells in design pattern classes appears to be a contextual factor affecting the defect-proneness of the subject code.

INDEX TERMS Code smells, design patterns, defect proneness.

I. INTRODUCTION

Design patterns are generic object-oriented solutions to frequently occurring design problems. They were first introduced in the Gang of Four's (GOF) book [18], which promoted design patterns as implementations of good practices that can improve re-usability, maintainability, and understandability.

Over the years, the relationship between design patterns and measurable quality attributes has attracted the interest of many researchers. While several studies found patterns to be positively correlated with various code characteristics, such as maintainability [42], defects [52] or the absence of code smells [53], other studies reported the negative impact of patterns on software maintainability [54] and code evolution [28]. Additionally, several other studies were inconclusive or attributed the observed effects to the nature of the patterns, to the way they are used, or to other contextual factors [6], [39]. Those findings not only conjecture that the relationship between design patterns and code quality attributes is not direct, but they also suggest that various contextual factors may have played an important role in this relationship. In the current study, we investigate whether the presence of code smells could be a contextual factor in the relationship between patterns and defects.

Code smells [17] are surface-level indicators that may correspond to deeper problems in software systems. The relationship between smells and code quality attributes drew researchers' attention and several studies were conducted to investigate this relationship. Some of these studies reported the negative impact of smells on change/fault proneness [31], [41], defect proneness [11] and maintainability [55]. On the other hand, other studies found that the detrimental effect of smells on various quality characteristics is limited or even positive, e.g., defect proneness [30], changeability [38] and maintenance effort [47].

As in the case of design patterns, the relationship between smells and quality attributes is not decisively defined, which constitutes another reason for conducting this study. Moreover, besides the importance of researching the context in which the defects may exist, another motivation drove our study; the phenomena in question, i.e., patterns and smells, are different in nature: while design patterns are intentionally used to achieve specific design objectives, code smells are inadvertent side effects of software development.

Defects, colloquially called bugs, are conditions in software products which do not meet the requirements or the customers' expectations. Defects often arise because of an

The associate editor coordinating the review of this manuscript and approving it for publication was Taehong Kim.

incomprehensive understanding of the code, a fault interpretation of the business requirements, programmers' mistakes or due to frequent code changes [2]. The process of detecting and removing software defects is an important step in guaranteeing the fulfilment of end user satisfaction [20] and reducing the economic liability associated with releasing flawed software products [26]. Furthermore, building an efficient defect prediction model has raised increased interest from researchers in their quest to learn from the previous defects and to use this knowledge to predict future ones [15]. Defects were also heavily investigated in the literature, with several studies considering defects as a dependent variable affected by various independent variables, such as patterns [6], [21], [52], bad design [57] and the presence of code smells [30]. Defects should not be confused with code smells. They refer to two distinct quality characteristics: defect to reliability and smells to maintainability. In our study we consider them separately.

This article focuses on the impact of smelly or non-smelly design patterns as independent variables on defects, considered as binary variables (i.e., present/absent) or quantitative variables (i.e., the number of defects). We seek to determine the confounding effect of smells on code that contains design patterns, in terms of the resulting defects.

The paper is structured as follows. In Sec. II we present the literature review on the relationship between patterns and smells, and their separate effect on defect-proneness. Next, in Sec. III, we describe the design of the experiment and formulate the research questions. The results are presented in Sec. IV, then in Sec. V we discuss and interpret them. Finally, in Sec. VII we provide conclusions and propose possible directions for future investigation.

II. RELATED WORK

While the relationship between design patterns and code smells is a relatively new topic, both smells and patterns have been separate subjects of research for a long time. Below we summarize the major findings concerning their relationship with each other and their separate links with defects.

A. DESIGN PATTERNS AND DEFECTS

A number of studies investigated the relationship between design patterns and defects. However, the results were mixed and sometimes contradictory. Vokáč [52] observed the evolution of a commercial product for three years in an attempt to compare the defect ratios in classes with selected design patterns, namely Observer, Singleton, Factory and Template Method, with other classes. They concluded that both Observer and Singleton are more defect-prone than other patterns due to the larger code structures they contain. On the other hand, the Factory pattern displayed a lower number of defects than other classes, and the results for Template Method were inconclusive. Moreover, Gatrell and Counsell [19] examined a subset of a large commercial software system written in C♯ and found that classes implementing design patterns are more fault-prone than the non-pattern classes. In particular, the authors found some patterns, namely Adaptor, Template Method and Singleton to be more defect-prone than others.

Elish and Mohammed [13] found no difference in the fault density between classes participating in the creational or behavioural patterns and classes without patterns, while structural patterns appeared to have a lower fault density than other classes. Furthermore, Onarcan and Fu [39] presented a study of 26 open source software projects, showing that there is a little correlation between the number of pattern instances in those projects and the number of defects. They also concluded that individual design patterns may have either positive or negative impact on defect-proneness.

Additionally, Aversano *et al.* [6] reported an empirical study involving three open source systems and concluded that the defect density of pattern classes is higher if the implementation of those patterns includes crosscutting concerns. The study also asserted that the relationship between design patterns and defects varies depending on the type and the nature of the pattern.

B. CODE SMELLS AND DEFECTS

The relationship between code smells and defects has also attracted the attention of researchers. Li *et al.* [30] conducted an experiment to investigate, at a class level, the correlation between smells and defects. They reported that the presence of some code smells, e.g., God Class and Shotgun Surgery, is positively correlated with defects proneness, while there is no correlation for other smells, such as Data Class and Feature Envy. Furthermore, Olbrich *et al.* [37] monitored the development of three open source systems for 7-10 years and concluded that God and Brain classes tend to change more frequently and have more defects than other classes. However, after the values have been adjusted to the class size, the findings were reversed.

Bán and Ferenc [7] also addressed the effect of code smells on defects. Their study analyzed several systems from the PROMISE dataset and their aggregated results showed a positive correlation between the studied code smells and defects. Furthermore, Jaafar *et al.* [25] studied three systems, Azureus, Eclipse and JHotDraw, and reported that the majority of classes affected by code smells tend to be more fault-prone than the smell-free classes. Another large scale empirical investigation was performed by Palomba *et al.* [41]. The results show that smelly classes have a higher fault-proneness than non-smelly classes. A similar conclusion was reported by Nascimento and Sant'Anna [35].

On the other hand, Hall *et al.* [23] argued that the presence of code smells in some circumstances may indeed indicate a fault-prone code. However, the effect of those smells on the defects is rather minor. The authors also suggested that refactoring smelly classes is unlikely to reduce the number of defects in the affected code. A similar observation was reported by D'Ambros *et al.* [11] who concluded that none of the studied smells could be considered more harmful with respect to software defects.

Tufano *et al.* [51] addressed the issue from a different perspective; they investigated the reasons behind the introduction of smells, and to this end they studied the change history of 200 open source projects, concluding that in many cases the refactoring and bug-fixing activities lead to the introduction of smells.

Finally, Cairo *et al.* [8] performed an SLR aimed at the analysis of the impact of code smells on defects. They concluded that the correlation between smells and defects is weaker than had been conjectured, and that no individual code smell could be identified as positively correlated with increased defect frequency.

C. DESIGN PATTERNS AND CODE SMELLS

In our previous study [53], we investigated the relationship between design patterns and code smells at a class level and the results indicated that pattern classes are less frequently affected by smells. The study also found that the strength of the relationship varies among different patterns and smells.

A study by Cardoso and Figueiredo [9] found two pattern-smell links: Command with God Class and Template Method with Code Duplication. The study also presented a possible explanation of those co-occurrences and offered guidelines on how to eliminate the smells from these patterns. Additionally, a recent study by Alfadel *et al.* [4] found that design pattern classes are less smell-prone than other classes. However, Command pattern classes are associated with God Class, Blob and External Duplication smells.

Furthermore, Sousa *et al.* [49] investigated if the use of patterns is negatively correlated with occurrences of code smells, and examined their collocations in the same classes. They concluded that the use of patterns does not prevent code smells. The study also reported that some patterns, such as Composite, Factory Method and Singleton, are intrinsically modular and they produce code of higher quality, while other patterns, such as Adapter, Proxy and State/Strategy, are linked with a higher frequency of code smells.

Finally, Sousa *et al.* presented a systematic mapping study on the relationship between patterns and smells [48]. They identified 16 papers and concluded that the misuse of certain patterns is the main cause of their co-occurrence with code smells. The paper also found that the Command pattern is correlated with the greatest number of smells.

III. EXPERIMENTAL DESIGN

A. RESEARCH QUESTIONS

In the study we consider three research questions that examine the defect-proneness of pattern classes, depending on the presence or absence of code smells in them.

1) **RQ1** What is the impact of code smells on the presence/absence of defects in classes involved in design patterns?
2) **RQ2** What is the impact of code smells on the defect distribution (number of defects) in classes involved in design patterns?
3) **RQ3** What is the effect of code smells on the relationship between specific design patterns and defects?

B. VARIABLES

The definitions of variables we use in the analysis differ, depending on the construct employed in a specific research question.

1) INDEPENDENT VARIABLES (IVs)

For RQ1 and RQ2, we consider sets of classes C, with respect to their involvement in *any* design pattern or if they are affected by *any* code smells; this produces four values of the IV: *DP*, *nDP*, *SDP* and *nSDP*;

In RQ3 the definitions are different: we consider classes involved in *specific* patterns that are affected by *any* smell. That also produces four values of IVs, but separately for each DP. For example, for Decorator we have four values of the IV: $DP_{Decorator}$ – all classes involved in Decorator, *nDP* – classes not involved in *any* pattern, $SDP_{Decorator}$ – smelly classes involved in Decorator, and $nSDP_{Decorator}$ – non-smelly classes involved in Decorator.

2) DEPENDENT VARIABLES (DVs)

For RQ1 the DV is a *binary value* {0, 1} indicating the presence or absence of defects in a class.

For RQ2 the DV is the *count* of defects in the class.

For RQ3 we use both definitions of DVs.

For RQ1 and RQ2 we consider each system separately. All subject systems are developed by communities, with their own habits, standards and routines, which cannot be directly identified. As such, they should be considered latent variables that could bias the results. For that reason, we decided to report the results separately for each system. However, for RQ3, due to the small size of data samples, we merged the systems into a single dataset.

C. NOTATION

To describe the sets that define the variables, we use the following notation:

- *ALL*: All analyzed classes
- *DP*: Classes involved in one or more design pattern, in any role;
- *nDP*: Classes not involved in any design pattern;
- *SDP*: Classes simultaneously involved in design pattern(s) and affected by code smell(s);
- *nSDP*: Classes involved in design pattern(s) and not affected by code smell(s);
- *SnDP*: Classes not involved in design pattern(s) and affected by code smell(s);
- *nSnDP*: Classes not involved in design pattern(s) and not affected by code smell(s);
- *DEF*: Classes with at least one defect;
- *DEF-DP*: Classes involved in design pattern(s) and with at least one defect;
- *DEF-nDP*: Classes not involved in design pattern(s) and with at least one defect;

TABLE 1. List of subject systems.

System	Description
Ant-1.7	Java library and command-line tool to compile, assemble, test and run Java applications.
JEdit-4.2	A modular and extensible text editor with hundreds of customizable plugins.
Lucene-2.4	Java library for performing advanced indexing and searching.
Camel-1.6	A message-oriented middleware and integration framework that provides an object-based implementation of the enterprise integration patterns.
Log4j-1.2	An extensible logging framework for Java applications.
Xalan-2.7	An XML processor for applying XSL transformations and XPath queries.
Poi-3.0	A library for manipulating MS Office documents.
Ivy-2.0	An extensible dependency manager.
Xerces-2.0	An XML parser.
Velocity-1.6	A template engine with a built-in expression language.

TABLE 2. Design patterns analyzed in the study. Category: C–Creational, B–Behavioral, S–Structural.

Name (Category)	Description
Composite (S)	Defines tree-like structures that represent part-whole hierarchies.
Prototype (C)	Allows the creation of a prototype instance and clone it later to produce new instances.
State-Strategy (B)	State: allows a class to change its behavior in response to a change in its state. Strategy: encapsulates a family of algorithms and makes them interchangeable.
Factory Method (C)	Defines the type of created object, but defers its initiation to the subclasses.
Template Method (B)	Defines an abstract base class to represent an algorithm and postpones the implementation to its subclasses.
Decorator (S)	Enables an object to be wrapped by other objects, to allow adding or modifying its features.
Singleton (C)	Creates a single instance of a class and ensures its uniqueness.
Proxy (S)	Controls access to the object by providing an intermediate layer that intercepts requests sent to the object.
Adapter (S)–Command (B)	Adapter: Allows two classes with incompatible interfaces to work together. Command: Allows clients to be parameterized with objects representing operations.
Observer (B)	Creates a one-to-many relationship between a subject and its observers. When the subject changes its state, its observers are notified.
Visitor (B)	Allows for handling non-related objects within a collection.
Chain Of Responsibility (B)	Decouples the sender and receiver of a request by creating a chain of request handlers.
Bridge (S)	Decouples abstraction from its implementation, so that the two can change independently.

D. SUBJECT SYSTEMS

We performed our analysis on 10 small- and medium-size Java systems coming from PROMISE [1], one of the largest public repositories of empirical software data. In this study we use one of the datasets that provides information about defects. The original dataset includes 14 open source java systems: Ant, Camel, Ckjm, Forrest, Ivy, JEdit, Log4J, Lucene, PBeans, Poi, Synapse, Velocity, Xalan and Xerces. We decided to exclude four systems: Ckjm, PBeans, Synapse and Forrest, due to the negligible number of patterns (< 5) in them.

The list of systems used in the study is presented in Table 1.

E. DESIGN PATTERNS

To identify the design patterns we used a pattern-detection tool written by Tsantalis.[1] This tool uses the Similarity Scoring Approach (SSA), which calculates the similarity between subject code and graphs representing canonical patterns. If the score exceeds a defined threshold value, the pattern is positively identified [50].

Specifically, we used the most recent version of the tool, v4.12. It was verified against several Java systems with a reported precision of 100% and a recall of 66.7-100 % [3], which makes its performance comparable to other approaches that use *exact/inexact graph matching* like Discovery Matrix (DP-Miner) [12], the *sub-patterns approach* [56], or *metrics-based approaches*, e.g., MAISA [40] and FUJABA [36].

Although the tool detects only selected GoF patterns, they cover all three categories in the GoF taxonomy: creational, structural and behavioral [18]. In Table 2 we present the list of the analyzed design patterns.

To validate the effectiveness of the automatic pattern detection, we manually verified a random sample of ca. 400 pattern classes, which constitute ca. 10% of the cases identified by the tool. The review reported no false positives.

F. CODE SMELLS

Initially, code smells were mostly subject to human intuition and experience [17]. Today virtually all flaws can be detected automatically, based on various properties of the subject code or other artifacts. An SLR presented by Guéhéneuc and Albin-Amiot [22], based on 60 studies investigating various methods of smell detection, identified only four cases in which no automated approach was used.

In this article, for detecting code smells we used *inCode*, a proprietary Eclipse plugin that detects the smells based on the static code analysis. The tool employs an approach

[1] https://users.encs.concordia.ca/~nikolaos/pattern_detection.html

TABLE 3. Code smells analyzed in the study. Categories: B–Bloaters, C–Couplers, D–Dispensables, CP–Change Preventers. Levels: C–Class, M–Method.

Name (Acronym, Category, Level)	Description
God Class (GC, B, C)	Large, complex class with a number of lines of code, methods and attributes, that plays the role of a complex controller, associated with many data classes.
Feature Envy (FE, C, M)	References members of other objects more frequently than members of its own object.
Message Chains (MC, C, C)	Violates the Law of Demeter by featuring dependency on a chain of calls that connect objects.
Data Clumps (DCl, B, M)	A set of variables that frequently appear together as method parameters or class attributes.
Data Class (DC, D, C)	A class that stores attributes, but provides no actual behavior.
External Duplication (ED, D, M)	Duplication of code in different classes.
Schizophrenic Class (SC, CP, C)	A class with several unrelated responsibilities, which is used by several client classes.
Tradition Breaker(TB, CP, C)	A subclass does not use methods inherited from a superclass.
Internal Duplication (ID, D, M)	Duplication of code in a single class.
Sibling Duplication(SD, D, M)	Duplication of code in sibling classes.

TABLE 4. Numbers of classes that belong to respective datasets.

System	*ALL*	*DEF*	*DP*	*nDP*	*SDP*	*DEF-DP*	*DEF-nDP*
Ant-1.7	745	166	110	635	12	38	128
JEdit-4.2	366	47	66	300	11	15	32
Lucene-2.4	335	198	164	171	12	105	93
Camel-1.6	964	187	188	776	13	75	112
Log4j-1.2	205	189	44	161	5	43	146
Xalan-2.7	904	893	151	753	23	151	742
Poi-3.0	442	281	57	385	11	39	242
Ivy-2.0	350	38	101	249	13	18	20
Xerces-2.0	586	435	110	476	19	74	361
Velocity-1.6	229	78	89	140	8	40	38

TABLE 5. Results of the OR test and FET.

	OR		$log(OR)$		
System	*DP*	*nDP*	*DP*	*nDP*	*FET*
Ant-1.7	2.09	0.478	0.737	−0.737	Significant ($p = 0.002$)
JEdit-4.2	2.463	0.406	0.901	−0.903	Significant ($p = 0.014$)
Lucene-2.4	1.664	0.669	0.509	−0.401	Not significant ($p = 0.076$)
Camel-1.6	3.935	0.254	1.370	−1.370	Significant ($p < 0.001$)
Log4j-1.2	4.417	0	1.485	$-\infty$	Not significant ($p = 0.202$)
Xalan-2.7	∞	0	-	$-\infty$	Not significant ($p = 0.227$)
Poi-3.0	1.280	0.781	0.247	−0.247	Not significant ($p = 0.463$)
Ivy-2.0	2.483	0.403	0.909	−0.909	Significant ($p = 0.013$)
Xerces-2.0	0.654	1.527	−0.424	0.423	Not significant ($p = 0.070$)
Velocity-1.6	2.192	0.456	0.785	−0.785	Significant ($p = 0.007$)

TABLE 6. Summarized findings from the OR test and FET.

System	Extracted Rules
Ant-1.7	$DP \implies$ Defects, $nDP \implies \neg$ Defects
JEdit-4.2	$DP \implies$ Defects, $nDP \implies \neg$ Defects
Lucene-2.4	No significant rules
Camel-1.6	$DP \implies$ Defects, $nDP \implies \neg$ Defects
Log4j-1.2	No significant rules
Xalan-2.7	No significant rules
Poi-3.0	No significant rules
Ivy-2.0	$DP \implies$ Defects, $nDP \implies \neg$ Defects
Xerces-2.0	No significant rules
Velocity-1.6	$DP \implies$ Defects, $nDP \implies \neg$ Defects

called *detection strategies* [34], which relies on Boolean expressions composed of selected code metrics and respective thresholds. As in any metrics-based approach, the chosen threshold values play an important role in the accuracy of the detected instances. For this study, we used the default settings of inCode, following the recommendation by Lanza and Marinescu [29].

inCode has several advantages. First, the approach it implements for smell detection is commonly used in both academia and industry; additionally, the detection strategies were found to be fairly accurate in detecting smells ($\approx$ 70%, according to [34]), and a comparative study by Arcelli Fontana *et al.* [16] found inFusion, a commercial version of inCode that employs the same detection rules, to report the lowest number of false positives among four analyzed smell detectors.

In this study, we consider nine code smells (see Table 3) that represent four out of five smell categories in the taxonomy proposed by Mäntylä [33]: *Bloaters*, *Couplers*, *Dispensables*, and *Change preventers*.

G. DEFECTS

The defect dataset which we analyze in the study was acquired from the PROMISE repository. Data about defects in PROMISE has been collected by the Buginfo tool. It evaluates every commit in the repository of the analyzed system. The tool labels the commit as a bug fix if it solves an issue reported as a bug in the bug tracking system. For each analyzed project, the bug fixes commenting guidelines were discovered and formalized as regular expressions. Buginfo compares the regular expressions with the comment associated with the commit. If the comment matches the regular expression, Buginfo reports detecting a defect and increases the defect count for every class modified in the commit [27].

The PROMISE dataset has been validated and used several times in different research papers, e.g., concerning bug prediction [14], [15].

H. IDENTIFICATION OF DATASETS

In the study we analyzed classes with combinations of various properties.

First, we collected the classes in the *ALL* dataset, by identifying the fully qualified class names inside specific system releases within the source PROMISE dataset. Then, based on the results of the design pattern and code smell detection, described in III-E and III-F, we identified *DP* and *S* datasets.

TABLE 7. The OR test and FET results, considering the effect of code smells.

System	*SDP*		*nSDP*	
	OR	FET	OR	FET
Ant-1.7	5.054	Significant; $p = 0.007$	1.755	Significant; $p = 0.026$
JEdit-4.2	6.210	Significant;$p = 0.007$	1.646	Not significant; $p = 0.195$
Lucene-2.4	1.4	Not significant;$p = 0.767$	1.431	Not significant; $p = 0.119$
Camel-1.6	3.646	Significant; $p = 0.025$	3.701	Significant; $p < 0.001$
Log4j-1.2	∞	Not significant; $p = 1$	3.775	Not significant; $p = 0.317$
Xalan-2.7	∞	Not significant; $p = 1$	∞	Not significant; $p = 0.383$
Poi-3.0	∞	Significant; $p = 0.009$	0.879	Not significant; $p = 0.747$
Ivy-2.0	11.516	Significant; $p < 0.001$	1.243	Not significant; $p = 0.557$
Xerces-2.0	∞	Significant; $p = 0.006$	0.462	Significant; $p = 0.002$
Velocity-1.6	17.788	Significant; $p = 0.002$	1.573	Not significant; $p = 0.144$

TABLE 8. The findings from the OR test and FET, considering the effect of code smells.

System	Extracted rules
Ant-1.7	$SDP \implies DEF$ $nSDP \implies DEF$
JEdit-4.2	$SDP \implies DEF$
Lucene-2.4	no significant rules
Camel-1.6	$SDP \implies DEF$ $nSDP \implies DEF$
Log4j-1.2	no significant rules
Xalan-2.7	no significant rules
Poi-3.0	$SDP \implies DEF$
Ivy-2.0	$SDP \implies DEF$
Xerces-2.0	$SDP \implies$ Defects $nSDP \implies \neg\, DEF$
Velocity-1.6	$SDP \implies DEF$

TABLE 9. Descriptive statistics for the number of defects inside pattern and non-pattern classes.

System	*DP*		*nDP*	
	Mean	Median	Mean	Median
Ant-1.7	0.772	0	0.399	0
JEdit-4.2	0.772	0	0.18	0
Lucene-2.4	2.158	1	1.228	1
Camel-1.6	1.345	0	0.307	0
Log4j-1.2	2.909	2	2.298	2
Xalan-2.7	1.311	1	1.338	1
Poi-3.0	2.403	1	0.942	1
Ivy-2.0	0.277	0	0.096	0
Xerces-2.0	5.618	1.5	1.974	1
Velocity-1.6	1.146	0	0.628	0

Since the code smells can be attributed to classes or methods, while patterns involve classes, we had to adjust the granularity of the datasets to the class level by re-assigning the method-level smells to the enclosing classes.

In the next step we identified intersections of the sets to produce *SDP*, *SnDP*, *nSDP* and *nSnDP*. The resulting datasets were disjoint and complete, i.e., each class was reported exactly once in all datasets, and no class was omitted. However, defects in the PROMISE dataset are assigned to files, which can contain one or more classes. To address the issue of several classes being included in a single file, but assigned to different datasets, we manually verified and excluded such cases.

The replication package is available at `https://zenodo.org/record/3992730#.X2NZamgzY2x`

TABLE 10. Results of the Shapiro-Wilk test.

System	*DP*				*nDP*			
	W_{crit}	σ	W	p	W_{crit}	σ	W	p
Ant-1.7	0.976	1.566	0.540	≈ 0	0.995	1.040	0.441	≈ 0
JEdit-4.2	0.963	2.044	0.431	≈ 0	0.990	0.634	0.312	≈ 0
Lucene-2.4	0.983	2.706	0.768	≈ 0	0.984	1.591	0.766	≈ 0
Camel-1.6	0.985	3.276	0.440	≈ 0	0.996	1.084	0.308	≈ 0
Log4j-1.2	0.947	1.697	0.622	≈ 0	0.983	1.435	0.847	≈ 0
Xalan-2.7	0.948	1.736	0.673	≈ 0	0.995	0.743	0.571	≈ 0
Poi-3.0	0.958	4.105	0.580	≈ 0	0.992	1.263	0.601	≈ 0
Ivy-2.0	0.974	0.680	0.463	≈ 0	0.988	0.346	0.298	≈ 0
Xerces-2.0	0.976	9.515	0.619	≈ 0	0.993	3.039	0.545	≈ 0
Velocity-1.6	0.971	2.086	0.584	≈ 0	0.981	1.485	0.473	≈ 0

TABLE 11. Results of WMW and Hedges' g tests.

System	*DP* vs. *nDP*	Conclusion	Hedges' g
Ant-1.7	$z = -3.406, p < 0.001$	$DP > nDP$	0.330 (small)
JEdit-4.2	$z = -2.782, p = 0.005$	$DP > nDP$	0.570 (medium)
Lucene-2.4	$z = -3.091, p = 0.001$	$DP > nDP$	0.420 (small)
Camel-1.6	$z = -8.104, p < 0.001$	$DP > nDP$	0.595 (medium)
Log4j-1.2	$z = -2.202, p = 0.027$	$DP > nDP$	0.407 (small)
Xalan-2.7	$z = 0.599, p = 0.548$	none	
Poi-3.0	$z = -2.391, p = 0.016$	$DP > nDP$	0.775 (medium)
Ivy-2.0	$z = -2.745, p = 0.006$	$DP > nDP$	0.386 (small)
Xerces-2.0	$z = -2.106, p < 0.035$	$DP > nDP$	0.736 (medium)
Velocity-1.6	$z = -2.780, p = 0.005$	$DP > nDP$	0.295 (small)

IV. RESULTS

A. RQ1: WHAT IS THE IMPACT OF CODE SMELLS ON THE PRESENCE/ABSENCE OF DEFECTS IN CLASSES INVOLVED IN DESIGN PATTERNS?

In this section we check if design pattern classes are associated with the presence or absence (considered as a binary value) of defects in them. For this purpose, we used the Odds Ratio (OR) test [46] to find the associations between the presence of patterns as an exposure and the presence of defects as an outcome. The OR function is specified as follows: $OR = \frac{a/c}{b/d}$ where:

- a = number of exposed cases
- b = number of exposed non-cases
- c = number of unexposed cases
- d = number of unexposed non-cases

The result of the OR test is interpreted as follows:

- $OR = 1$ The exposure does not affect the odds of the outcome,
- $OR > 1$ The exposure is associated with higher odds of the outcome,

TABLE 12. Results of the WMW test for specific patterns, considering the effect of code smells.

System	*SDP* vs. *nSDP*	*SDP* vs. *nDP*	*nSDP* vs. *nDP*
Ant-1.7	$z = -2.557, p = 0.010$	$z = -3.764, p < 0.001$	$z = -2.460, p = 0.013$
JEdit-4.2	$z = -2.323, p = 0.020$	$z = -3.823, p < 0.001$	$z = -1.601, p = 0.109$
Lucene-2.4	$z = -0.724, p = 0.468$	$z = -1.702, p = 0.088$	$z = -2.916, p = 0.003$
Camel-1.6	$z = -0.059, p = 0.952$	$z = -3.049, p = 0.002$	$z = -2.232, p = 0.025$
Log4j-1.2	$z = 0.535, p = 0.592$	$z = -0.392, p = 0.695$	$z = -2.786, p = 0.005$
Xalan-2.7	$z = -4.991, p < 0.001$	$z = -3.945, p < 0.001$	$z = -2.289, p = 0.022$
Poi-3.0	$z = -2.839, p = 0.004$	$z = -3.801, p < 0.001$	$z = -0.923, p = 0.355$
Ivy-2.0	$z = -3.783, p < 0.001$	$z = -5.489, p < 0.001$	$z = -1.267, p = 0.204$
Xerces-2.0	$z = -4.769, p < 0.001$	$z = -6.246, p < 0.001$	$z = 0.305, p = 0.760$
Velocity-1.6	$z = -2.307, p = 0.021$	$z = -3.519, p < 0.001$	$z = -2.129, p = 0.033$

TABLE 13. The extracted rules from the WMW test results presented in Table 12 together with Hedges' g results.

System	*SDP* vs. *nSDP*	*SDP* vs. *nDP*	*nSDP* vs. *nDP*
Ant-1.7	*SDP* > *nSDP*. H'g=1.327 (large)	*SDP* > *nDP*. H'g=1.878 (large)	*nSDP* > *nDP*. H'g=0.156 (negligible)
JEdit-4.2	*SDP* > *nSDP*. H'g=1.319 (large)	*SDP* > *nDP*. H'g=2.765 (large)	not significant
Lucene-2.4	not significant	not significant	*nSDP* > *nDP*.H'g=0.420 (small)
Camel-1.6	not significant	*SDP* > *nDP*. H'g=0.284 (small)	*nSDP* > *nDP*.H'g=0.624 (medium)
Log4j-1.2	not significant	not significant	*nSDP* > *nDP*. H'g=0.404 (small)
Xalan-2.7	*SDP* > *nSDP*.H'g=1.379 (large)	*SDP* > *nDP*. H'g=0.926 (large)	*nSDP* > *nDP*. H'g=-0.224 (small)
Poi-3.0	*SDP* > *nSDP*. H'g=1.328 (large)	*SDP* > *nDP*. H'g=3.245 (large)	not significant
Ivy-2.0	*SDP* > *nSDP*. H'g=1.321 (large)	*SDP* > *nDP*. H'g=2.149 (large)	not significant
Xerces-2.0	*SDP* > *nSDP*. H'g=0.976 (large)	*SDP* > *nDP*. H'g=2.992 (large)	not significant
Velocity-1.6	*SDP* > *nSDP*. H'g=1.229 (large)	*SDP* > *nDP*. H'g=1.559 (large)	*nSDP* > *nDP*. H'g=0.195 (negligible)

- $OR < 1$ The exposure is associated with lower odds of the outcome.

Next, we use Fisher's exact test (FET) to determine if the OR results are significant [44]. The results are presented in Table 5 and the extracted association rules are summarized in Table 6.

The extracted rules indicate that patterns are positively associated with the presence of defects. This is in line with the results of other studies, e.g., [19], and in contradiction to the common understanding of patterns in terms of their positive impact on code quality [43]; on the other hand, they reinforce our conjectures concerning contextual factors that may play a role in this association. Our next step is to investigate the role of code smells as a confounding factor. Table 7 presents the results from the OR and FET tests and the extracted rules are listed in Table 8.

B. RQ2: WHAT IS THE IMPACT OF CODE SMELLS ON THE DEFECT DISTRIBUTION (NUMBER OF DEFECTS) IN CLASSES INVOLVED IN DESIGN PATTERNS?

For this question we are interested whether the presence of patterns in classes is associated with a higher/lower number of defects, and what the effect of smells as a contextual factor in this relationship is. To answer this, we first tested the normality of distributions in the subject datasets. In Table 9 we present the descriptive statistics of the datasets and in Table 10 we report the results of the Shapiro-Wilk test [45]. They show that both pattern and non-pattern values are not normally distributed.

Since all values are not normally distributed, we used a nonparametric Wilcoxon-Mann test (WMW) [32] to verify if two populations have different medians of defect distributions. In this section we verify the hypothesis that pattern and non-pattern classes do not differ with respect to defects. Analogously, similar hypotheses are used when comparing any other two groups using WMW throughout this study.

1) H0: $DP = nDP$ w.r.t. defects
2) Ha: $DP \neq nDP$ w.r.t. defects
3) Ha1: $DP < nDP$ w.r.t. defects
4) Ha2: $DP > nDP$ w.r.t. defects

The results of the WMW test are presented in Table 11.

As follows from the results, $DP > nDP$ for 9 out of the 10 analyzed systems, and the only remaining case is inconclusive. We repeated the same steps to measure the effect of smells on these results. In Table 12 we summarize the WMW test results between the smelly- and non-smelly pattern classes, and between the classes in those two groups and the non-pattern classes. We also present the extracted rules from this analysis in Table 13.

To assess the significance of our extracted rules, we performed an effect size analysis to measure the mean difference between the different groups. We used the Hedges' g [24] test with a corresponding 95% confidence interval (CI). Hedges' g provides a measure of the effect size weighted by the relative size of each sample. The results are interpreted according to Cohen's d conventions [10]:

- Negligible effect < 0.2
- Small effect = 0.2
- Medium effect = 0.5
- Large effect = 0.8

The results and interpretation are reported in Tables 11 and 13.

TABLE 14. The total number of classes for each pattern in each group.

Pattern	*ALL*	*DEF*	*SDP*	*nSDP*	def-*SDP*	def-*nSDP*
(Object) Adapter	417	232	55	362	39	193
Bridge	18	12	3	15	3	9
Chain of Responsibility	3	3	1	2	1	2
Composite	16	10	1	15	1	9
Decorator	103	65	1	102	1	64
Factory Method	79	36	2	77	1	35
Observer	14	9	2	12	2	7
Prototype	14	10	2	12	2	8
Proxy	29	26	3	26	3	23
Singleton	80	39	7	73	6	33
State	202	116	37	165	30	86
Template Method	60	25	8	52	4	21
Visitor	38	12	5	33	5	7

TABLE 15. Results of the OR test and FET for the specific patterns.

Pattern	OR	$log(OR)$	FET
(Object)Adapter	1.336	0.289	p=0.004
Bridge	2.086	0.735	p=0.159
Chain of Responsibility	∞	∞	p=0.117
Composite	1.737	0.552	p=0.323
Decorator	1.800	0.587	p=0.003
Factory Method	0.883	-0.124	p=0.571
Observer	1.897	0.640	p=0.292
Prototype	2.607	0.958	p=0.111
Proxy	9.102	2.208	p<0.001
Singleton	0.989	-0.011	p=1.000
State	1.423	0.352	p=0.014
Template Method	0.740	-0.301	p=0.298
Visitor	0.477	-0.740	p=0.034

C. RQ3. WHAT IS THE EFFECT OF CODE SMELLS ON THE RELATIONSHIP BETWEEN SPECIFIC DESIGN PATTERNS AND DEFECTS?

1) THE BINARY RELATIONSHIP

In this section we investigate the binary relationship between individual patterns and defects and how the presence of smells impacts this relationship. As the number of specific patterns is too low in each system, we merged the datasets. In Table 14 we present the descriptive data on the resultant dataset.

Next, we performed again the OR test for each pattern and applied FET to measure the significance of the results (see Table 15).

From the results we can conclude that the classes involved in Adapter, Decorator, Proxy and State are more defect-prone than non-pattern classes. The Visitor pattern appears to be associated with the absence of defects, but due to a large p-value we consider the result to be uncertain.

We followed these steps by measuring the effect of smells on these extracted associations and for that we repeated the same tests taking into consideration the effect of smells (see Table 16). The results suggest that the positive association between Adapter, State, and Visitor patterns with defects exists only if they are affected by smells.

2) THE DISTRIBUTION OF DEFECTS

In this section we investigate whether specific patterns attract more or fewer defects than non-pattern classes, and how the introduction of smells affects this distribution.

In Table 17, we present the results of the WMW test that compare the distribution of defects for specific patterns with the distribution of defects in non-pattern classes.

According to the results, the Adapter, Decorator, Prototype, Proxy and State patterns are more defect-prone than non-pattern classes, while the results for the other patterns are inconclusive. We also performed a Hedges' g test and the results show that the extracted rules have a different significance depending on the pattern type. They report that none of the extracted rules has a large effect size and that the significance varies between medium, small or negligible, depending on the type of the pattern.

To investigate the effect of smells on the previous rules, we performed a similar analysis for the classes with code smells. In Table 18 we present the results of the WMW test that compares the defects distribution in the smelly and non-smelly classes involved in patterns, with a distribution of defects in non-pattern classes. Table 19 reports the extracted rules from the WMW test and the significance of those extracted rules based on yhe results of the Hedges' g test.

V. DISCUSSION

A. RQ1: WHAT IS THE IMPACT OF CODE SMELLS ON THE PRESENCE/ABSENCE OF DEFECTS IN CLASSES INVOLVED IN DESIGN PATTERNS?

If we consider the binary relationship between patterns and defects (i.e., defective and defect-free classes) the results show that in five of the analyzed systems (Ant-1.7, JEdit-4.2, Camel-1.6, Ivy-2.0 and Velocity-1.6), the presence of patterns is positively associated with the presence of defects. For the other five systems we could not find any significant rules. It is also important to point that no rule that contradicts the extracted rules was identified, so in no system could we relate patterns to the absence of defects. After the introduction of the effect of smells to the analysis, we found that three out of five systems that were found to have a positive relationship between patterns and defects exhibit this relationship only if the patterns are affected by smells (JEdit-4.2, Camel-1.6, Ivy-2.0 and Velocity-1.6), and for the other two (Ant-1.7 and Camel-1.6) the relationship exists regardless of the presence of smells in the patterns.

In the remaining five systems, for which we initially could not reject the null hypothesis, patterns in Poi-3.0 were found to be positively related with defects only if they were smelly; in Xerces-2.0 the effect was even more evident: smelly patterns have been positively associated with defects, while the association for non-smelly patterns was negative.

By including code smells we did not only extract more rules, but we also found that in the majority of systems, patterns are positively associated with defects only when they are smelly. On the other hand, the results for non-smelly patterns are mixed, so while they are positively associated with defects in a couple of systems, they also have negative or no relationship with defects in other systems. This observation provides a possible explanation for the mixed [52] or small

TABLE 16. Results of OR test and FET for specific patterns, considering the effect of code smells.

Pattern	*SDP*			*nSDP*		
	OR	$Log(OR)$	FET	OR	$Log(OR)$	FET
(Object)Adapter	2.634	0.968	$p = 0.001 < 0.05$	1.204	0.185	$p = 0.091$
Decorator	∞	∞	$p = 0.490$	1.772	0.572	$p = 0.006 < 0.05$
Proxy	∞	∞	$p = 0.117$	8.042	2.084	$p = 0 < 0.05$
State	4.501	1.504	$p = 0.0001 < 0.05$	1.137	0.128	$p = 0.429$
Visitor	∞	∞	$p = 0.028 < 0.05$	0.278	-1.280	$p = 0.001 < 0.05$

TABLE 17. Results of the WMW test for specific patterns, together with Hedges 'g results.

Pattern	*DP* vs. *nDP*	Conclusion	Hedges' g
(Object)Adapter	$z = -5.284, p < 0.001$	$DP > nDP$	0.574 (medium)
Bridge	$z = -1.598, p = 0.109$	H0 cannot be rejected	
Chain Of Responsibility	$z = -1.758, p = 0.078$	H0 cannot be rejected	
Composite	$z = -1.839, p = 0.065$	H0 cannot be rejected	
Decorator	$z = -2.907, p = 0.003$	$DP > nDP$	0.146 (negligible)
Factory Method	$z = -0.308, p = 0.757$	H0 cannot be rejected	
Observer	$z = -1.925, p = 0.054$	H0 cannot be rejected	
Prototype	$z = -2.331, p = 0.019$	$DP > nDP$	0.351 (small)
Proxy	$z = -4.761, p < 0.001$	$DP > nDP$	0.517 (medium)
Singleton	$z = -1.921, p = 0.054$	H0 cannot be rejected	
State	$z = -4.385, p < 0.001$	$DP > nDP$	0.629 (medium)
Template Method	$z = 0.117, p = 0.906$	H0 cannot be rejected	
Visitor	$z = 1.857, p = 0.063$	H0 cannot be rejected	

TABLE 18. Results of WMW test for specific patterns, considering the effect of code smells.

Pattern	*SDP* vs. *nSDP*	*SDP* vs. *nDP*	*nSDP* vs. *nDP*
(Object) Adapter	$z = -3.35, p < 0.001$	$z = -5.323, p < 0.001$	$z = -3.716, p < 0.001$
Bridge	$z = -2.594, p = 0.009$	$z = -2.991, p = 0.002$	$z = -0.416, p = 0.677$
Chain Of Responsibility	insufficient data	insufficient data	$z = -1.702, p = 0.088$
Composite	insufficient data	insufficient data	$z = -1.468, p = 0.141$
Decorator	insufficient data	insufficient data	$z = -2.786, p = 0.005$
Factory Method	$z = 0.171, p = 0.863$	$z = 0.182, p = 0.855$	$z = -0.341, p = 0.732$
Observer	$z = -2.167, p = 0.030$	$z = -2.657, p = 0.007$	$z = -0.996, p = 0.318$
Prototype	$z = -1.613, p = 0.106$	$z = -2.235, p = 0.025$	$z = -1.607p = 0.107$
Proxy	$z = -1.779, p = 0.075$	$z = -2.658, p = 0.007$	$z = -4.131, p < 0.001$
Singleton	$z = -2.183, p = 0.028$	$z = -2.858, p = 0.004$	$z = -1.135, p = 0.256$
State	$z = -3.987, p < 0.001$	$z = -5.656, p < 0.001$	$z = -2.199, p - 0.027$
Template Method	$z = -0.536, p = 0.591$	$z = -0.566, p = 0.570$	$z = 0.347, p = 0.728$
Visitor	$z = -3.269, p = 0.001$	$z = -1.987, p = 0.046$	$z = 2.767, p = 0.005$

relationship [39] reported in the literature between design patterns and defect-proneness. The presence of code smells appears to be a factor that interacts with design patterns and has a decisive impact on defects in the subject code by amplifying the previously existing defect-proneness. Consequently, it has a practical consequence for software developers. The intense use of patterns can lead to their interactions and the proliferation of cross-cutting effects [6], resulting in some types of code smells. That, in turn, could effectively diminish or revert the expected advantages of applying design patterns, even if the pattern classes attract fewer smells than the non-pattern ones [53].

B. RQ2: WHAT IS THE IMPACT OF CODE SMELLS ON THE DEFECT DISTRIBUTION (NUMBER OF DEFECTS) IN CLASSES INVOLVED IN DESIGN PATTERNS?

The results for nine out of ten of the analyzed systems indicate that design pattern classes are linked with a higher number of defects than the non-pattern classes. Only in the case of Xalan-2.7 no significant rules were identified. The effect size analysis reported that the mean difference between the smelly and non-smelly patterns in terms of defects is between [0.2-0.5] of standard deviation, which entails that the significance of those extracted rules are either small or medium, depending on the system.

By introducing information about code smells into the analysis, we obtained new insights into those results. First, in the majority of systems (seven out of ten), smelly patterns have a higher number of defects than the non-smelly patterns, and no system produced contradictory results. The effect size for the majority of those extracted rules is large, indicating that the difference between the two groups is of a large significance.

With regard to the effect of smells on the relationship between pattern vs. non-pattern classes with defects, the extracted rules were difficult to interpret, because while smelly patterns are associated with more defects than non-pattern classes in eight systems, the non-smelly patterns also have more defects than non-pattern classes in six systems. Those results initially suggested that the smelliness

TABLE 19. The conclusions from the WMW test results presented in Table 18, together with the Hedges' g effect size test results.

Pattern	*SDP* vs. *nSDP*	*SDP* vs. *nDP*	*nSDP* vs. *nDP*
(Object) Adapter	*SDP* > *nSDP*. H'g=0.429(small)	*SDP* > *nDP*. H'g= 1.730(large)	*nSDP* > *nDP*. H'g=0.475(small)
Bridge	*SDP* > *nSDP*. H'g= 2.120(large)	*SDP* > *nDP*. H'g= 9.72(large)	not significant
Chain Of Resp	not significant	not significant	not significant
Composite	not significant	not significant	not significant
Decorator	not significant	not significant	*nSDP* > *nDP*. H'g=0.14(negligible)
Factory Method	not significant	not significant	not significant
Observer	*SDP* > *nSDP*. H'g=4.294(large)	*SDP* > *nDP*. H'g=5.163(large)	not significant
Prototype	not significant	*SDP* > *nDP*. H'g=1.349(large)	not significant
Proxy	not significant	*SDP* > *nDP*. H'g=1.349(large)	*nSDP* > *nDP*. H'g=0.421(small)
Singleton	*SDP* > *nSDP*. H'g=0.6(medium)	*SDP* > *nDP*. H'g=1.708(large)	not significant
State	*SDP* > *nSDP*. H'g=0.813(large)	*SDP* > *nDP*. H'g= 2.135(large)	*nSDP* > *nDP*. H'g=0.334(small)
Template Method	not significant	not significant	not significant
Visitor	*SDP* > *nSDP*. H'g=1.476(large)	*SDP* > *nDP*. H'g=1.469(large)	*nSDP* < *nDP*.H'g=-0.308(small)

TABLE 20. The extracted rules from the WMW test, together with the effect size interpretation of the Hedges' g test.

System	*SDP* vs. *SnDP*	*SDP* vs. *nSnDP*	*nSDP* vs. *SnDP*	*nSDP* vs. *nSnDP*
Ant	not significant	$SDP > nSnDP$(large)	not significant	$nSDP > nSnDP$(small)
JEdit	not significant	$SDP > nSnDP$(large)	not significant	not significant
Lucene	not significant	not significant	not significant	$nSDP > nSnDP$(small)
Camel	not significant	not significant	$nSDP > SnDP$(small)	$nSDP > nSnDP$(medium)
Log4j	not significant	not significant	not significant	$nSDP > nSnDP$(small)
Xalan	$SDP > SnDP$(small)	$SDP > nSnDP$(large)	$SnDP > nSDP$(small)	not significant
Poi	$SDP > SnDP$(large)	$SDP > nSnDP$(large)	not significant	not significant
Ivy	$SDP > SnDP$(large)	$SDP > nSnDP$(large)	not significant	not significant
Xerces	$SDP > SnDP$(large)	$SDP > nSnDP$(large)	$SnDP > nSDP$(negligible)	not significant
Velocity	$SDP > SnDP$(large)	$SDP > nSnDP$(large)	not significant	not significant

of a pattern is not a valid contextual factor for analyzing defect-proneness. A thorough analysis of the extracted rules shows that the extracted rules for the relationship between smelly patterns and defects are stronger than those which show the relationship between non-smelly patterns and defects. All the rules extracted for the smelly patterns are significant at $\alpha = 0.01$, while only two rules are significant at the same level for the non-smelly patterns. The effect size analysis also strengthens this conclusion, since for the smelly pattern rules the mean difference is large enough to be of a practical significance, while the effect size for the non-smelly pattern rules is either small or even negligible.

To have a more comprehensive understanding of the results and to isolate the effect of smells, we again performed a WMW test to compare the smelly vs. non-smelly patterns with smelly vs. non-smelly non-pattern classes. The results are summarized in Table 20.

The results confirm our initial observations that smelly patterns, in the majority of systems, have a higher defect distribution than the non-pattern classes, regardless of whether those classes are smelly or not. On the other hand, the comparison of non-smelly patterns with smelly and non-pattern classes showed that they have similar defect distribution in the majority of the systems and that non-smelly patterns have a higher defect distribution only in case of Camel, while they have a lower number of defects in case of Xalan and Xerces. This may suggest that the effect of smells and patterns on defects is cumulative and the results of comparing data that belong to only one group (smells or patterns) could be attributed to other contextual factors. It is also worth mentioning that the effect size analysis of the relationship between *nSDPvs.SnDP* shows that the significance of the mean difference between those two groups is small or even negligible. The results also suggest that classes which are not participating in a pattern and are not affected by smells tend to attract fewer defects than pattern classes.

Finally, the results of the effect size analysis strengthen our conclusion, as they show the large significance of the extracted rules related to smelly design patterns, while they demonstrate the small significance of the rules related to the non-smelly patterns.

C. RQ3: WHAT IS THE EFFECT OF CODE SMELLS ON THE RELATIONSHIP BETWEEN SPECIFIC DESIGN PATTERNS AND DEFECTS?

In the subsequent sections we discuss the binary and the cumulative relationships between specific patterns and defects, and the effect of smells on these relationships.

1) THE BINARY RELATIONSHIP

We are interested in analyzing the binary relationship between specific patterns and defects, and in describing how the presence of smells affects this relationship. However, the dataset has a very small number of instances for some patterns, e.g., Chain of Responsibility (3 instances, all of them are smelly), Bridge (18 instances, only three of them are smelly), Observer and Prototype (14 instances each, only 2 instances in each case are smelly). Because of the small sample size, FET reported the insignificance of the extracted rules even if they were supported in 100% of cases. Nevertheless, the detailed analysis reported some significant associations such as the Adapter, Decorator, Proxy and State patterns are positively related with the presence of defects, while the presence of the Visitor pattern is associated with the absence

of defects. The Visitor case contradicts our findings reported in Sec. IV-A and contradicts our findings for all other patterns in this section.

The case of Visitor is unique: among 38 instances, 12 of them are defective and 26 are defect-free. All of them come from a single system, velocity-1.6, and are located in a single package `org.apache.velocity.runtime.parser.node`. All instances represent objects that visit and parse a specific type of a node. As the amount of the code inside the Visitor pattern is minimal, and its logic is clear and simple, no defect was reported for the majority of those instances. In the remaining classes, defects were cosmetic or related to special cases which have not been covered.

After introducing the effect of smells to our analysis, the results showed that in the case of Adapter, State and Visitor, smelly patterns are positively associated with the presence of defects. However, the Decorator and Proxy patterns require the investigation to be replicated on a larger dataset: although the presence of smells in the pattern classes was associated with defects in 100% of the cases, the small number of smelly classes (1 for Decorator and 3 for Proxy) invalidated the FET results. Furthermore, we found that the non-smelly Visitor classes are associated with the absence of smells. These results are consistent with our findings reported in Sec. IV-A.

On the other hand, for both the Decorator and Proxy patterns, the results suggest that smell-free pattern classes are also associated with the presence of defects, which compels us to conduct further investigation. For the other patterns, the small number of detected instances prevented us from extracting any rules.

For the Decorator pattern, the majority of defective non-smelly instances belong to two systems, Xalan-2.7 and Camel-1.6. In both systems the evolution of the Decorator pattern scattered its functionalities into many small objects representing crosscutting concerns, which in turn became hard to comprehend and maintain, and as a consequence, produced defects.

With regards to the Proxy pattern, there are 23 defective non-smelly instances, and 16 of them belong to Xalan-2.7. Almost all the Visitor instances in this system belong to a single package `org.apache.xalan.xsltc.compiler`. Those instances parse specific types of instructions before passing the parsed segments to a converter object. Those instances extend a single parent class, `Instruction`, and their tight coupling with this shared parent causes them to also share the same defects.

2) THE DISTRIBUTION OF DEFECTS

The results of the detailed analysis are consistent with our findings reported in IV-B. They indicate that the Adapter, Decorator, Prototype, Proxy and State patterns are linked with more defects than non-pattern classes. While we could not extract any rules for other patterns, no extracted rules in any of the patterns contradicted our findings. The introduction of smells into our analysis resulted in the observation that, in case of the Adapter, Bridge, Observer, Singleton, State and Visitor patterns, smelly patterns attract more defects than non-smelly patterns. For all other patterns, we could not extract any rules that contradict our findings.

With respect to the effect of smells on the relationship between patterns and defects we found that for the majority of pattern types (Adapter, Bridge, Observer, Prototype, Proxy, Singleton, State and Visitor), the smelly design patterns attracted more defects than non-pattern classes. The effect size analysis reported that the mean difference between those smelly patterns and non-pattern classes is greater than the 0.8 of standard deviation, which indicates the large significance of those extracted rules. For the non-smelly patterns, we found that only in the case of Adapter, Decorator, Proxy and State did the non-smelly patterns attract more smells than the non-pattern classes and those extracted rules have a small significance. Moreover, in case of the Visitor pattern, we concluded that non-smelly patterns have fewer defects than non-pattern classes.

The defect proneness of specific smells and patterns has also been studied in the literature. Our findings partially confirm the results reported by Aversano *et al.* [6], Vokáč [52] and Onarcan and Fu [39] with respect to the Singleton and Observer patterns and their positive association with defects. Our results show that for both of them the presence of smells additionally amplified the defect proneness of the affected classes. For other patterns, the results differ, which may also indicate the confounding role of code smells addressed in our work.

VI. THREATS TO VALIDITY

A. CONSTRUCT VALIDITY

Due to limitations related to the granularity of the PROMISE dataset, our analysis had to be performed at the class level. This inevitably affected the results, as some code smells have been reassigned from methods to classes.

The other issue concerns the detection tools used to detect both patterns and smells. Although those tools were chosen for their high precision, recall and accuracy compared to other detection tools [3], [16], we have not performed cross-validation with other tools, just manual verification for a sample of detected patterns. Our results are as accurate as those tools are.

Finally, in our analysis we consider the presence of smells as a binary variable, regardless of their types and number. Research on relationships among various code smells shows that collocated smells seem to have more detrimental impact on quality than individual smells [5]. This issue could be further elaborated in replication studies.

B. EXTERNAL VALIDITY

Our exploratory analysis was performed on 10 small- and medium-size Java systems, and our findings should be interpreted accordingly. Furthermore, for some patterns we have

only a small number of detected instances, which limits extrapolating the results beyond the scope of the study.

C. CONCLUSION VALIDITY

Most of our findings were driven and validated statistically, so our conclusion is highly related with the accuracy of the statistical tests which we used. The non-normal distribution dictated that we use non-parametric tests which have a lower statistical power than parametric tests. Additionally, the size of the subject projects makes us cautious when evaluating the impact of the results.

VII. CONCLUSION

In the paper we investigated the links between design patterns and defects, and how the presence/absence of smells affects these relationships. Our analysis included 10 small- and medium-size Java systems. The findings suggest that pattern classes are associated with more defects than non-pattern classes, and that smells could be considered as a contextual factor in this relationship since smelly pattern classes attract more defects than both non-smelly pattern and non-pattern classes.

The paper findings are three-fold:

- Investigating the binary relationship between patterns and defects showed that patterns are positively associated with the presence of defects, thus validating the results reported in previous studies. However, by including the presence of smells as a confounding variable in this relationship, our results indicate that only smelly patterns have a unanimously positive association with defects, while non-smelly patterns delivered mixed results.
- Our results show that pattern classes have a greater number of defects than non-pattern classes. Introducing the effect of smells into the analysis reveals that smelly classes attract more defects than non-smelly classes, and that both smelly- and non-smelly pattern classes have, in a different rate, a higher defect distribution than non-pattern classes. The findings also suggest that the relationship between smelly patterns and defects is more significant than the relationship between non-smelly patterns and defects.
- The relationship between specific patterns and defects varies, both in terms of the binary and quantitative relationships. This variation still holds true if smells are introduced into this relationship.

  Nevertheless, there are some common findings between all the patterns. For example, our analysis did not reveal a pattern that attracts a lower number of defects than non-pattern classes. In contrast, the Adapter, Decorator, Prototype, Proxy and State patterns tend to have a higher defect distribution than non-pattern classes. The introduction of smells into the analysis showed that the majority of smelly pattern classes attract more defects than non-pattern classes, and that non-smelly patterns attract more or fewer defects, depending on their type. For example, the non-smelly Adapter, Decorator, Proxy and State classes attract more defects than the non-pattern classes. On the other hand, smelly Visitor classes are linked with a lower defect distribution than non-pattern classes.

  What is also noticeable in our results is that no non-smelly pattern attracts a higher defect number than a smelly pattern. On the contrary, smelly Adapter, Bridge, Observer, Singleton, State and Visitor classes tend to have more defects than non-smelly pattern classes.

  Furthermore, the binary association between different patterns and defects also varies between patterns. The Adapter, Decorator, Proxy and State patterns are associated with the presence of defects, while the Visitor pattern is associated with the absence of defects. Taking into consideration the effect of smells on the previous findings showed that the majority of smelly patterns tend to have positive associations with defects, but with a different confidence and significance.

The results, albeit preliminary, can inspire and foster further research on the contextual factors that affect defect-proneness, changeability and other important software properties. Understanding their role may help in isolating their individual impact and the interactions they play a role in.

Our findings can have an impact on the development of practice. Design patterns promote good practices. However, if pattern classes are affected by code smells, the advantages of patterns could be challenged by defects resulting from their interaction with smells. Therefore, we conclude that preventing and removing code smells may reduce the defect-proneness of the code, so we advise programmers to take this possibility into account.

In the future, we plan to replicate this experiment on a larger scale and to investigate the individual effect of specific smells on the relationship between patterns and defects.

REFERENCES

[1] J. S. Shirabad and T. J. Menzies, "The PROMISE repository of software engineering databases," School Inf. Technol. Eng., Univ. Ottawa, Ottawa, ON, Canada, Tech. Rep., 2005. [Online]. Available: http://promise.site.uottawa.ca/SERepository

[2] P. Afric, L. Sikic, A. S. Kurdija, G. Delac, and M. Silic, "REPD: Source code defect prediction as anomaly detection," in *Proc. IEEE 19th Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Jul. 2019, pp. 227–234.

[3] M. Al-Obeidallah, M. Petridis, and S. Kapetanakis, "A survey on design pattern detection approaches," *Int. J. Softw. Eng.*, vol. 7, pp. 41–59, Dec. 2016.

[4] M. Alfadel, K. Aljasser, and M. Alshayeb, "Empirical study of the relationship between design patterns and code smells," *PLoS ONE*, vol. 15, no. 4, Apr. 2020, Art. no. e0231731.

[5] V. Ferme, A. Marino, F. A. Fontana, "Is it a real code smell to be removed or not," in *Proc. Int. Workshop Refactoring Test. (RefTest), Co-Located Event XP Conf.* SN, 2013.

[6] L. Aversano, L. Cerulo, and M. D. Penta, "Relationship between design patterns defects and crosscutting concern scattering degree: An empirical study," *Software, IET*, vol. 3, pp. 395–409, Nov. 2009.

[7] D. Bán and R. Ferenc, "Recognizing antipatterns and analyzing their effects on software maintainability," in *Proc. Int. Conf. Comput. Sci. Appl.*, Jun. 2014, pp. 337–352.

[8] A. Cairo, G. Carneiro, and M. Monteiro, "The impact of code smells on software bugs: A systematic literature review," *Information*, vol. 9, no. 11, p. 273, Nov. 2018.

[9] B. Cardoso and E. Figueiredo, "Co-occurrence of design patterns and bad smells in software systems: An exploratory study," in *Proc. Anais do Simpósio Brasileiro de Sistemas de Informação (SBSI)*, May 2015, pp. 347–354.
[10] J. Cohen, "Statistical power ANALYSIS for the behavioral sciences," *SERBIULA, Sistema Librum 2.0*, vol. 2nd, Jan. 1988, doi: 10.1016/B978-0-12-179060-8.50012-8.
[11] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Proc. 10th Int. Conf. Qual. Softw.*, Jul. 2010, pp. 23–31.
[12] J. Dong, D. S. Lad, and Y. Zhao, "DP-miner: Design pattern discovery using matrix," in *Proc. 14th Annu. IEEE Int. Conf. Workshops Eng. Comput.-Based Syst. (ECBS)*, Mar. 2007, pp. 371–380.
[13] M. O. Elish and M. A. Mohammed, "Quantitative analysis of fault density in design patterns: An empirical study," *Inf. Softw. Technol.*, vol. 66, pp. 58–72, Oct. 2015.
[14] E. Erturk and E. Akcapinar Sezer, "Iterative software fault prediction with a hybrid approach," *Appl. Soft Comput.*, vol. 49, pp. 1020–1033, Dec. 2016.
[15] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy, "A public unified bug dataset for java," in *Proc. 14th Int. Conf. Predictive Models Data Anal. Softw. Eng.*, Oct. 2018, pp. 12–21.
[16] F. Arcelli Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment.," *J. Object Technol.*, vol. 11, no. 2, p. 5:1, 2012.
[17] M. Fowler, *Refactoring: Improving Design Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.
[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Reading, MA, USA: Addison-Wesley, 1994.
[19] M. Gatrell and S. Counsell, "Design patterns and fault-proneness a study of commercial C# software," in *Proc. 5th Int. Conf. Res. Challenges Inf. Sci.*, May 2011, pp. 1–8.
[20] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*. Upper Saddle River, NJ, USA: Prentice-Hall, 1992.
[21] Y.-G. Gueheneuc and H. Albin-Amiot, "Using design patterns and constraints to automate the detection and correction of inter-class design defects," in *Proc. 39th Int. Conf. Exhib. Technol. Object-Oriented Lang. Syst. TOOLS*, 2001, pp. 296–305.
[22] A. Gupta, B. Suri, and S. Misra, "A systematic literature review: Code bad smells in java source code," in *Proc. Int. Conf. Comput. Sci. Appl.*, Jul. 2017, pp. 665–682.
[23] 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.*, vol. 23, no. 4, pp. 1–39, Sep. 2014.
[24] L. V. Hedges, "Estimation of effect size from a series of independent experiments.," *Psychol. Bull.*, vol. 92, no. 2, pp. 490–499, 1982.
[25] F. Jaafar, A. Lozano, Y.-G. Gueheneuc, and K. Mens, "On the analysis of co-occurrence of anti-patterns and clones," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2017, pp. 274–284.
[26] C. C. Jones, *Software Quality: Analysis and Guidelines for Success*, 1st ed. Boston, MA, USA: Thomson Learning, 1997.
[27] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, vol. 9, Dec. 2010, p. 9.
[28] F. Khomh and Y.-G. Gueheneuc, "Do design patterns impact software quality positively?" in *Proc. 12th Eur. Conf. Softw. Maintenance Reeng.*, Apr. 2008, pp. 274–278.
[29] 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, 2007.
[30] 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.*, vol. 80, no. 7, pp. 1120–1128, Jul. 2007.
[31] H. Liu, B. Li, Y. Yang, W. Ma, and R. Jia, "Exploring the impact of code smells on fine-grained structural change-proneness," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 28, no. 10, pp. 1487–1516, Oct. 2018.
[32] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, Mar. 1947.
[33] M. V. Mantyla, "An experiment on subjective evolvability evaluation of object-oriented software: Explaining factors and interrater agreement," in *Proc. Int. Symp. Empirical Softw. Eng.*, Nov. 2005, pp. 277–286.
[34] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. 20th IEEE Int. Conf. Softw. Maintenance*, Sep. 2004, pp. 350–359.
[35] R. Nascimento and C. Sant'Anna, "Investigating the relationship between bad smells and bugs in software systems," in *Proc. 11th Brazilian Symp. Softw. Compon., Archit., Reuse SBCARS*, 2017, pp. 1–10.
[36] J. Niere, W. Schafer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," in *Proc. 24th Int. Conf. Softw. Eng. ICSE*, 2002, pp. 338–348.
[37] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–10.
[38] S. Olbrich, S. D. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proc. 3rd Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Oct. 2009, pp. 390–400.
[39] M. O. Onarcan and Y. Fu, "A case study on design patterns and software defects in open source software," *J. Softw. Eng. Appl.*, vol. 11, no. 05, pp. 249–273, 2018.
[40] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. Verkamo, "Software metrics by architectural pattern mining," in *Proc. Int. Conf. Softw., Theory Pract. (16th IFIP World Comput. Congr.)*, Jan. 2000.
[41] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," *Empirical Softw. Eng.*, vol. 23, no. 3, pp. 1188–1221, Aug. 2017.
[42] L. Prechelt, B. Unger, W. F. Tichy, P. Brossler, and L. G. Votta, "A controlled experiment in maintenance: Comparing design patterns to simpler solutions," *IEEE Trans. Softw. Eng.*, vol. 27, no. 12, pp. 1134–1144, Dec. 2001.
[43] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy, "Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance," *IEEE Trans. Softw. Eng.*, vol. 28, no. 6, pp. 595–606, Jun. 2002.
[44] G. Shan and S. Gerstenberger, "Fisher'exact approach for post hoc analysis of a chi-squared test," *PLoS ONE*, vol. 12, no. 12, pp. 1–12, Dec. 2017.
[45] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, nos. 3–4, pp. 591–611, Dec. 1965.
[46] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 4th ed. London, U.K.: Chapman & Hall/CRC, 2007.
[47] D. I. K. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.
[48] B. Sousa, M. Bigonha, and A. M. K. Ferreira, "A systematic literature mapping on the relationship between design patterns and bad smells," in *Proc. 33rd Annual ACM Symp. Appl. Comput.*, Apr. 2018, pp. 1528–1535.
[49] B. Sousa, M. Bigonha, and K. Ferreira, "An exploratory study on cooccurrence of design patterns and bad smells using software metrics," *Softw., Pract. Exper.*, May 2019, doi: 10.1002/spe.2697.
[50] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, Nov. 2006.
[51] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 403–414.
[52] M. Vokac, "Defect frequency and design patterns: An empirical study of industrial code," *IEEE Trans. Softw. Eng.*, vol. 30, no. 12, pp. 904–917, Dec. 2004.
[53] B. Walter and T. Alkhaeir, "The relationship between design patterns and code smells: An exploratory study," *Inf. Softw. Technol.*, vol. 74, pp. 127–142, Jun. 2016.
[54] P. Wendorff, "Assessment of design patterns during software reengineering: Lessons learned from a large commercial project," in *Proc. 5th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2001, pp. 77–84.
[55] A. Yamashita, "Assessing the capability of code smells to explain maintenance problems: An empirical study combining quantitative and qualitative data," *Empirical Softw. Eng.*, vol. 19, no. 4, pp. 1111–1143, Aug. 2014.
[56] D. Yu, Y. Zhang, and Z. Chen, "A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures," *J. Syst. Softw.*, vol. 103, pp. 1–16, May 2015.
[57] X. Zhang, Y. Zhou, and C. Zhu, "An empirical study of the impact of bad designs on defect proneness," in *Proc. Int. Conf. Softw. Anal., Test. Evol. (SATE)*, Nov. 2017, pp. 1–9.

TAREK ALKHAEIR received the master's degree in computing from the Poznań University of Technology, Poznań, Poland, and the master's degree in web science from Syrian Virtual University, Syria. He is currently pursuing the Ph.D. degree with the Poznań University of Technology. He is currently a Software Engineer and a Researcher. He also works as a Senior Software Engineer with Roche Poland. His research interests include code quality, design patterns, and code smells.

BARTOSZ WALTER received the Ph.D. degree from the Poznań University of Technology, Poland. He is currently affiliated to his Alma Mater, where he is also an Assistant Professor, and also works as a Senior Researcher with the Poznań Supercomputing and Networking Center, Poznań, Poland. His research interests include software maintenance and evolution, as well as several aspects of object-oriented design and software testing.