The Yin and Yang of Software Quality: On the Relationship between Design Patterns and Code Smells

Giammaria Giordano, Giulia Sellitto, Aurelio Sepe, Fabio Palomba, Filomena Ferrucci
Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy
giagiordano@unisa.it, gisellitto@unisa.it, a.sepe21@studenti.unisa.it, fpalomba@unisa.it, fferrucci@unisa.it

***Abstract*—Software reuse is considered the silver bullet of software engineering. It has been largely demonstrated that the proper implementation of design and reuse principles can substantially reduce the effort, time, and costs required to develop software systems. Design patterns are one of the most affirmed techniques for source code reuse. While previous work pointed out their benefits in terms of maintainability and understandability, some seem to raise the opposite concern, suggesting that they can negatively impact code quality from the developers' perspectives. We recognize such discrepancy in the literature, and we aim to fill this gap by investigating whether and how design patterns are related to the emergence of issues compromising code understandability, namely the *Complex Class*, *God Class*, and *Spaghetti Code* smells, which have been also shown to increase the change- and fault-proneness of code. We perform an empirical evaluation on 15 JAVA projects evolving over 542 releases, and we find that, although design patterns are supposed to improve code quality without prejudice, they can be related to dangerous issues, as we observe the emergence of code smells in the classes participating in their implementation. From our findings, we distill a number of implications for developers and project managers to support them in dealing with design patterns.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance Effort; Empirical Software Engineering.**

I. INTRODUCTION

Software reusability is considered the silver bullet of Software Engineering. The term refers to reusing available source code or already tested solutions to solve a similar problem when implementing new features or refactoring the existing ones [6]. The proper application of reusability practices guarantees developers to reduce time, effort, and costs of the maintenance tasks [21], [30]. Most programming languages, especially the ones implementing the Object-Oriented paradigm, provide a wide range of mechanisms to support developers' in applying encouraged best practices of software reuse, *i.e.*, leveraging third-party libraries, implementing program abstractions, and introducing design patterns [12].

The idea of design patterns was proposed in 1995 by the *Gang of Four*, who defined them as reusable solutions to commonly occurring problems that arise during the design and development of software applications [11]. Adopting such reusability mechanisms can provide several advantages from the developers' perspectives, as their flexibility makes them re-appliable by changing the context, the environment, and the programming languages, without changing the philosophy driving a given pattern [37]. The large spread of Object Oriented programming languages boosted developers to reuse instance classes and to create hierarchies that can be easily used as a basis for the introduction of design patterns. Previous studies investigated the use of design patterns in JAVA [14], [7], [29], mainly because (1) JAVA offers, by design, mechanisms and data structures that make large use of reusability principles, especially linked to inheritance, and (2) although the fluctuating trends, JAVA is still one the most adopted programming languages in large companies and open-source communities.[1] While most research emphasize the importance of reusability mechanisms to guarantee high quality of the software, a number of studies seem to go in the opposite direction, highlighting that a sub-optimal implementation of design patterns can, in turn, increase the code complexity and negatively impact the code in terms of maintainability and comprehension [19]. Fowler and Beck identified *code smells* as indicators of the poor quality of code, affecting its cohesion, coupling, and comprehensibility, ultimately making the code difficult to maintain [10].

In this paper, we investigate the role that design patterns play in the presence of code smells. We analyze 15 open-source JAVA projects spanning over 542 releases, by extracting information about the implemented design patterns and the code smells affecting the classes, and assessing (1) the co-occurrences of design patterns and code smells, and (2) whether the presence of design patterns is correlated with the formation of code smells. We find that, although design patterns are intended to improve the quality of the code, as they represent a reuse mechanism, there is no guarantee on them enhancing the goodness of the software; on the contrary, design patterns can in fact determine the appearance of code smells in certain cases. We point out the importance of carefully dealing with design patterns by applying them properly and monitoring their evolution in the software projects. Our main points of contribution can be summarized as follows:

1) An empirical investigation of design patterns and their impact on code smells, that can enhance the state of the art on software reusability and, at the same time, can aid practitioners in monitoring the changes in complexity and comprehension when implementing design patterns;

[1] Source: https://www.tiobe.com/tiobe-index/

2) A publicly available online appendix containing all the scripts, raw data, and additional materials used to perform our experiments, that can be leveraged for replication and verification of our work [13].

In the following, we first report the state of the art on design patterns and code smells, highlighting how our work is positioned in the current body of knowledge. Then, we present the design of our study, discussing the rationale driving our goal, research questions, and methodological choices. Afterwards, we comment the obtained results and draw out the practical implications of our findings. Finally, we consider the threats to the validity of our work, and we conclude the paper by distilling an exhaustive summary.

II. Related Work

In the context of our work, we summarize the state of the art focused on the benefits and drawbacks of design patterns and the current literature on code smells.

Design patterns have been introduced by the *Gang of Four* in 1995 [11] and, since then, have been praised as the *Holy Grail* of software reusability. They consist in ready-to-apply solutions to recurring problems in software development, and can aid developers in the design and implementation of source code adhering to good cohesion and coupling principles of Object Oriented programming.

Previous work has argued that design patterns may be beneficial for the overall quality of the code. Hegedűs *et al.* [17] investigated the connection between design patterns and software maintainability by performing an empirical study on more than 300 revisions of JHotDraw, a well-known Java framework. They estimated the level of maintainability of source code in terms of different quality attributes, such as number of classes, lines of code, and density of code, and found that each introduction of a design pattern instance generated an improvement in the quality of the project.

However, related work has also discussed that design patterns are not always beneficial for guaranteeing maintainability, especially in terms of understandability and modifiability of the code. Vokáč *et al.* [36] compared the maintainability of programs designed with and without design patterns, by performing a controlled experiment with 44 professionals. They asked participants to execute a number of maintenance tasks on two versions of C++ programs, *i.e.*, one implemented with design patterns and one without. They evaluated the correctness of the executed tasks and the time required by developers, assessing the positive or negative impact of design patterns on software maintainability. They argued that each design pattern has its own nature and proper place of use; they cannot be classified as *good* or *bad* in general terms, but training sessions can improve both the speed and quality of maintenance activities. More recently, Khomh and Guéhéneuc [19] suggested that design patterns may not always have a positive impact on code quality as seen from practitioners' perspective. By performing a survey study with 20 developers, they assessed the perceived impact that design patterns have on the understandability of code. They found that design patterns do not always impact quality attributes positively, as the participants considered that, although they are useful to solve design problems, they often decrease simplicity, learnability, and understandability of the software.

Software quality attributes can be compromised also by anti-patterns, also called *code smells*, introduced in the source code either willingly or by accident [25]. Fowler and Beck defined code smells as symptoms of bad design that can lead to an increase in terms of maintenance effort and defect proneness of source code [10]. Such issues could be introduced in the code due to sub-optimal use of reusability mechanisms, e.g., making overuse of delegation strategies could ultimately lead to a *God Class*, i.e., a class implementing an excessive amount of responsibilities, also wrapping other classes' duties. From an empirical standpoint, several studies have been conducted to understand the relationship between reusability mechanisms and code smells; however, most of them are focused on the concepts of inheritance and delegation [20], [26].

To the best of our knowledge, the only work investigating the connection between design patterns and code smells was conducted by Walter and Alkhaeir in 2015 [37]. They selected two medium-size Java projects and considered 10 design patterns and seven code smells related to maintainability, *e.g.*, *Feature Envy*, occurring when a method calls methods on another class more times than on the source class, and *Message Chains*, consisting in a client requesting another object, which requests yet another one, and so on, navigating the class structure. Their findings revealed that, in some cases, the presence of design pattern was positively correlated with the presence of code smell, *e.g.*, the *Proxy* design pattern sometimes led to the introduction of a *Middle Man* code smell.

Following the path traced by Walter and Alkhaeir [37] and by Khomh and Guéhéneuc [19], we dive deep into the impact that design patterns have on the presence of code smells compromising the understandability and maintainability of software. We apply a research method inspired by the work of Walter and Alkhaeir [37] to assess the findings of Khomh and Guéhéneuc [19] from a quantitative standpoint. We investigate whether design patterns induce those specific kinds of code smells that are perceived by most critics by developers during maintenance activities [27], [32] and are connected with the understandability of code, *i.e.*, (1) *Complex Class*, affecting code having high cyclomatic complexity, (2) *God Class*, implementing several responsibilities, and (3) *Spaghetti Code*, consisting in poorly organized control flow.

III. Study Design

The *goal* of this study was to understand whether and how design patterns are related with code smells. The *context* consisted in 10 design patterns, *i.e.*, *Adapter/Command*, *Bridge*, *Singleton*, *Template Method*, *Proxy*, *State/Strategy*, *Decorator*, *Factory Method*, *Component*, and *Observer*. The *perspective* was of both researchers and practitioners, as the former are interested in increasing the body of knowledge on this topic, and the latter are concerned about understanding how design patterns impact code quality in software systems.

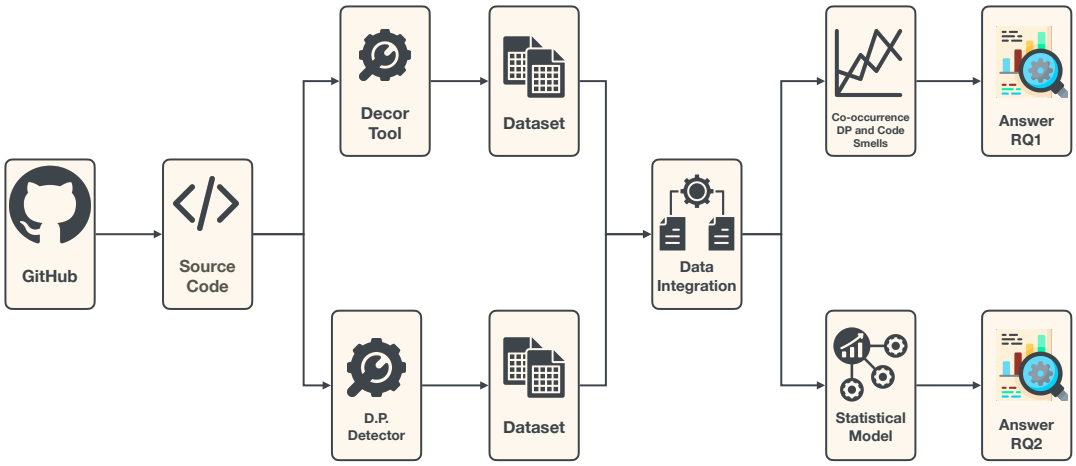

Fig. 1: Overview of the research method applied in this work.

Based on our *goal*, we formulated two research questions.

> **RQ$_1$.** *What are the co-occurrences in terms of classes between design patterns and code smells?*

RQ$_1$ aimed at comprehending the fluctuations in the frequency of classes participating in a particular design pattern, and the co-occurrence of code smells in such classes. We wanted to assess whether classes implementing design patterns contain code smells themselves, and we expected to see a low frequency of smells in classes participating in design patterns.

We were interested in understanding whether and how design patterns are correlated with the presence of code smells, and for this reason, we asked:

> **RQ$_2$.** *To what extent does the presence of design patterns affect code smells?*

To answer our research questions, we performed an empirical study (1) analyzing the co-occurrences of code smells in classes participating in design patterns, and (2) applying statistical models to understand the impact of design patterns on the emerging of code smells.

Figure 1 depicts the method applied in this work, which we designed following the guidelines by Wohlin et al. [38] and the *ACM/SIGSOFT Empirical Standards*[2]; in particular, we leveraged the "General Standard", "Data Science", and "Repository Mining" guidelines. First, we selected 15 JAVA projects from GITHUB and manually built 542 releases. Then, we extracted design patterns implemented in the projects by leveraging the detection tool proposed by Tsantalis *et al.* [35], and we identified code smells affecting the projects by running DECOR [24] on each release. We combined these pieces of information to understand, on the one hand, the co-occurrences of classes that collaborate into design patterns and, simultaneously, are involved in some code smell. On the other hand, we investigated whether design patterns affect code smells from a statistical standpoint.

In the following, we report the detailed design of our work. The complete dataset, scripts, and raw results of our study are available in the online appendix of this paper [13].

A. Dataset Collection

Table I provides an overview of the dataset used in this work. We defined two main criteria for the selection of the projects to consider in our study:

Build Availability. We selected only JAVA projects that can build without errors. This criterion is driven by the constraints dictated by the design pattern detection tool by Tsantalis *et al.* [35], which we selected to obtain data on the design patterns implemented in the projects. The tool requires that target projects build without errors, as it leverages JAVA BYTECODE to generate an intermediate code representation. Thus, it can only be executed on projects that can build successfully. To ensure that, we manually compiled the candidate projects and assessed their compliance with this criterion.

Number of Stars on GITHUB. We only selected projects with a minimum of 2K stars on GITHUB. We set such a threshold to avoid the inclusion of toy projects or personal projects developed by users. The number of stars has been demonstrated to be a good proxy metric to estimate the popularity of repositories and their overall quality [28].

Considering the points above, we manually identified GITHUB projects meeting the criteria. Due to the time-consuming activity, we limited our search to the first 10 pages of GITHUB results filtered to JAVA, finding 45 candidate projects. Starting from the initial set of candidates, the first and second authors manually set up the projects leveraging the build system and directions provided in the corresponding

[2]Available at: https://github.com/acmsigsoft/EmpiricalStandards

Project Name	Description	Stars	Forks	N. Releases	N. Releases Analyzed	LOC	Link
Arthas	Java Diagnostic Tool	31,9k	6,9	47	43	61K – 46K	https://github.com/alibaba/arthas
Apollo	Configuration Management System for Microservices	27,8k	10,1k	38	8	88K – 90K	https://github.com/apolloconfig/apollo
Caffeine	High Performance Caching Library	13,2k	1,4k	65	9	70K – 83K	https://github.com/ben-manes/caffeine
Data Transfer Project	Transfer Data Online	3,4k	442	55	47	40K – 41K	https://github.com/google/data-transfer-project
ApkTool	Reverse Engineering	15,8k	3,3K	16	14	15K – 18K	https://github.com/iBotPeaches/Apktool
JSQL Parser	RDBMS agnostic SQL	4,2k	1,2K	30	4	50K – 55k	https://github.com/JSQLParser/JSqlParser
Disruptor	High Performance Inter-Thread Messaging Library	15,8k	3,8K	13	6	20K – 20K	https://github.com/LMAX-Exchange/disruptor
Mockito	Framework for Unit Tests	13,7k	2,4K	198	112	89K – 88K	https://github.com/mockito/mockito
MyBatis-3	SQL mapper framework for Java	18,2k	12,1K	39	13	100K – 98K	https://github.com/mybatis/mybatis-3
Eureka	AWS Service Registry	11,7k	3,7K	146	109	50K – 53K	https://github.com/Netflix/eureka
Hystrix	Latency and Fault Tolerance Library	23,2k	4,7K	79	40	75K – 48K	https://github.com/Netflix/Hystrix
Zuul	Gateway Service	12,5k	2,3K	5	4	35K – 31K	https://github.com/Netflix/zuul
RxJava	Library for Composing Asynchronous and Event-Based Programs for Java-VM	46,8k	7,7K	231	101	41K – 42K	https://github.com/ReactiveX/RxJava
Jadx	Dex to Java Decompiler	33,6k	4,2K	27	19	118K – 70K	https://github.com/skylot/jadx
Spring Data JPA	Data Access Layer Simplify	2,6k	1,2K	78	13	45K – 44K	https://github.com/spring-projects/spring-data-jpa

TABLE I: Overview of the projects analyzed.

GITHUB repository. However, 50% of the projects could not be successfully configured and built, due to incompatibility problems with the versions of some libraries. This issue is not uncommon in the context of mining software repositories, and was pointed out by Hassan *et al.* [16] when they performed a comparison among the main JAVA building systems. After filtering out the projects which could not be built, we were left with 23 candidates. To avoid considering projects irrelevant to our research questions, we ran the design pattern detection tool [35] and discarded projects containing no instances of design patterns. At the end of this process, we had identified 15 JAVA meeting the selection criteria and useful to our experiments, reported in Table I.

B. Design Pattern Extraction

To extract the design pattern instances implemented in the considered projects, we leveraged the tool by Tsantalis *et al.* [35], which we selected on the basis of two main aspects:

Detection Confidence. The tool can detect 10 kinds of design patterns, *i.e., Adapter/Command*, *Bridge*, *Singleton*, *Template Method*, *Proxy*, *State/Strategy*, *Decorator*, *Factory Method*, *Component*, and *Observer*, with 100% precision and no false positives [35]. These performances make the tool the state-of-the-art for design pattern detection. However, due to the identical UML structure of *Adapter/Command* and *State/Strategy*, the tool aggregates them into a single type, as they cannot be distinguished by an automated process [35].
Flexibility Taken Into Account. Due to the internal implementation of the tool, it can also identify custom implementation of known design patterns types.

To perform its task, the tool executes a number of steps. First, it analyzes the characteristics of the projects in terms of associations, generalizations, method invocations, and so on. At the end of this step, an *n x n* adjacency matrix is generated, where *n* represents the number of classes. Afterward, the tool identifies the inheritance hierarchies among the classes, considering all kinds of inheritance implemented in JAVA, *i.e.,* specification inheritance, implementation inheritance, and abstract classes, and leverages them to build a tree modeling the hierarchical structure of the project. Such a tree generates one or more subsystems that are then provided as input to a similarity score algorithm, which compares the identified subsystems with the structure of design patterns.

C. Code Smell Detection

To detect the code smells affecting the considered projects, we used DECOR [24], as previously done in similar work [9], [12], [15], [18] because it represents a good compromise between execution time and performance [5], [8], [24], reporting 100% recall, and precision greater than 50%.

In particular, DECOR employs a combination of multiple heuristic approaches to detect code smells in source code. Given a class A, the tool considers A to be affected by a code smell S *if and only if* for each metric used to estimate the presence of S, the following condition is verified: $metric_i \geq threshold_i$. The higher the difference between $metric_i$ and $threshold_i$, the greater the intensity of *S*.

We leveraged DECOR to detect three smells, *i.e., Complex Class*, *God Class*, and *Spaghetti Code*, as they represent the quality of the code in terms of understandability.

D. $\boldsymbol{RQ_1}$: Analyzing the Co-Occurrences of Design Patterns and Code Smells

To answer RQ_1, we calculated the frequency of classes participating in design patterns and, at the same time, being affected by code smells. To do this, we merged the data coming from the execution of the design pattern detection tool by Tsantalis *et al.* [35] and the code smell detector DECOR. We computed the number of classes that are involved in some design pattern and, at the same time, are affected by code smells. We normalized all the results using MIN-MAX in the range $[0; 1]$, and plotted the frequency of co-occurrences by means of heatmaps.

E. $\boldsymbol{RQ_2}$: Analyzing the Correlation between Design Patterns and Code Smells

To address $\mathbf{RQ_2}$, we built a statistical model analyzing the impact that the presence of a design pattern has on the emerging of code smells. In the following, we report the independent, dependent, and control variables involved in the analysis with the statistical model.

Independent Variables. We were interested in understanding whether and to what extent the presence of design patterns impacts code smells. For this reason, we considered design patterns as independent variables. We focused on 10 design patterns, *i.e., Adapter/Command*, *Bridge*, *Singleton*, *Template Method*, *Proxy*, *State/Strategy*, *Decorator*, *Factory Method*, *Component*, and *Observer*. The selection of such set of design patterns was driven by their availability, as they can be

extracted by the design pattern detection tool by Tsantalis *et al.* [35]. To avoid possible threats to validity, we considered the same aggregation on design patterns made by the authors of the tool; as the patterns *Adapter/Command* and *State/Strategy* share the same UML structure, it is not possible to automatically distinguish them by means of a tool.

Dependent Variables. As we aimed at understanding the impact of design patterns on the emerging of code smells, the presence of code smells affecting the code represented the dependent variable in our study. We focused on three code smells [10], *i.e.,* (1) *God Class*, affecting a class that implements several responsibilities, and it is invoked by most of the system to perform their actions, (2) *Spaghetti Code*, representing a class that implements long methods without parameters, and (3) *Complex Class*, that is a class being hard to understand and showing a high level of cyclomatic complexity. The main reason driving the selection of such smells is given by the claims made in previous work about them being representative of code complexity and comprehensibility, which are perceived as crucial for maintenance tasks in the perspective of developers [1], [2], [19], [25]. Furthermore, we decided not to consider additional known code smells, such as *Parallel Inheritance*, *Middle Man*, or *Refused Bequest* due to issues in the detection mechanisms. They have been formerly identified leveraging a custom version of DECOR that implements a dynamic approach for the detection [22]. Unfortunately, such version is not publicly available, and its re-implementation by the authors of this paper could have led to the introduction of errors or bias in the detection.

Control Variables. Conscious that external unconsidered factors can impact the fluctuation of the dependent variable, we considered a set of code quality metrics as control variables for our experiment, to avoid possible threats to the conclusion validity of our study. We selected five control metrics, *i.e., Lines of Code* (LOC), *Lack of Cohesion of Methods* (LCOM), *Number Of Attributes* (NOA), *Weighted Methods per Class* (WMC), and *McCabe's Cyclomatic Complexity* (CC); these metrics have been demonstrated to be good estimators for code quality [31], [33]. We extracted the control metrics by using DECOR; however, we remark that DECOR does not consider such variables during the estimation of the presence of code smells, which means that there is no direct correlation between the dependent and control variables of our study [4]. We manually assessed the possibility of multi-collinearity among the variables involved in our study, to avoid threats to the validity of our work, as explained in the following.

Statistical Model. Given the nature of the dependent variable, *i.e.,* the presence or the absence of a certain code smell, the *Generalized-Linear-Model* was used [34]. We selected this statistical model because it can be applied to estimate nominal variables that can assume two levels. We built the model using the *multinom* function provided by the *nnet*[3] package in R. Before running the statistical model, we took into account the multi-collinearity problem, occurring when

[3]https://cran.r-project.org/web/packages/nnet/nnet.pdf

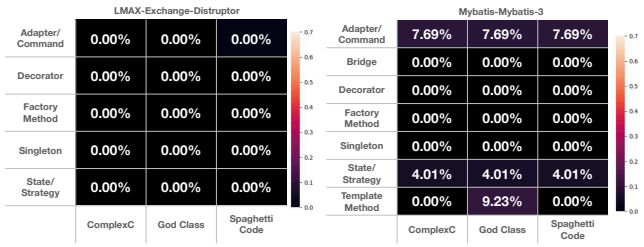

Fig. 2: Co-occurrences of code smells and design patterns.

two or more independent variables are bounded with a high level of correlation, and one of them can be used to predict the other. The presence of multi-collinearity among variables can bias the results, therefore, we followed the guidelines proposed by Allison *et al.* [3] to mitigate this threat. We did not remove any independent variable, because the standard error was, in any case, lower than 0.9, and interpretability problems arise with a standard error higher than 2.5 considering 95% prediction interval [23].

IV. Analysis and Discussion of the Results

In this section we report the results of our study and discuss about the implications of our findings.

A. On the Co-occurrence of Design Patterns and Code Smells

RQ_1 was focused on understanding whether and to what extent design patterns and code smells co-occur in the same classes. Figure 2 provides an overview of the results obtained from the co-occurrences analysis. For a matter of space, in this paper we report the data related to four projects, and we make the complete results available in the online appendix [13]. For each project, the figure depicts a heatmap reporting the extent to which classes participating in design patterns contained code smells. For example, in the MYBATIS-3 project, 7.69% of the classes participating in an instance of *Adapter/Command* were affected by the *God Class* smell.

The results obtained from the 15 analyzed projects were variegated, hinting at the observation that the co-occurrence of design patterns and code smells may vary depending on the project. By analyzing the frequencies reported in each heatmap, we noticed that two main patterns emerged, describing two families of projects. The first kind of project was characterized by design pattern instances completely free from code smells. That was the case of projects APOLLO, APKTOOL, DATA TRANSFER PROJECT, JSQL PARSER, DISRUPTOR, MOCKITO, and SPRING DATA JPA. In these projects, none of the classes participating in design patterns were affected by code smells. The opposite pattern arose from a set of seven projects which presented a high frequency of co-occurrence of design patterns and code smells, *i.e.*, HYSTRIX, CAFFEINE, MYBATIS-3, EUREKA, RXJAVA, JADX and ZUUL. Such projects exhibited code smells affecting classes participating in design patterns, and in each project the threatened types of design patterns went from two to

four. A single project, *i.e.*, ARTHAS, presented only one co-occurrence, in fact the *State/Strategy* pattern was the sole affected by the *God Class* and *Spaghetti Code* smells.

An interesting observation emerged from the analysis of co-occurrence, which showed that the *State/Strategy* pattern was touched by code smells in every project—except those not presenting any co-occurrence. In particular, in all the projects revealing at least or exactly one co-occurrence, classes implementing the *State/Strategy* pattern were affected by the *God Class* smell, in eight projects they also showed *Spaghetti Code* issues, and in four projects they presented *Complex Class* smells. We conjecture that this result is driven by the characteristics of the *State/Strategy* pattern itself, as its goal is to provide different behaviors depending on the current state of an object [11]. We suppose that as the behaviors to implement grow in number and size, the complexity of the involved classes also tends to increase. This observation remarks the non-triviality of the use of design patterns to enhance code quality and maintainability; as design patterns themselves risk being affected by the problems they aim at avoiding. The *Adapter/Command* pattern showed a similar trend to the *State/Strategy* one, as it was threatened by *God Class* and *Spaghetti Code* in five of nine projects, and by *Complex Class* in two projects. We observed that instances of the *Singleton* and *Bridge* patterns appeared in co-occurrence with code smells in four projects, followed by the *Template Method*, which emerged in three projects. Classes implementing the *Observer* and *Decorator* patterns were affected by code smells in two projects, while *Factory Method* implementations resulted being smelly in one project.

Key findings of RQ_1.

Classes participating in design patterns may be affected by code smells, resulting impacted by the same problems they are supposed to avoid. The *State/Strategy* pattern emerged in all projects as being threatened by code smells, followed by the *Adapter/Command* pattern, which resulted being compromised in six projects.

B. On the Impact of Design Patterns on Code Smells

With our second research question, we aimed at assessing how the presence of design patterns impacts the arising of code smells. By performing statistical analysis, we found that most design patterns did not influence the code into being affected by code smells. However, in nine cases, the analysis revealed that the implementation of design patterns determined the presence of code smells in a statistically significant way. Table II reports the complete results of the statistical analysis. The first observation we noticed studying the results was related to the *State/Strategy* pattern, as it appeared as the most co-occurring with code smells in the first phase of our research. Nevertheless, the statistical analysis revealed that its presence did not significantly affect the emerging of *Complex Class* and *God Class* smells, but only determined the code being *Spaghetti*. On the other hand, the *Adapter/Command* pattern turned out to be significant in the occurrence of the *God Class* smell and in a minor manner also for the *Spaghetti code*, in concordance with the results observed in RQ_1. The presence of *God Class* was significantly affected also by the *Bridge*, *Singleton*, and *Template Method* patterns, although the co-occurrences were found in few projects.

TABLE II: Results of the statistical model concerning the relationship between design patterns and code smells.

Design Pattern	Complex Class	God Class	Spaghetti Code
Adapter/Command	8.341	0.594***	-0.384*
Bridge	17.129	17.129***	0.265
Component	166.346	1.544***	-12.602
Decorator	125.670	-0.189	-0.808
Factory Method	1,063.142	-11.255	1.157**
Observer	68.508	1.801	1.380
Proxy	1,008.371	-11.891	-10.424
Singleton	93.883	1.722***	-2.392**
State/Strategy	-4.897	-0.036	0.349***
Template Method	-16.859	0.586***	-0.146
LCOM	-0.0001	-0.0003***	-0.0003***
LOC	0.014	0.007***	0.007***
McCabe	7.598	-0.005***	0.002***
NOA	-0.088	0.034***	0.004***
WMC	-0.008	0.074***	0.033***

Significance: *p<0.1; **p<0.05; ***p<0.01.

In contrast with the purposes of design patterns, which include guaranteeing code maintainability and comprehension, we found that their presence often leads to the introduction of code smells, which are signs of poor implementation practices instead. This led us to reflect on the importance of properly designing and applying best practices for code maintainability, as the effects of our choices can produce unexpected outcomes. We observed that, although design patterns are supposed to make the perfect code, they can be determinant for the arise of code smells. We conjecture that the motivation behind this phenomenon can be connected with the intention driving the introduction of design patterns; attempting to reorganize the code to make it better structured, developers actually introduce degrees of complexity, ultimately leading to code smells threatening the overall program comprehension.

Key findings of RQ_2.

The presence of design patterns does not regardless guarantee good quality, as they can be affected by code smells. In particular, the presence of a *God Class* can be associated with a number of patterns, such as *Adapter/Command*, *Bridge*, *Singleton*, and *Template Method*.

V. FURTHER DISCUSSION AND TAKE-AWAY MESSAGES

The analysis of the results ignited our reflection and thinking, leading us to distill a number of take-away messages we believe are meaningful for researchers and practitioners. In the following, we argue our discussion points that led to deriving such practical implications.

Design Patterns: The Double-edged Sword. By definition, design patterns are intended to improve the quality of code,

as they are introduced with the purpose of organizing the classes in such a way that good attributes of inheritance and delegation are enhanced. However, implementing such reuse mechanisms is not trivial, as developers may accidentally fall into mistakes, such as overuse, *e.g.,* adding design patterns when not necessarily needed, or misuse, *e.g.,* making a sub-optimal choice on the kind of pattern to employ for a specific problem. Such mistakes can eventually lead to bad effects, unnecessarily adding atoms of complexity to the code, which in turn can result being less comprehensible and even affected by smells, as observed in our study. Therefore, there is the need for developers to focus on properly applying design patterns [39], to avoid declining their positive effects into worsening the code quality.

☞ *A positive solution, if improperly used, can lead to unwanted consequences. Developers should commit to taking advantage of design patterns in a careful way, to avoid misuse or overuse mistakes.*

Awareness is the Hilt. We conjecture that one of the motivations behind the misuse and overuse mistakes discussed before lies in the implementation of design patterns without the appropriate attention. As the name suggests, design patterns are meant to be introduced at design time, when the responsibilities and relationship of classes are planned. At such a high level of abstraction, practitioners should not feel like applying design patterns just for the sake of implementing a reuse mechanism, but they should naturally get inclined towards the choice, by observing the problem definition and the relationships among classes. Afterwards, the introduction of a particular kind of design pattern should be supported by a dedicated phase of focused analysis, to assess whether it is worth employing, or risks adding unnecessary complexity. Nevertheless, practitioners should be conscious of the threats connected with misuse and overuse practices, to avoid applying them recklessly. We believe that awareness is the hilt empowering developers to handle the double-edged sword of reuse mechanisms. This raises the need (1) for practitioners to keep staying focused on the thinking and reasoning that is necessary at design time, and (2) for managers to encourage their teams to constantly learn and train.

☞ *Awareness of the intended purpose and misuse risks associated with design patterns is the key to them being properly used. The introduction of design patterns should be carefully planned at design time, thoroughly reasoning on the specific problem, to avoid making sub-optimal choices.*

VI. Threats to Validity

In this section, we recognize the possible threats that could impact the results of our study, and discuss the mitigation strategies that we applied.

Construct Validity. Construct validity refers to the relationship between *theory* and *observation*. The main concern regards the selection of the dataset leveraged in the experiments, as the choice of the dataset can influence the observed results. To mitigate this aspect, we adopted a rigorous process to select projects based on empirical evidence of their characteristics. On the one hand, we selected only popular projects publicly hosted on GitHub, estimating their popularity based on the number of stars. On the other hand, we only selected projects for which a building system was provided, and we manually inspected projects to ensure compatibility with the tools adopted to extract design patterns and code smells. Another possible threat to construct validity is concerned with the tool leveraged to extract data on dependent, independent, and control variables. To mitigate this aspect, we chose the state-of-the-art tools (1) to extract code smells and CK metrics, *i.e.*, DECOR, and (2) to detect design patterns, *i.e.*, the tool proposed and validated by Tsantalis *et al.* [35]. Although it comes with possible imprecision, *i.e.,* design patterns sharing the same UML structure (*Adapter/Command* and *State/Strategy*) are considered the same, it still represents the state-of-the-art for design pattern detection.

Internal Validity. Threats to internal validity are factors that could influence the observed results. In order to avoid threats affecting the statistical model employed to answer RQ_2, we kept an eye on CK metrics, which acted as control variables.

Conclusion Validity. The major threat to conclusion validity regards the application of statistical models to answer our second research question. We selected the Multinomial Logistic Linear model [34] due to the nature of the problem, and we also considered possible multi-collinearity to avoid any interpretation bias.

External Validity. Threats to external validity are linked to the generalizability of the observed results. We analyzed 542 releases of 15 different projects in terms of scope and size. We are aware that generalizability can depend on multiple aspects, such as programming language; however, as part of our future work, we plan to extend this study, considering a broader set of projects to analyze, selecting them according to a variety of programming language, domain, and size.

VII. Conclusion and Future Work

This paper presented a preliminary analysis of the relationship between design patterns and code smells. We analyzed 15 Java projects consisting in over 542 releases, as we were interested in (1) assessing the co-occurrence of design patterns and code smells, and (2) measuring how the presence of design patterns impacts the appearance of code smells.

We found that classes participating in design patterns are often affected by code smells themselves, hinting at the fact that not everything that is supposed to be beneficial for code quality is actually advantageous without prejudice. As in the concept of *Yin and Yang*, nothing is completely and purely white or black; but there is always some darkness in the light. Even though we expect design patterns to be absolutely good for the code, they can be associated with some drawbacks. In fact, we observed that out of 10 design patterns analyzed, seven showed a positive correlation with the presence of at least one code smell. This finding drives us to encourage managers, designers and developers to continuously monitor

the evolution of design patterns in their software, as they may end up being affected by quality issues.

As a future part of our agenda, we want to extend our experiments by considering a larger dataset to assess the reported findings. Finally, by surveying developers, we aim to grasp the developers' perspectives on the impact of design patterns on code smells.

ACKNOWLEDGMENT

Fabio is partially funded by the Swiss National Science Foundation through the SNF Projects No. PZ00P2_186090.

REFERENCES

[1] F. Albalooshi. The metrics of multiple inheritance and the reusability of code–java and c++. *Journal of Advances in Mathematics and Computer Science*, pages 1–12, 2016.

[2] F. Albalooshi and A. Mahmood. A comparative study on the effect of multiple inheritance mechanism in java, c++, and python on complexity and reusability of code. *International Journal of Advanced Computer Science and Applications*, 8(6), 2017.

[3] P. Allison. When can you safely ignore multicollinearity. *Statistical horizons*, 5(1):1–2, 2012.

[4] J. S. Almeida. Predictive non-linear modeling of complex data by artificial neural networks. *Current opinion in biotechnology*, 13(1):72–76, 2002.

[5] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138, 2019.

[6] J. M. Bieman and J. X. Zhao. Reuse through inheritance: A quantitative study of c++ software. *ACM SIGSOFT Software Engineering Notes*, 20(SI):47–52, 1995.

[7] A. Blewitt, A. Bundy, and I. Stark. Automatic verification of design patterns in java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 224–232, 2005.

[8] E. V. de Paulo Sobrinho, A. De Lucia, and M. de Almeida Maia. A systematic literature review on bad smells—5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering*, 2018.

[9] M. De Stefano, F. Pecorelli, F. Palomba, and A. De Lucia. Comparing within-and cross-project machine learning algorithms for code smell detection. In *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*, pages 1–6, 2021.

[10] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[11] E. Gamma, R. Johnson, R. Helm, R. E. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[12] G. Giordano, A. Fasulo, G. Catolino, F. Palomba, F. Ferrucci, and C. Gravino. On the evolution of inheritance and delegation mechanisms and their impact on code quality. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 947–958. IEEE, 2022.

[13] G. Giordano, G. Sellitto, A. Sepe, F. Palomba, and F. Ferrucci. The yin and yang of software quality: On the relationship between design patterns and code smells – online appendix. https://giammariagiordano.github.io/TheYinAndYangOfSoftwareQuality/.

[14] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, 2002.

[15] C. Hasantha. A systematic review of code smell detection approaches. *Journal of Advancement in Software Engineering and Testing*, 2021.

[16] F. Hassan, S. Mostafa, E. S. Lam, and X. Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 38–47, 2017.

[17] P. Hegedűs, D. Bán, R. Ferenc, and T. Gyimóthy. Myth or reality? analyzing the effect of design patterns on software maintainability. In *ASEA and DRBC 2012, held in conjunction with GST 2012, Jeju Island, Korea, November 28-December 2, 2012*. Springer, 2012.

[18] S. Jain and A. Saha. Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection. *Science of Computer Programming*, 212:102713, 2021.

[19] F. Khomh and Y.-G. Gueheneuc. Do design patterns impact software quality positively? In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 274–278, 2008.

[20] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17:243–275, 2012.

[21] B. M. Lange and T. G. Moher. Some strategies of reuse in an object-oriented programming environment. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 69–73, 1989.

[22] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis. Identification of refused bequest code smells. In *2013 IEEE International Conference on Software Maintenance*, pages 392–395. IEEE, 2013.

[23] D. N. McCloskey and S. T. Ziliak. The standard error of regressions. *Journal of economic literature*, 34(1):97–114, 1996.

[24] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2009.

[25] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 482, New York, NY, USA, 2018. Association for Computing Machinery.

[26] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 482–482, 2018.

[27] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 101–110, 2014.

[28] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 155–165, New York, NY, USA, 2014. Association for Computing Machinery.

[29] N. Shi and R. A. Olsson. Reverse engineering of design patterns from java source code. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 123–134. IEEE, 2006.

[30] S. Singh, S. Singh, and G. Singh. Reusability of the software. *International journal of computer applications*, 7(14):38–41, 2010.

[31] G. Succi, W. Pedrycz, S. Djokic, P. Zuliani, and B. Russo. An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. *Empirical Software Engineering*, 10(1):81–104, 2005.

[32] D. Taibi, A. Janes, and V. Lenarduzzi. How developers perceive smells in source code: A replicated study. *Information and Software Technology*, 92:223–235, 2017.

[33] D. A. Tamburri, F. Palomba, and R. Kazman. Success and failure in software engineering: A followup systematic literature review. *IEEE Transactions on Engineering Management*, 2020.

[34] H. Theil. A multinomial extension of the linear logit model. *International Economic Review*, 10(3):251–259, 1969.

[35] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006.

[36] M. Vokáč, W. Tichy, D. I. Sjøberg, E. Arisholm, and M. Aldrin. A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment. *Empirical Software Engineering*, 9:149–195, 2004.

[37] B. Walter and T. Alkhaeir. The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology*, 74:127–142, 2016.

[38] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[39] C. Zhang and D. Budgen. What do we know about the effectiveness of software design patterns? *IEEE Transactions on Software Engineering*, 38(5):1213–1231, 2012.