CHALMERS
UNIVERSITY OF TECHNOLOGY

Impact of Co-Occurrences of Code Smells and Design Patterns on Internal Code Quality Attributes

Wiley
IET Software
Volume 2025, Article ID 5579438, 13 pages
https://doi.org/10.1049/sfw2/5579438

IET The Institution of Engineering and Technology | WILEY

Research Article

Impact of Co-Occurrences of Code Smells and Design Patterns on Internal Code Quality Attributes

Sania Imran,[1] Irum Inayat ,[2] and Maya Daneva [3]

[1]*Department of Software Engineering, FAST National University of Computer and Emerging Sciences, Islamabad, Pakistan*
[2]*Department of Computer Science and Engineering, Chalmers | University of Gothenburg, Gothenburg, Sweden*
[3]*Semantics, Cybersecurity, and Services Group, University of Twente, Enschede, Netherlands*

Correspondence should be addressed to Irum Inayat; irum@chalmers.se

Received 21 November 2024; Revised 31 May 2025; Accepted 22 September 2025

Academic Editor: Hui Liu

The structural features of a code section that may indicate a more serious issue with the design of a system or code are known as code smells. Design patterns, on the other hand, are meant to describe the best reusable solution for creating object-oriented software systems. Even though design patterns and code smells are very different, they may co-occur. In fact, there may be a significant connection among the two, which requires further research. This study aims to (i) identify design patterns and code smells in web gaming code, (ii) investigate the co-occurrence of the two, and (iii) analyze the effects of these co-occurrences on internal quality aspects of code. An experiment is carried out on JavaScript (JS) web games utilizing machine learning classifiers to investigate the influence of co-occurrence on potential code smells and design patterns to evaluate games from a quality perspective. Moreover, statistical testing is performed to identify the impact of co-occurrences of code smells and design patterns on internal quality attributes. After examining the data, we determined that random forest is the most effective classifier, achieving an accuracy of 99.126% and 98.99% for both experimental situations, respectively. Moreover, on applying the Wilcoxon signed rank test, we found that co-occurrence has no impact on the coupling and complexity of web games codes, whereas there is a significant impact of co-occurrence on cohesion, size, and inheritance. Our results may guide developers in writing efficient games code to add to this swiftly growing market.

1. Introduction

Many software systems must continually adapt to changes to meet new needs and environmental modifications. Therefore, high-quality source code is essential, as it should be easy to understand, analyze, modify, maintain, and reuse. However, as software systems change overtime, their quality structure may deteriorate [1, 2]. Quality attributes are significant indications of high-quality software development. More often than not, these quality constructs, be it internal or external are attained through design patterns. Empirical research [3, 4] suggests that design patterns [5, 6] influence software quality attributes [7, 8]. This suggests that such reusable solutions in software design could either positively [6] or negatively [8] affect software quality attributes, which can hinder system performance and result in unnecessarily code if not applied appropriately. However, software engineers sometimes generate suboptimal code that may introduce design issues, i.e., code structures contradicting fundamental software engineering (SE) principles. These modifications to software systems are implemented without any consideration due to market competition, work deadline pressure, or the developer's lack of experience. Among the most common sources of code smells is violating certain design principles [9]. According to Sharma [10], in most cases, code smells are not the problem by and in themselves; instead, they are just indicators of the presence of a problem. This means that code snippets exhibiting code smells are not considered defects. However, these structures may hinder the implementation of a software system's internal components [11, 12] and harm its design and interpretation. When such scenarios occur, it is recommended that the software system in question be refactored [11, 13] to remove potentially

problematic parts of the source code. This opens up the need to deal with these challenging areas by adopting design patterns to attain system quality.

To address quality issues about a software system that has undergone numerous changes, empirical SE researchers investigated the relationship between the existence of reusable solutions in the design of software and undesirable code flaws. Previously published research on the subject matter [14–16] emphasized on the linkages and the dependencies between two or more smells and how they affect quality traits. Furthermore, existing empirical investigations are either language (e.g., Java or C++) or domain-specific. In addition, most of the published papers addressed isolated instances about code problems and how they affected quality aspects in specific case study settings. However, empirical researchers rarely examine the connections between concurrently observable smells and design patterns in the same research environment [17, 18]. So there exists a scarcity of empirical knowledge about the coexistence of design patterns and code issues and how these co-occurrences effect quality metrics that hinder code structures and maintainability. In an effort to reduce these deeper problems in code and enhance system quality, SE scholars initially tried to comprehend the nature of the connection between design patterns and code smells. With the significant exception of Javascript JS, it is clear from the body of previous work that the relationship between the two has been studied in many of the widely used programming languages. To our knowledge, no such study has investigated the context of this language. On the other hand, JS is a mainstream language, and its use is still growing [19–21] in every application domain of software development. Thus, there is still opportunity for empirical research on the connection between JS design patterns and code smells. Our goal for this research was to look into this relationship in the gaming sector [22, 23].

To explore the impact of the co-occurrence of code smells and design patterns on quality attributes, this work aims to identify the code smells and design patterns in web games along with co-occurrences. To this end, five open-source medium-sized web games in JS from GitHub are selected, then data is extracted. The extracted data serves as input to a dataset formulization phase where data preprocessing normalizes the data and replaces missing values with the matching feature average. After normalization, data splitting is done, on which machine learning models such as Discriminant Analysis (LDA, QDA), Naive Bayes (MNB, CNB, GNB), SVM, linear SVM, KNN, Random Forest, Decision Tree, and Neural Network are applied. At this stage, the classifiers are trained on balanced and unbalanced datasets. In order to determine which classifier performs best, the impact of co-occurrence of code smells and design pat- terns on quality attributes was examined. Then each trained classifier was evaluated using testing data in terms of precision, accuracy, F-measure, recall, and kappa. Once these results are obtained, to test this hypothesized relationship, a Wilcoxon signed-rank statistical test is applied to validate the proposed solution. As we will see in the article, we found a statistical significance of the impact of potential code problems and the co-occurrence of design patterns on internal quality attributes.

The remaining sections of the paper are organized in the following manner: Section 2: discusses the theoretical groundwork required for comprehending the key themes of this work. It also includes a brief literature review. Section 3 outlines the experimental design utilized to evaluate the research topics as well as the research technique used in this work. Our results are reported and discussed in Section 4. It also contains solutions to our research concerns. The risks to this empirical study's validity are discussed in Section 5. The conclusion and reflection on the constraints and our prospects for the future are covered in Section 6.

2. Related Work

The connection between software code quality and design patterns has preoccupied scholars in the empirical SE community for more than a decade. However, as Alkhaier and Walter [24] state, while much has been published, it is with mixed results. In this part, we describe what is known about the link between design patterns and the quality of software, covering three streams of publications (1) investigations into the effects of code flaws and design patterns on the quality of software, (2) exploration of code smells in various programing languages, and (3) studies on the connection among design patterns and code smells. These three streams of research form the background for our work.

2.1. Code Smells in Different Programing Languages. The structural characteristics regarding a segment of code that might point to a more serious issue with the code or the design of systems are known as code smells. As a result, code smells serve as a barrier in the quality improvement process of software. Fowler [25] recognized smells from code as indicators of design faults, which may make maintenance of software problematic. But, as was already stated in the introduction, code smells are merely indicators that developers should look for a more serious problem. Code smells might point to areas of code that need to be investigated further.

Various studies have been published on different programing languages in the context of code smells [15, 16, 19]. Modern software applications increasingly combine multiple programing languages instead of just using one. For example, Ramos et al. [26] conducted an empirical study considering five different languages: Java, C, C++, JS, and Python. The study focuses on three specific characteristics such as efficacy, efficiency, and influence to explore the application of a transfer learning approach. The preliminary results show that transfer learning can help developers and researchers apply the same code smell detection strategies in every programing language. However, due to the availability of structural differences in code, Python and JS show different behaviors and strategies. In a comparable way, a recent SLR by dos Reis et al. [27] on the most advanced methods and resources for code smell detection and visualization revealed that code smell detection is a challenging task. Additional efforts are still required to improve the diversity of code smells and supported programming languages, as well as to lessen the subjectivity involved in their description and detection. In their analysis, the authors observed that 77.1% of the studies use Java as a target language followed by the C

language. The least number of studies (1.2% and 2.4%) target the Java Android and JS web languages, respectively.

In light of the above finding, as part of preparing this paper, we searched for publications that specifically explored code smells in the contexts of web-based languages. We found four studies [19, 28–30] particularly relevant to our work. Rio and Brito e Abreu [28] conducted a long-term investigation examining the reliability of code smells in PHP-based web applications. Using a survival analysis technique, these authors looked at the appearance of six code smells in eight server-side PHP web applications. The findings show that the scope of code smells determines their survival. This means localized code smells have a 4-year lifespan, while dispersed code smells have a 5-year lifespan. Next, the mapping study [19] explored the extent to which code smells are present in the JS language used in information systems. In the context of the JS language, these authors discovered 26 different kinds of code smells. The conclusion, however, is that no growing trend is observable because the number of work investigating bad smells in JS is still quite limited.

It is also important to note that although these studies [21] looked at the prevalence of code smells in web-based projects, they failed to take into account their presence in the area of gaming applications. In the field of web-based games, which is recognized for its rapid expansion, not much study has been published on examining code smells. Agrahari and Chimalakonda [29] used JSNose, a JS code smell detection tool, to perform an exploratory study. This study determines the prevalence and distribution of code smell in web games. This was done in order to better understand the extent of code smells in games and validate the need for game-specific code smell detection tools. The study revealed violations of existing game programing patterns. Similarly, the work of Khanve [30] looked at the existence of code smell in games. According to the findings, it is not enough to use the JS code smell detection tool to find code smells unique to a given web game.

2.2. Impact of Code Smells and Design Patterns on Quality Attributes. Code smells arise as a result of program changes throughout the evolution phase, making maintainability more difficult. While much research efforts on the effects of individual code smells on quality attributes, there are only a few reviews [31–33] on the concurrent presence of two or more code smells and their combined impact on quality attributes. Kaur [31] conducted a thorough litera- ture review to ascertain how code smells affected software quality metrics. The results of this review showed that distinct code smells have conflicting effects on different software quality parameters, suggesting that code smells have a variable impact on software quality. Similarly, reviews of code smells by Lacerda et al. [32] and de Paulo Sobrinho et al. [33] showed that further research is needed to fully understand certain specific smells, such as Shotgun Surgery and Refused Parent Bequest. However, when it comes to co-occurrences, code smells like Feature Envy, God Class, and Long Method are the ones that are most studied. The results of both reviews [32] and [33] also imply that code smells in source code could indicate problems with the architecture of software and maintainability, respectively.

Abbes et al. [34] investigate the relationships and consequences of code smells. The authors came to the conclusion that isolated smells from code had no effect on maintainability, while interconnected code smells resulted in a considerable maintenance effort. Likewise, Yamashita and Moonen [35] investigated the impact of co-occurrences on the maintenance of four medium-sized Java systems. The investigators discovered that co-occurring smells have a negative impact on software maintainability and efforts. Last but not least, Oizumi et al. [36] carried out a study to ascertain whether the co-occurrence of code smells, or what the authors call "agglomerations" can be a symptom of software design problems. According to the findings, code smells that co-occur can lead to issues with software design and are a good way to find these issues.

In addition, we looked at studies on how code affects internal quality aspects [11, 12]. The majority of refactoring types, according to these authors, enhanced one or more internal quality attributes, while re-refactoring had an impact on internal quality attributes that was comparable to that of general refactoring. Politowski et al. [37] likewise carried out a research with 372 comprehension tasks and 133 study participants that involved a combination of two code smells, Blob and Spaghetti Code. Based on these two abnormalities occurring together, the study sought to determine the developers' level of source code understanding. The authors' findings revealed that as developers spent longer to complete their tasks, the readability and comprehension of the code deteriorated.

In addition, Martins et al. [38, 39] looked at how code smell co-occurrences affected internal quality attributes in closed-source Java systems. Based on the developers' viewpoint, authors determined which co-occurrences should be eliminated. Last, using the bit string approach, Almadi et al. [40] examine the co-occurrence of bad smells in open-source software. With the help of this approach, the authors analyzed and implemented a detection algorithm to reveal bad smell patterns and hierarchical relationships of their occurrences at different levels of abstraction. As a result, it was determined that software versions with minor differences in their co-occurrence percentage exhibit co-occurrence patterns.

In every pertinent study on how code smells affect software quality attributes, researchers also looked at how design patterns affect software quality attributes. This is because design patterns strive to reduce coupling and increase flexibility. By deferring choices until run-time, many design patterns increase code flexibility and facilitate the addition of new features without significantly chang- ing the existing code. In this regard, Ampatzoglou et al.'s [3] SLR documented design patterns to enhance designer-developer commu- nication by creating shared terminology. This is observed to improve the code readability, the understandability, and the maintainability of system designs by leveraging well-known and well-understood ideas. Additionally, Vokac's [7] empirical study examined the link between the assessed patterns and defect frequency, both positively and negatively.

2.3. Relation Between Code Smells and Design Patterns. While poor smells indicate that there are no design or code flaws,

Table 1: Details of the games included in this research.

Game name	JS files	JS LOC	Stars	Issues	Category
Clumsy bird	7	571	1210	3	Arcade
Javascript snake	5	846	3	1	Arcade
Diablo-js	1	743	767	5	Role Play-Ing
Hextris	15	1935	1478	16	Puzzle
3D-Hartwig-chess-set	4	1511	322	5	Board Game

good software design and code are linked to design patterns. Because they reflect oppositional structures, design patterns and bad smells are rarely studied in the same research setting. Several research have been undertaken in this area, focusing on the association of the two concepts. Sousa et al. [15] studied the correlation between code smells and design patterns at the category level. Nevertheless, the authors concluded that there is no connection between design patterns and code smell at different granularity levels based on the results of the empirical analysis. To uncover co-occurring instances. Sousa et al. [41] studied five systems and found correlations between the two. The findings identified the circumstances where design patterns are overused or misused and then developed criteria for their proper application.

Moreover, the mapping study of Sousa et al. [16] investigated the link between design trends and code issues by identifying software external quality attributes, co-occurrences, and refactoring. Another pertinent study [42] examined object-oriented software that uses design patterns that could be associated with smells. For this purpose, a case study is carried out with five Java open-source software and considered 11 design patterns. The results indicate a low co-occurrence of bad smells with the factory and composite methods. In contrast, the Template Method and Observer have a significant co-occurrence with God Class and Long Method [42].

Given the contradictory results of the impact of code smells and design patterns on quality attributes and the inconclusive findings about relationships between code smells and maintainability aspects as a result of design pattern applications, more empirical research is needed to add empirical evidence to the pool of knowledge already produced. Only then could empirical SE researchers arrive at conclusions regarding the joint effects of design patterns and code problems on important internal and external quality attributes.

3. Our Empirical Investigation

This empirical investigation is threefold: (i) find patterns of design and code flaws, (ii) look into the co-occurrences of these two phenomena, and (iii) assess how co-occurrences affect internal quality attributes. In line with this aim, we used the Goal-Question-Metric method [43] widely employed in empirical SE research, to formulate the goal as follows:

Analyze the influence of co-occurrence on internal quality attributes, to look into the influence of co-occurrences, concerning quality metrics that are internal in nature; from the viewpoint of pioneers in SE research or developers, in the context of medium-size open-source JS based web games. In light of the research goal, we arrive at the following research questions and working hypotheses.

RQ-1: What types of code smells are commonly found in web-based games?

RQ-2: Which design patterns are used in web games?

RQ-3: Which code smell and design pattern co-occurrences occur in web games?

RQ-4: What is the impact of the co-occurrence of CSs and DPs on the internal quality attributes?

In consideration of this, the null and alternative hypotheses listed below have been developed:

Null Hypothesis = H_0: Co-occurrence of CSs and DPs has no signif- icant impact on the internal quality attributes.

Alternate Hypothesis = H_1: Co-occurrence of CSs and DPs has a significant impact on the internal quality attributes of the system.

We employed Wohlin et al.'s [44] methodological principles for our research design. In the sections that follow, we first present information about our study context before reporting on data collection, analysis, and findings.

3.1. Context Overview. The scope of this investigation involves deciding on target software systems, obtaining quality metrics and attributes, and finding design patterns and code smells.

3.1.1. Selection of Target Systems. Our target systems are from GitHub. These are medium-sized open-source software systems and are selected by applying the selection criteria already used by researchers in similar research contexts [29, 30]. These criteria state: (1) all the systems must be in JS programing language, (2) the systems must be from the domain of web games, and (3) the size of the systems must be bigger in lines of code (LOC) than the specified threshold of 250 LOC. Moreover, the systems selected are from different categories of games, such as arcade, puzzle, board games, and roleplaying games. Table 1 list the key features of the games used for this research: game name and category, number of JS files, number of JS LOC, number of stars, and number of issues.

After selecting those projects, we cloned them. The specific process of cloning these projects onto local devices is reported in the "Data Collection" phase below. Once the source code from GitHub is cloned, we extract quality metrics from these projects as discussed below.

3.1.2. Extracting Quality Metrics and Attributes. To examine the effect of co-occurrence, we chose five aspects of code quality. These were selected because they are (i) widely recognized and documented in the literature, (ii) capable of evaluating a wide range of code-level elements, including class, methods

Table 2: Details of the quality attributes selected for the proposed study.

Quality attributes	Friendly name	API name
Coupling	Coupling between objects (CBOs) fan-in (FANIN)-base classes fanout (FANOUT)	Count class couple CountClassBase CountOutPut
Cohesion	Lack of cohesion of methods (LCOM)	PercentLackOfCohesion
Inheritance	Depth of inheritance tree (DIT) Number of children (NOC)	MaxInheritanceTree CountClassDerived
Size	Lines of code (LOC) lines with comments (CLOC) instance variables (NIV) instance methods (NIM)	AltCountLineCode CountLineComment CountDeclInstanceVariable CountDeclInstanceMethod
Complexity	Cyclomatic complexity (CC) weighted method count (WMC) response for class (RFC) essential complexity (Evg) paths (NPATH)	Cyclomatic SumCyclomatic CountDeclMethodAll Essential CountPath

Table 3: GOF design patterns identified.

	Purpose		
Scope	Creational	Structural	Behavioral
Class level	Factory	Adapter	Interpreter Template method chain of responsibility command
Object level	Abstract factory builder prototype singleton	Bridge composite decorator facade proxy	Iterator meditator momento flyweight observer state strategy visitor

and package, and (iii) can be computed using available static code analysis tools. Table 2 lists the most studied quality attributes along with software metrics and their API Name.

Furthermore, we used the Understand Tool [45, 46] to extract the metrics. We chose it, because it takes code in JS language as input and it computes all five internal quality attributes and their related measures [45, 46]. Moreover, this tool is a reverse engineering code exploration tool that is used in many different industries to both analyze and develop software.

3.1.3. Identification of Code Smells. A two-phase method is used to identify the code issues. The first phase is about identifying code smells from the existing literature, while the second consists of identifying code smells using some automated tools. While various tools for detecting JS code smells are available in the literature, the two most popular are JSNose and JSpIRIT [29, 30]. Our work used the JSNose tool, a metric-based methodology that blends dynamic and static analysis. This choice is because of the following: (i) the JSNose tool is a subset of the JSpIRIT-detected smells; (ii) we chose this automated method since manual code smell identification is time-consuming and error-prone, and it only produces a limited number of generic smells; (iii) the tool reduces development costs by assisting developers in identifying code smells in their source code, and inferring dynamic changes in object properties and functions at runtime [47].

3.1.4. Identification of Design Patterns. To find the design patterns, we used the automated technique provided in the tool of Tsantalis et al. [48]. It employs a similarity scoring method between graph vertices and uses a collection of metrics that represent all the important characteristics of their static structure. Table 3 summarizes all the design patterns found in the source code.

As already stated, after selecting the projects for our empirical research, we cloned them to extract the information we needed for our hypothesis-testing. For this purpose, a Python script was written to clone the game repositories using game repo links. We first stored all the links to our selected projects in a single text file to do this. Then, we extracted the issue count, stars, and language from the game repositories to create a metadata file. This information is reported in Table 1. Once done with cloning, the next step is to prepare the dataset for analysis.

3.2. Data Collection. As reported earlier, we used the source code of the five selected medium-sized open-source JS-based web games for the data collection phase. We cloned those web games onto the local device, after which we obtained a total of 32 JS files that contained a total of 5606 JS LOC that we used for experimentation. We used the Understand static analysis tool to extract quality metrics, which allows complete code navigation, control flow graph generation, and metric generation. We computed the 15-quality metrics described in Table 2 to extract the internal type of quality attributes. For example, to calculate the coupling of the files, we will have to calculate the following metrics: "Coupling between Objects (CBO)", "Fanout

Table 4: Selected design patterns.

Design patterns	Description
Adapter	Combine interfaces from distinct classes
Bridge	The interface and implementation of an object are separated.
Composite	A tree-like structure is used to arrange simple as well as intricate objects.
Facade	An entire subsystem represented by just one class.
Decorator	Dynamically assign responsibilities to objects
Proxy	An object that is a representation of another object.

(FANOUT)", and "Fanin (FANIN)". Following the project data analysis, the next stage is to discover code smells in projects. To find out how often code smells occur in JS web games and where they are available, we used an existing tool called JSNose [47]. More in detail, to identify the code smells in each project, the tool uses as input a text file with the links of the project. The output is the smells themselves, grouped into 13 types of smells. Next, we used the design pattern detection similarity scoring (DPDSS) tool to extract design patterns. As we are interested in those structural types of design patterns concerned with class and object composition, we selected six structural design patterns presented in Table 4 for our extraction phase.

3.3. Data Interpretation. The following part details the analysis we performed on the gathered data to address our research issues. Figure 1 shows how our data analysis fits with our data collection and forms a coherent whole in our empirical research process. The stages of our data analysis are described below.

Stage 1: Dataset building: Given our sets of identified code smells and design patterns, we employed an association rule mining technique to identify the co-occurrence relation between the two. This mining technique indicates how often adapter pattern and lazy object code smell are simultaneously present in a project. For this purpose, 78 (i.e. $6 \times 13 = 78$) different significant rules are made. Those rules consist of a combination of six structural design patterns with 13 types of code smells. Based on these rules, available co-occurrence are identified. In this way, the dataset is prepared by combining quality metrics and results of co-occurrences.

Stage 2: Dataset preprocessing: This stage is concerned with cleaning and preparing our data for statistical analysis. Since not all algorithms could handle missing or negative values in the input datasets, we performed a data preparation step. This step normalizes the data in the [0, 1] range and fill in the missing values with the relevant feature average. In this work, we removed the columns with constant or not a number (NaN) values. Given the preprocessed data, our next step was to perform data splitting. For this, we adopted a cross-fold validation technique. This method divides data into k distinct sections. Each iteration uses K-1 parts to train the model, with the remaining portion acting as a validation set.

Stage 3: Model training: We used a variety of machine learning classifiers to train the models at this step. This includes Neural Network, Random Forest, Decision Tree, SVM, Linear SVM, KNN, Naive Bayes (MNB, CNB, GNB), and Discriminant Analysis (LDA, QDA). We prefer using machine learning models [49, 50] for the following reasons: (i) As mentioned earlier, our selected open-source software systems are medium-sized, resulting in a smaller dataset of co-occurrences and quality attributes. Since extensive training is not required, ML techniques are suitable. (ii) For structured, well-labeled smaller datasets, ML techniques minimized the risk of overfitting. (iii) Given that we have employed rule-based association mining, traditional ML techniques offer better interpretability. To train the hyperparameter tuning and evaluate the models, we conducted an experiment with and without data balancing to find the best classifier that points out the impact of co-occurrence of code smells and design patterns on internal quality attributes. For the case of the imbalanced dataset, we used synthetic minority over-sampling technique (SMOTE). Well, according to authors [51, 52], SMOTE isn't the most effective technique, but in our experiment, it improves performance and reduces false negatives. Ultimately, every classifier that is trained has been assessed using testing data in terms of performance metrics. This includes the use of Cohen's kappa, accuracy, precision, F-measure, and recall. The reason these performance metrics are used is because they are popular and frequently utilized in the literature.

Stage 4: Statistical analysis: We used the Wilcoxon signed rank test [53] on both settings to determine whether the null hypothesis was accepted or rejected. The nonparametric Wilcoxon test was employed to investigate if the co-occurrence of design patterns and code smells had a significant effect on the quality attributes. The nonnormal distribution of our data led to the selection of this test. Ultimately, the p-value is calculated, and findings are produced with a 95% confidence interval, while taking a threshold of significance of 0.05 into account.

4. Results and Discussion

We summarized our findings in this section. As previously stated, our list of code smells in web games was compiled from two distinct sources: (1) prior studies on code smells in web games and (2) JS code smells discovered via the JSNose tool. After employing JSNose and manual literature analysis, we were able to identify 13 different kinds of smells in selected web games.

To make things clearer, the code smells in selected projects are visually displayed in the image below. The available code smells with a list of projects, are depicted in Figure 2. Individual code smells pertaining to each project are noted on the graph's

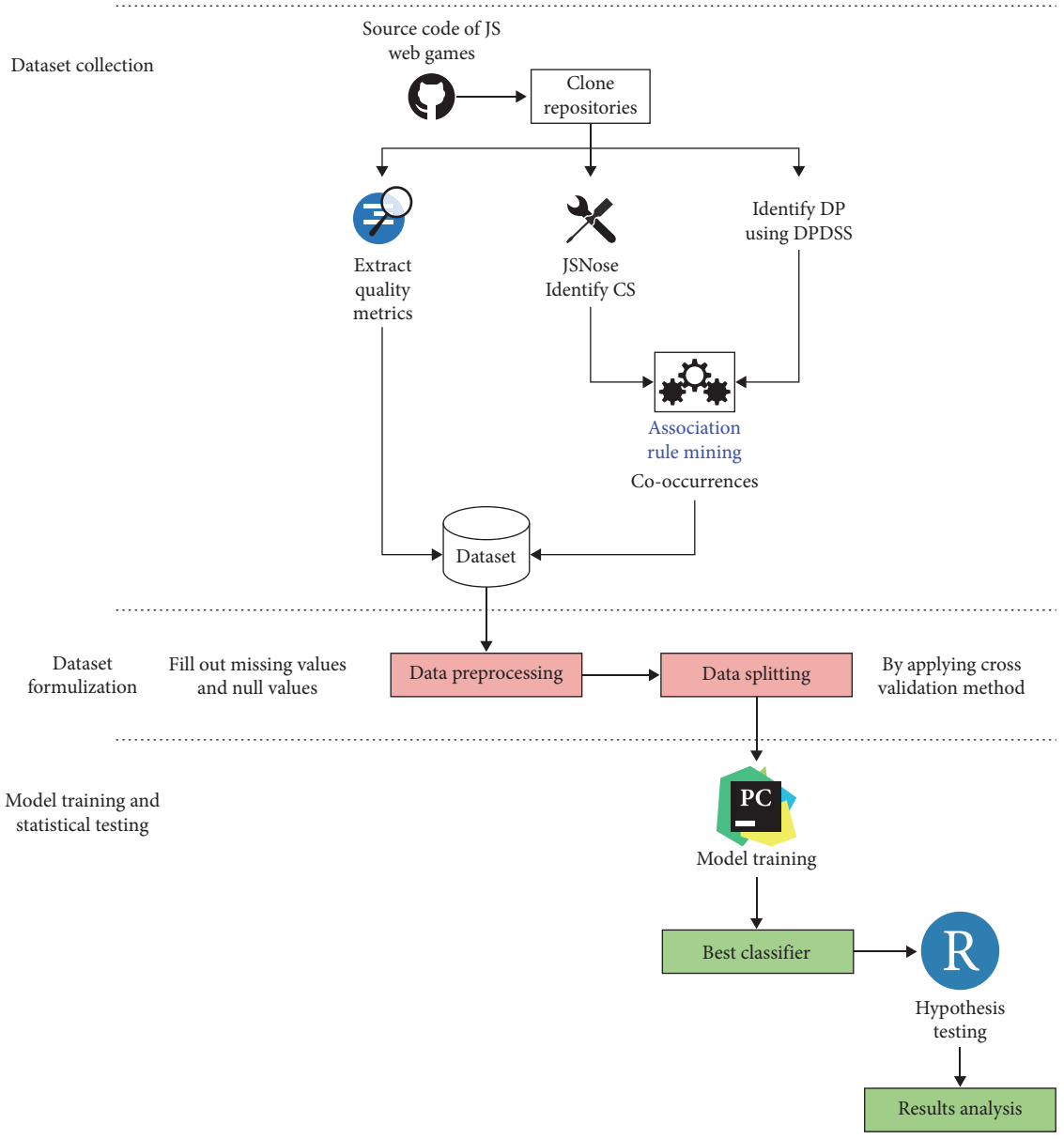

FIGURE 1: Our data analysis process.

horizontal axis, while details of a selected projects are noted on the vertical axis. Additionally, the overall count of code smells for each project is mentioned. Color-coding is done for readability.

From the graphics in Figure 2, it is evident that all five projects contain many lazy object types of code smell compared to other code smells. This means that object levels of code smells are present in large amounts that may be collapsed or combined into other classes to improve the structural quality.

According to the findings obtained from the first question we investigated (RQ1), 13 different types of code smells can be found in the web games we selected. We may deduce from the visuals in Figure 2 that the most common design approach, while creating web games is the lazy object kind of code smell. Following that comes the closure smell, which is prevalent in the majority of the projects chosen for this empirical investigation. Variables from the inner function, accessibility of the outer function, and use of several nested functions in the games code are the main causes.

To discover architectural patterns, a DPDSS tool is used, which relies on the notion of similarity scoring among graph vertex. However, before identifying patterns in web games, a representation of the structure that includes all the information essential to pattern recognition must be defined. This is accomplished by mapping the class diagram into a matrix of squares and then using a similarity scoring technique. The rationale for mapping class diagrams into square matrices is because this kind of representation is appealing to engineers and computer scientists, as stated by Tsantalis et al. [48]. Table 5 depicts the available structural type of design patterns and their occurrence with respect to all selected projects.

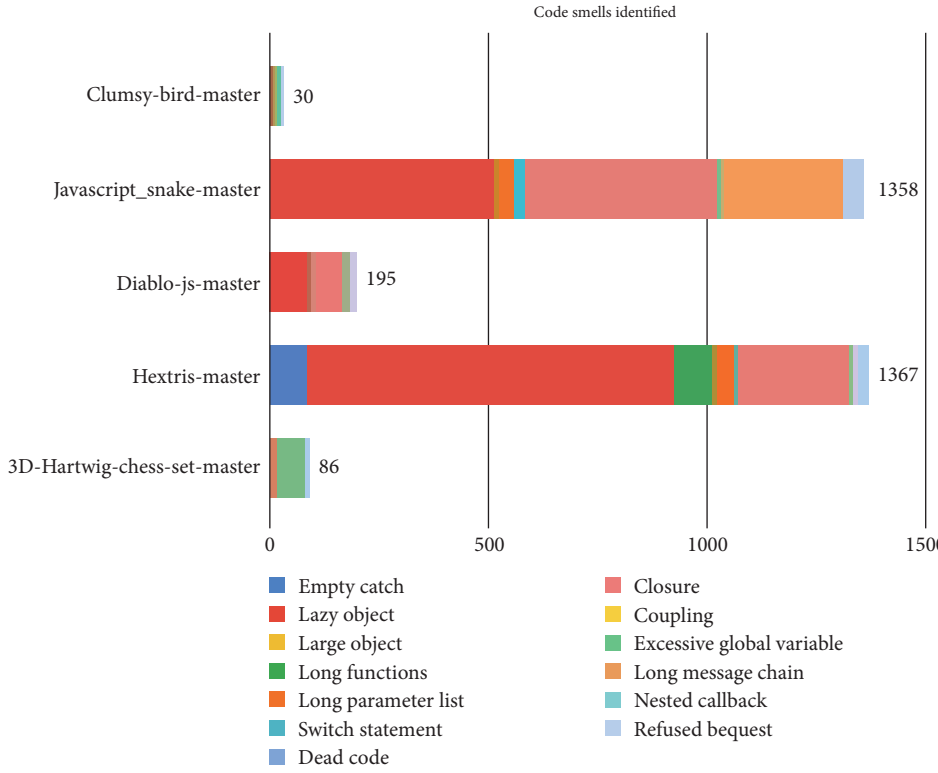

FIGURE 2: Code smells identified.

TABLE 5: Identified structural design patterns in web games.

File name	Adapter	Bridge	Composite	Decorator	Facade	Proxy
Clumsy-bird-master	2	1	0	0	8	1
Javascript snake-master	3	5	7	2	12	23
Diablo-js-master	0	19	21	5	12	9
Hextris-master	7	1	0	0	0	1
3D-Hartwig-chess-set-master	5	19	1	5	12	9

Similar to how our findings were represented in the preceding section, the graphics in Figure 3 provide a clear visual representation of the structural kind of design pattern. The occurrence of distinct structural types of design patterns concerning each project is indicated on the horizontal axis of the graph. In contrast, the detail of a selected project is mentioned on the vertical axis. The graphics indicate that a proxy type of structural design pattern exists in all projects, followed by a bridge design pattern. Whereas the projects contained the least number of decorator and adapter design patterns. The results pertaining to RQ2 clearly show that selected web games exhibit structural design trends. The distribution of these design patterns depicts that the bridge pattern occurs the most in selected games, followed by the proxy design pattern.

To investigate the types of co-occurrences of CS and DP in web games, we employed association rule mining. This technique helps us classify the dependencies among attributes. We used two standard metrics, confidence and support, to construct rules for each significant rule. The results of these co-occurrences are reported in Table 6. The table depicts the available co-occurrences of code smells and design patterns.

Based on the findings regarding our third research question (RQ3), it is evident that there exist co-occurrences of code smells and design patterns in selected web games. The distribution of these co-occurrences indicates that the long message chain-composite co-occurrence is the one most occurring in selected games followed by the large object-facade and refused bequest-adapter co-occurrence.

In order to address RQ4 and examine the influence of co-occurrence on internal quality parameters, we carried out our experimental study in two stages on balanced and unbalanced datasets. The original dataset, in which the values are dispersed at random across training and testing sets, was used for classification in the first phase. Then ML classifiers are used to analyze various performance metrics. The results of this process are listed in Table 7.

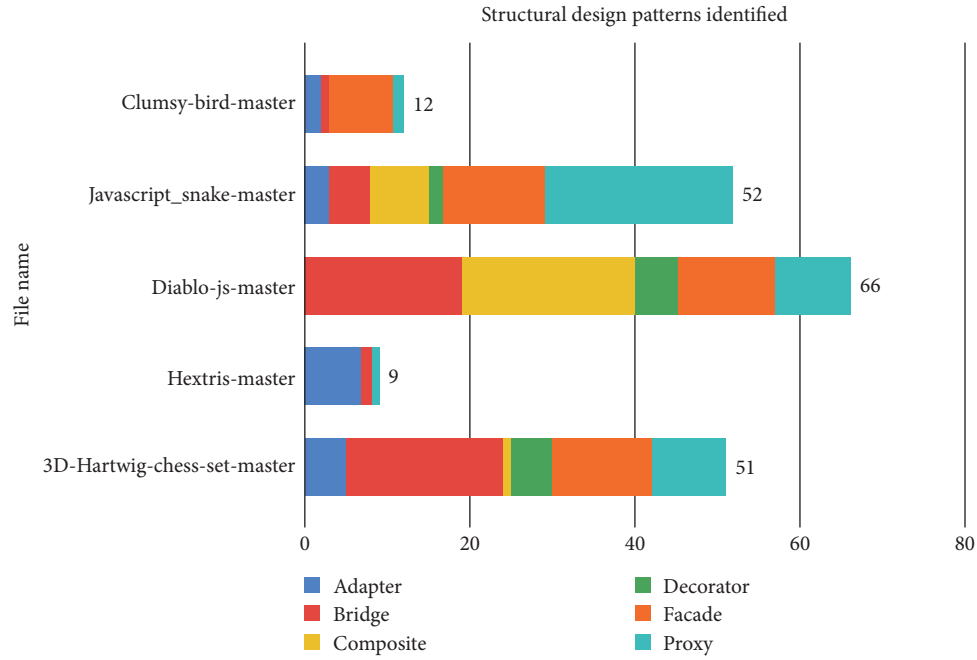

Figure 3: Design patterns identified.

Table 6: Identified co-occurrences in web games.

DP/CS	Closure	Coupling	Excessive global	Long message chain	Nested callback	Refused bequest	Empty catch	Lazy object	Large object	Long functions	Long parameter	Switch -state- ment	Dead code
Adapter	2	0	1	1	0	6	0	1	3	0	2	1	0
Bridge	2	0	1	1	1	2	4	1	0	0	2	0	0
Composite	0	0	3	7	0	0	0	4	1	2	1	0	0
Decorator	0	0	2	1	0	1	0	1	0	2	1	2	0
Facade	0	0	5	0	3	0	1	2	6	2	1	0	0
Proxy	2	0	1	2	0	2	0	1	0	0	2	0	0

Table 7: Performance metrics without data balancing.

Classifier	Accuracy	F-measure	Precision	Recall	kappa
Linear discriminant analysis	97.5453	0.976	0.977	0.975	0.7001
Quadratic discriminant analysis	96.3	0.968	0.977	0.959	0.644
Multinomial naive bayes	76.1183	0.83	0.96	0.728	0.171
Complement naive bayes	76.12	0.834	0.96	0.733	0.171
Gaussian naive bayes	95.432	0.96	0.976	0.944	0.604
Multilayer perceptrons	96.49	0.96	0.978	0.942	0.65
Support vector machine	97.93	0.97	0.98	0.960	0.74
Linear support vector machine	96.632	0.97	0.979	0.961	0.67
Decision tree	98.18	0.98	0.98	0.980	0.77
K-nearest neighbors	97.91	0.98	0.98	0.980	0.75
Random forest	98.99	0.99	0.99	0.980	0.87

With respect to performance metrics calculated in RQ4, we could say that the use of the random forest classification algorithm is most prevalent. This is because, in our empirical setup, this method performs best on the original dataset. For the second phase, we normalized the dataset, and then classification in the experiment was performed on that balanced dataset. By applying the ML classifiers, different performance measures are evaluated that are available in Table 8.

Table 8: Performance metrics with data balancing.

Classifier	Accuracy	F-measure	Precision	Recall	kappa
Linear discriminant analysis	97.81	0.996	0.977	1.0077	0.80
Quadratic discriminant analysis	96.9	0.968	0.977	0.9794	0.67
Multinomial naive bayes	76.1183	0.83	0.96	0.8473	0.171
Complement naive bayes	77.12	0.89	0.97	0.9038	0.19
Gaussian naive bayes	96.95	0.973	0.98	0.9829	0.69
Multilayer perceptrons	98.80	0.98	0.978	0.9910	0.839
Support vector machine	98.56	0.985	0.98	0.9951	0.80
Linear support vector machine	98.307	0.982	0.982	0.9910	0.74
Decision tree	98.33	0.983	0.98	0.9930	0.771
K-nearest neighbors	98.60	0.985	0.98	0.9951	0.80
Random forest	99.126	0.991	0.99	0.9960	0.88

Table 9: *p*-Values and impact.

Quality attribute	Metrics	*p*-Value (experiment 1)	*p*-Value (experiment 2)	Impact
Coupling	CBO	0.0125	0.0125	No impact
	FANIN	0.0615	0.0615	No impact
	FANOUT	0.0615	0.0615	No impact
Cohesion	LCOM	0.0437	0.0481	Impact
Inheritance	DIT	0.0315	0.0435	Impact
	NOC	0.625	0.625	No impact
Size	LOC	0.0474	0.0499	Impact
	CLOC	0.0325	0.0472	Impact
	NIV	0.0196	0.0318	Impact
	NIM	0.0625	0.0625	No impact
Complexity	CC	0.0713	0.0713	No impact
	WMC	0.0625	0.0625	No impact
	RFC	0.0625	0.0625	No impact
	Evg	0.0713	0.0713	No impact
	NPATH	1	1	No impact

Table 8 shows that normalizing the data improves the performance measures of different classifiers. The best classifier, according to the data, is random forest. After deciding on the optimal classifier, the final step is to test our assumptions to validate the proposed solution. To accomplish the final step, data is imported into the R studio and examined in relation to the test assumptions. The test's assumptions include: The *p*-value is calculated using the Wilcoxon signed-rank test. This explores the impact of co-occurrence on internal quality indicators by accepting or rejecting the null hypothesis. The *p*-values for both settings and their effect on quality criteria are reported in Table 9.

The results in the table suggest that co-occurrence has no impact on coupling and complexity of web games. In contrast, there is a significant impact of co-occurrence on cohesion, size, and inheritance. This means that the null hypothesis is rejected, and an alternate hypothesis is accepted. Our overall conclusion is that the co-occurrence of CS and DP has a significant impact on the internal quality attributes of the system.

5. Threats to Validity

To ensure the accuracy of the findings of our empirical investigation, we employed Wohlin et al. [44] criteria. Through this criteria, we identify and evaluate potential risks to ensure the quality of our findings. Every potential risk is examined, and countermeasures are implemented in this section.

Construct validity: This threat focuses on the relation between the theory and the observation. In our study, the main threat is using code smell and design pattern detection tools. Despite being built using various strategies, these tools provide a restricted count of code flaws and design patterns. To minimize this risk, manual detection was implemented; however, it's a time-consuming and error-prone process. To deal with this threat, we took help from the existing literature published in the domain of interest. The strategy to mitigate this threat could be the usage of different tools for cross comparison. This could not only enhance results but help

developers and researchers to evaluate the usage of tools before integrating that in their workflow.

Internal validity: An internal validity threat was identified in the quality metrics used as dependent variables to identify the impact of co-occurrences. Since the chosen metrics span different granularities (method, class, package, and project), future research should explore more refined and standardized metrics tailored to specific software contexts. However, we acknowledge that better metrics might be used as dependent variables for investigating the impact of co-occurrences.

External validity: These threats concern the generalization from our empirical study conclusions. The key hazard here is the type of software systems utilized in the dataset. Throughout our research, we have only opted an open-source projects that fall into several domain and size groups. Therefore, it is unclear if similar findings could possibly be expected if other game systems were used. Moreover, the usage of just one programing language poses an additional risk here. It could be, therefore, the case that game systems written in different languages might lead to different findings.

Conclusion validity: These risks deal with the treatment-outcome relationship. This possible risk in our empirical study originates from statistical evaluations used to find out whether the co-occurrence had a significant effect. The underlying assumptions of a statistical test determine which one should be used. Any deviation from the statistical test's presumptions could lead to incorrect conclusions. To verify the data normalcy assumption test, the Shapiro–Wilk test was employed. Because of the nonnormal distribution of the data, we used the non-parametric Wilcoxon signed-rank test. To reduce this risk, we explored changing the p-value using the threshold of 0.05, based on which we accept or reject the null hypothesis.

6. Conclusion

The purpose of this empirical study was to examine how co-occurring code smells and design patterns affect quality attributes that are internal in nature. We conducted a study with three stages for this purpose: data collection, data formulization, model training, and statistical testing. Through the identification, we discovered the imapct of code smells and design patterns co-occurring on internal quality metrics. A total of 11 structural measures and five internal quality criteria were analyzed. Findings reveal no impact on coupling and complexity but a significant influence on cohesion, size, and inheritance. Data balancing is crucial in assessing this impact, highlighting the beneficial role of design patterns in enhancing structural quality for developers. The research suggests game developers and project managers consider design patterns as good practices for maintaining code quality amid changes.

This empirical research has some implications for practice and research. First, using five real-world game systems, we provided empirical evidence that suggests game developers should consider the use of design patterns if they wish to maintain the internal quality of the code they produce, and in turn, slow down the deterioration of a game system subjected to many changes. Second, project managers in game development companies might consider the use of patterns as a good practice related to the actions their teams take to assure the quality of the games they produce. Finally, our work has some research implications. As generalizability of our findings is our most important concern, in the future, we would like to perform replications in similar but different contexts. We plan to replicate the conducted experiments on different emerging programing languages or different emerging domains such as AI-based applications, for example, explainable AI.

Data Availability Statement

The data that support the findings of this study are available from the corresponding author upon reasonable request.

Conflicts of Interest

The authors declare no conflicts of interest.

Author Contributions

Sania Imran: conceptualization, methodology, data collection, formal analysis, writing – original draft and revised draft. **Irum Inayat**: conceptualisation, supervision, methodology, writing, reviewing and editing original and revised draft. **Maya Daneva**: methodology, review and editing original version.

Funding

No funding was received for this manuscript.

Acknowledgments

No AI based tools were used for data collection, analysis, or interpretation. The content was carefully reviewed, edited, and verified by the authors to ensure accuracy and alignment with scientific content. The final manuscript reflects the authors interpretations and conclusions.

References

[1] A. Bandi, B. J. Williams, and E. B. Allen, "Empirical Evidence of Code Decay: A Systematic Mapping Study," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, (IEEE, 2013): 341–350.

[2] A. Uchôa, C. Barbosa, W. Oizumi, et al., "How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, (IEEE, 2020): 511–522.

[3] A. Ampatzoglou, A. Kritikos, E.-M. Arvanitou, A. Gortzis, F. Chatziasimidis, and I. Stamelos, "An Empirical Investigation on the Impact of Design Pattern Application on Computer Game Defects," in *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*, (ACM, 2011): 214–221.

[4] A. Ampatzoglou, G. Frantzeskou, and I. Stamelos, "A Methodology to Assess the Impact of Design Patterns on Software Quality," *Information and Software Technology* 54, no. 4 (2012): 331–346.

[5] F. Khomh and Y.-G. Guéhéneuc, "Design Patterns Impact on Software Quality: Where Are the Theories?" in *2018 IEEE 25th*

International Conference on Software Analysis, Evolution and Reengineering (SANER), (IEEE, 2018): 15–25.
[6] F. Khomh and Y.-G. Guéhéneuc, "Do Design Patterns Impact Software Quality Positively?" in *2008 12th European Conference on Software Maintenance and Reengineering*, (IEEE, 2008): 274–278.
[7] M. Vokac, "Defect Frequency and Design Patterns: An Empirical Study of Industrial Code," *IEEE Transactions on Software Engineering* 30, no. 12 (2004): 904–917.
[8] B. Walter and T. Alkhaeir, "The Relationship Between Design Patterns and Code Smells: An Exploratory Study," *Information and Software Technology* 74 (2016): 127–142.
[9] M. Tufano, F. Palomba, G. Bavota, et al., "When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)," *IEEE Transactions on Software Engineering* 43, no. 11 (2017): 1063–1088.
[10] T. Sharma, "Detecting and Managing Code Smells: Research and Practice," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeeding*, (IEEE, 2018): 546-547.
[11] F. Jolha, "Investigating the Impact of Refactoring Smelly Codes on Internal Quality Attributes: A Meta-Analytical Study," in *Proceedings of the 2024 13th International Conference on Networks, Communication and Computing*, (IEEE, 2024): 66–72.
[12] A. Santana, E. Figueiredo, J. A. Pereira, and A. Garcia, "An Exploratory Evaluation of Code Smell Agglomerations," *Software Quality Journal* 32, no. 4 (2024): 1375–1412.
[13] A. Nandini, R. Singh, and A. Rathee, "Code Smells and Refactoring: A Tertiary Systematic Literature Review," *International Journal of System of Systems Engineering* 14, no. 1 (2024): 83–143.
[14] B. Pietrzak and B. Walter, "Leveraging Code Smell Detection With Inter-Smell Relations," in *Extreme Programming and Agile Processes in Software Engineering*, (Springer, 2006): 75–84.
[15] B. L. Sousa, M. A. S. Bigonha, and K. A. M. Ferreira, "An Exploratory Study on Cooccurrence of Design Patterns and Bad Smells Using Software Metrics," *Software: Practice and Experience* 49, no. 7 (2019): 1079–1113.
[16] B. L. Sousa, M. A. Bigonha, and K. A. Ferreira, "A Systematic Literature Mapping on the Relationship Between Design Patterns and Bad Smells," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 10, (ACM, 2018): 1528-1528–1535-1535.
[17] A. Gupta, B. Suri, and S. Misra, "A Systematic Literature Review: Code Bad Smells in Java Source Code," in *Computational Science and Its Applications–ICCSA 2017*, (Springer, 2017): 665–682.
[18] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. do Nascimento, M. F. Freitas, and M. G. de Mendonça, "A Systematic Review on the Code Smell Effect," *Journal of Systems and Software* 144 (2018): 450–477.
[19] A. X. Barros and E. Adachi, "Bad Smells in Javascript—A Mapping Study," in *Anais do IX Workshop de Visualização, Evolução e Manutenção de Software*, (SBC, 2021): 1–5.
[20] N. Almashfi and L. Lu, "Code Smell Detection Tool for Java Script Programs," in *2020 5th International Conference on Computer and Communication Systems (ICCCS)*, (IEEE, 2020): 172–176.
[21] D. Johannes, F. Khomh, and G. Antoniol, "A Large-Scale Empirical Study of Code Smells in JavaScript Projects," *Software Quality Journal* 27, no. 3 (2019): 1271–1314.
[22] J. Holopainen and S. Björk, "Game Design Patterns," 2003, http://www.gents.it/FILES/ebooks/Game_Design_Patterns.pdf [Lecture Notes for GDC [Online]. Available:.
[23] Y. Guo, C. Seaman, N. Zazworka, and F. Shull, "Domain-Specific Tailoring of Code Smells: An Empirical Study," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2, (IEEE, 2010): 167–170.
[24] T. Alkhaeir and B. Walter, "The Effect of Code Smells on the Relationship Between Design Patterns and Defects," *IEEE Access* 9 (2021): 3360–3373.
[25] M. Folwer, *Refactoring: Improving the Design of Existing Programs* (Google Scholar Google Scholar Digital Library Digital Library, [Online]. Available:, 1999).
[26] M. Ramos, R. de Mello, and B. Fonseca, "On Transfer Learning in Code Smells Detection," 2022, https://easychair.org/publications/ EasyChair [Online]. Available:.
[27] J. P. dos Reis, F. Brito, G. de Figueiredo Abreu, and C. Anslow, "Code Smells Detection and Visualization: A Systematic Literature Review," *Archives of Computational Methods in Engineering* 29, no. 1 (2022): 47–94.
[28] A. Rio and F. Brito e Abreu, "Php Code Smells in Web Apps: Survival and Anomalies," arXiv preprint arXiv: 2101.00090 (2020).
[29] V. Agrahari and S. Chimalakonda, "An Exploratory Study of Code Smells in Web Games," arXiv preprint arXiv: 2002.05760 (2020).
[30] V. Khanve, "Are Existing Code Smells Relevant in Web Games? An Empirical Study," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (ACM, 2019): 1241–1243.
[31] A. Kaur, "A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes," *Archives of Computational Methods in Engineering* 27 (2020): 1267–1296.
[32] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations," *Journal of Systems and Software* 167 (2020): 110610.
[33] 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* 47, no. 1 (2021): 17–66.
[34] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension," in *2011 15Th European Conference on Software Maintenance and Reengineering*, (IEEE, 2011): 181–190.
[35] A. Yamashita and L. Moonen, "Do Developers Care about Code Smells? An Exploratory Survey," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, (IEEE, 2013): 242–251.
[36] W. Oizumi, A. Garcia, L. da Silva Sousa, B. Cafeo, and Y. Zhao, "Code Anomalies Flock Together: Exploring Code Anomaly Agglomerations for Locating Design Problems," in *Proceedings of the 38th International Conference on Software Engineering*, (ACM, 2016): 440–451.
[37] C. Politowski, F. Khomh, S. Romano, et al., "A Large Scale Empirical Study of the Impact of Spaghetti Code and Blob Anti- Patterns on Program Comprehension," *Information and Software Technology* 122 (2020): 106278.
[38] J. Martins, C. Bezerra, A. Uchôa, and A. Garcia, "Are Code Smell Co-Occurrences Harmful to Internal Quality Attributes?

A Mixed-Method Study," in *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, (ACM, 2020): 52–61.

[39] J. Martins, C. Bezerra, A. Uchôa, and A. Garcia, "How Do Code Smell Co-Occurrences Removal Impact Internal Quality Attributes? A Developers' Perspective," in *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, (ACM, 2021): 54–63.

[40] S. H. S. Almadi, D. Hooshyar, and R. B. Ahmad, "Bad Smells of Gang of Four Design Patterns: A Decade Systematic Literature Review," *Sustainability* 13, no. 18 (2021): 10256.

[41] B. L. Sousa, M. A. S. Bigonha, and K. A. M. Ferreira, "When Gof Design Patterns Occur With God Class and Long Method Bad Smells?—An Empirical Analysis," *INFOCOMP Journal of Computer Science* 17, no. 1 (2018): 11–22.

[42] F. Palomba, R. Oliveto, and A. De Lucia, "Investigating Code Smell Co-Occurrences Using Association Rule Learning: A Replicated Study," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, (IEEE, 2017): 8–13.

[43] V. R. B. G. Caldiera and H. D. Rombach, "The Goal Question Metric Approach," in *Encyclopedia of Software Engineering*, (The Mcgraw-Hill Companies, 1994): 528–532.

[44] C. Wohlin, E. Mendes, K. R. Felizardo, and M. Kalinowski, "Guidelines for the Search Strategy to Update Systematic Literature Reviews in Software Engineering," *Information and Software Technology* 127 (2020).

[45] R. Lincke, J. Lundberg, and W. Löwe, "Comparing Software Metrics Tools," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, (ACM, 2008): 131–142.

[46] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini, "On the Impact of Refactoring on the Relationship between Quality Attributes and Design Metrics," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, (IEEE, 2019): 1–11.

[47] A. M. Fard and A. Mesbah, "JSNOSE: Detecting JavaScript Code Smells," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, (IEEE, 2013): 116–125.

[48] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring," *IEEE Transactions on Software Engineering* 32, no. 11 (2006): 896–909.

[49] S. Tandon, V. Kumar, and V. B. Singh, "Study of Code Smells: A Review and Research Agenda," *International Journal of Mathematical, Engineering and Management Sciences* 9, no. 3 (2024): 472–498.

[50] P. S. Yadav, R. S. Rao, A. Mishra, and M. Gupta, "Machine Learning-Based Methods for Code Smell Detection: A Survey," *Applied Sciences* 14, no. 14 (2024): 6149.

[51] K. Alkharabsheh, S. Alawadi, V. R. Kebande, Y. Crespo, M. Fernández-Delgado, and J. A. Taboada, "A Comparison of Machine Learning Algorithms on Design Smell Detection Using Balanced and Imbalanced Dataset: A Study of God Class," *Information and Software Technology* 143 (2022): 106736.

[52] F. Li, K. Zou, J. W. Keung, X. Yu, S. Feng, and Y. Xiao, "On the Relative Value of Imbalanced Learning for Code Smell Detection," *Software: Practice and Experience* 53, no. 10 (2023): 1902–1927.

[53] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric Statistical Methods* (John Wiley & Sons, 2013).