www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 17(1), Jan-April 2026, 215-238 ISSN: 2007-1558. https://doi.org/10.61467/2007.1558.2025.v16i1.947

Systematic Review of Code Smell Patterns and Their Impact on Software Technical Debt

Claudia de las Mercedes Rodríguez Ponce, Rene Santaolaya Salgado, Blanca Dina Valenzuela Robles, Humberto Hernández García

Tecnológico Nacional de México/Centro Nacional de Investigación y Desarrollo Tecnológico
m24ce007@cenidet.tecnm.mx, rene.ss@cenidet.tecnm.mx, blanca.vr@cenidet.tecnm.mx, humberto.hg@cenidet.tecnm.mx

Abstract. The accumulation of technical debt in legacy software is primarily manifested through the recurring need to modify code in order to correct defects, adapt it to new requirements, or alter existing functionalities. This situation reduces the useful lifespan of software systems by increasing their fragility, as a consequence of the continuous interventions required for maintenance and correction. Code smells constitute one of the principal contributors to this problem in software development. Consequently, it is essential to address this issue through appropriate design and coding practices in order to enhance code quality, maintainability, and readability. The objective of the present research is to identify the relative frequency of practices employed by developers during the design and coding phases that lead to the generation of malformed code, thereby contributing to entropy and increased technical debt in legacy systems. The study focuses on evaluating factors related to structural design and programming practices, while also conducting statistical analyses to identify recurring patterns associated with code smells. The results provide a detailed understanding of the most common practices and their impact on software quality. Furthermore, they highlight relevant gaps in current software engineering research and development, suggesting potential directions for future investigations aimed at addressing these shortcomings.

Keywords: refactoring, code smell, smell code, technical debt, software entropy, design quality, programming design.

Article Info
Received February 28, 2025
Accepted March 4, 2025

1. Introduction

One of the most critical challenges in the field of software engineering arises when developers lack sufficient experience or skills in system construction, which frequently leads to the production of technically deficient software or components. This phenomenon is commonly referred to as technical debt.

Technical debt in legacy systems denotes the negative consequences of incorrect, suboptimal, or incomplete decisions made during the software development process. These consequences adversely affect the value of the final product and may include partial or complete system invalidation, increased delivery times, and additional costs. According to Lárez Mata (2020), technical debt management should be addressed from the earliest stages of development and continue throughout the system's lifecycle.

If left unmanaged, this problem can evolve into a situation in which the costs and risks associated with system maintenance exceed its benefits, potentially leading to what is often described as a "software crisis". In such circumstances, legacy systems become progressively more difficult to maintain and increasingly resistant to change. The present research refers to a potential crisis characterised by persistent and emerging challenges within the software industry, driven by rapid technological change, evolving development practices, and shifting user expectations.

The insufficient application of modular and object-oriented design principles frequently results in flawed system designs, which generate entropy or disorder within the software and contribute to the accumulation of technical debt. This, in turn, necessitates repeated code modifications to correct defects, increasing system fragility and shortening its useful lifespan. Although this situation does not constitute a crisis in the traditional sense, it can be understood as a call for innovative approaches and effective solutions to ensure sustainable success in software development.

Accordingly, the present study aims to identify the primary causes of technical debt in legacy systems and to propose a research direction focused on reducing technical debt through automated software refactoring processes.

2. Research materials and method

2.1 Research Objective

The overall objective of this research and development project in Software Engineering Science is to determine the relative recurrence of developer practices during the structural design and coding stages that lead to the emergence of code smells. The specific objectives are as follows:

1. Identify and disseminate poor practices in the structural design and coding stages of software systems.
2. Contribute to the development of higher-quality software systems, thereby supporting the advancement of best practices within the software development community.
3. Contribute to the scientific and technological progress of research in Software Engineering Science, with a focus on reducing the likelihood of technical debt accumulation.

2.2 Research Methodology

In this research project, the Kitchenham methodology was adopted, as it provides a rigorous and structured framework for conducting systematic literature reviews in the field of software engineering. Its primary objective is to collect, analyse, and synthesise existing evidence on a specific topic, ensuring that the resulting findings are transparent, reproducible, and relevant to the scientific community (Kitchenham, 2007). In this study, the methodology was applied to identify, analyse, quantify, and synthesise patterns of design and coding practices that affect structural quality in software development and that contribute to software entropy and the accumulation of technical debt.

2.2.1 Definition of Research Questions

Two research questions were formulated to address the main objective of the study:

Research Question 1: What are the main design and/or programming factors identified in the literature that generate code smells, thereby contributing to software entropy and technical debt?

Research Question 2: Which code smell factors that contribute to software entropy and technical debt remain under-researched and lack established solutions in the current literature?

2.2.2. Conducting the search

The proposed search string was designed to align with the research questions by incorporating key terms such as technical debt, code smell, and software entropy, together with factors related to design, quality, and refactoring. This strategy enabled the identification of the principal issues that generate code smells and contribute to software entropy and technical debt, while also allowing the exploration of factors that remain under-researched and for which no clear solutions have yet been proposed. The use of Boolean operators in combination with the TITLE-ABS-KEY specification was intended to ensure both accuracy and adequate coverage of relevant studies addressing structural quality and solution approaches in software design.

The search string used was:
“(TITLE-ABS-KEY ("technical debt" OR "code smell" OR "software entropy" OR "smell code") AND TITLE-ABS-KEY (quality AND factor OR software AND design OR program AND quality OR programming AND design) AND TITLE-ABS-KEY (design AND refactoring OR program AND refactoring OR program AND design AND refactoring))”.

2.2.3. Inclusion and Exclusion Criteria

Table 1. Inclusion and Exclusion Criteria

Inclusion criteria	Exclusion criteria
Publications by year: 2016-2024	Type of work: books, book chapters, theses, didactic material, reports, surveys, white papers, technical notes, short articles, opinions or discussion papers.
Conference Papers	Articles written in languages other than English
Primary and secondary studies	Tertiary studies

The study selection process was conducted in three stages. In the first stage, the publication date inclusion criterion was applied, restricting the timeframe to studies published between 2016 and 2024. The second stage involved the application of exclusion criteria using the filtering tools provided by each database. Subsequently, abstracts were reviewed by academic peers to assess their relevance. Finally, in the third stage, the full texts of the selected articles were examined, and the inclusion criteria were applied to identify studies suitable for detailed analysis.

The following evaluation criteria were considered to validate the relevance of the selected case studies:

- The article focuses on investigating factors associated with malformed code, including techniques, tools, standards, and best practices related to its detection and correction, as well as their impact on software entropy and technical debt in development projects.
- The research problem is clearly defined.
- The study follows a structured and well-grounded research process.
- The results are presented in a clear and detailed manner.
- The article has been published in a recognised journal, conference, or congress.
- The article has received citations from other authors.
- The article explicitly outlines future work or alternative research directions.

As a result of this process, 161 potentially relevant articles were initially identified. Of these, 43 articles were retrieved from IEEE Xplore, including 36 related to malformed code and 24 addressing technical debt. A further 118 articles were obtained from Scopus, of which 117 addressed malformed code, 59 focused on technical debt, and one addressed software entropy.

After completing the third stage of review and applying the evaluation criteria, a final set of 55 articles was selected, as presented in Table 2. These studies formed the basis for analysing factors associated with malformed code and design practice patterns that affect software structure, as well as for identifying areas where research gaps or the absence of proposed solutions appear to persist.

Table 2. Related Articles

No.	Reference	Article Name
1	(Xu et al., 2017)	A Log-linear Probabilistic Model for Prioritizing Extract Method Refactorings.
2	(Abdelaziz et al., 2018)	A Novel Approach for Improving the Quality of Software Code using Reverse Engineering.
3	(M.Sangeetha & P.Sengottuvelan, 2017)	A Perspective Approach for Refactoring Framework Using Monitor based Approach.
4	(Zhang et al., 2018)	A Preliminary Investigation of Self-Admitted Refactorings in Open Source Software.
5	(Mkaouer et al., 2017)	A robust multi-objective approach to balance severity and import of refactoring opportunities.
6	(Kaur et al., 2017)	A Support Vector Machine based Approach for Code Smell Detection.
7	(Singh et al., 2019)	A User Feedback Centric Approach for Detecting God Class Code Smell Using Frequent Usage Patterns.
8	(Akash et al., 2019)	An Approach of Extracting God Class Exploiting Both Structural and Semantic Similarity.

No.	Reference	Article Name
9	(Guggulothu & Moiz, 2019)	An Approach to Suggest Code Smell Order for Refactoring.
10	(Zhu et al., 2023)	An Efficient Design Smell Detection Approach with Inter-class Relation.
11	(Tufano et al., 2016)	An Empirical Investigation into the Nature of Test Smells.
12	(Mumtaz et al., 2018)	An empirical study to improve software security through the application of code refactoring.
13	(Mehta et al., 2018)	Analyzing Code Smell Removal Sequences for Enhanced Software Maintainability.
14	(Panigrahi et al., 2020)	Application of Naïve Bayes classifiers for refactoring prediction at the method level.
15	(Liu et al., 2018)	Are Smell-Based Metrics Actually Useful in Effort-Aware Structural Change-Proneness Predicton An Empirical Study.
16	(Sirikul & Soomlek, 2016)	Automated Detection of Code Smells Caused by Null Checking Conditions in Java Programs.
17	(Ubayawardana & Karunaratna, 2018)	Bug Prediction Model using Code Smells.
18	(Guggulothu & Moiz, 2020)	Code smell detection using multi-label classification approach
19	(Jaber, 2019)	Code Smells Analysis Mechanisms, Detection Issues, and Effect on Software Maintainability.
20	(Paulk, 2016)	Code smells incidence: does it depend on the application domain?
21	(Ganea et al., 2017)	Continuous quality assessment with inCode.
22	(Liu et al., 2021)	Deep Learning Based Code Smell Detection.
23	(Sharma, 2018)	Detecting and Managing Code Smells: Research and Practice.
24	(Kumar Das et al., 2019)	Detecting Code Smells using Deep Learning.
25	(Pecorelli et al., 2020)	Developer-Driven Code Smell Prioritization.
26	(Liu et al., 2016)	Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection.
27	(Rani & Chhabra, 2017a)	Evolution of Code Smells over Multiple Versions of Softwares: An Empirical Investigation.
28	(Lahti et al., 2021)	Experiences on Managing Technical Debt with Code Smells and AntiPatterns.
29	(Sae-Lim et al., 2017)	How Do Developers Select and Prioritize Code Smells? A Preliminary Study.
30	(Bibiano et al., 2020)	How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?
31	(Oliveira et al., 2016)	Identifying Code Smells with Collaborative Practices: A Controlled Experiment.
32	(Gradišnik & Heričko, 2018)	Impact of Code Smells on the Rate of Defects in Software: A Literature Review.
33	(Pantiuchina et al., 2018)	Improving Code: The (Mis)perception of Quality Metrics.
34	(Shen et al., 2020)	Improving Machine Learning-based Code Smell Detection via Hyper-parameter Optimization.
35	(Reeshti et al., 2019)	Measuring Code Smells and Anti-Patterns.
36	(Aras, 2022)	Metric and Rufe Based Automated Detection of Antipatterns in Object-Oriented Software Systems.
37	(Merzah & Selçuk, 2017)	Metric Based Detection of Refused Bequest Code Smell.
38	(M. M. Rahman et al., 2018)	MMRUC3: A recommendation approach of move method refactoring using coupling, cohesion, and contextual similarity to enhance software design.

No.	Reference	Article Name
39	(Ouni et al., 2017)	MORE: Amulti-objective refactoring recommendation approach to introducing design patterns and fixing code smells.
40	(Rani & Chhabra, 2017b)	Prioritization of Smelly Classes: A Two Phase Approach.
41	(M. Rahman et al., 2017)	Recommendation of Move Method Refactorings Using Coupling, Cohesion and Contextual Similarity.
42	(Nyamawe et al., 2018)	Recommending Refactoring Solutions based on Traceability and Code Metrics.
43	(Turkistani & Liu, 2019)	Reducing the Large Class Code Smell by Applying Design Patterns.
44	(Arif & Rana, 2020)	Refactoring of Code to Remove Technical Debt and Reduce Maintenance Effort.
45	(Peruma et al., 2022)	Refactoring Debt: Myth or Reality? An Exploratory Study on the Relationship Between Technical Debt and Refactoring.
46	(Gupta et al., 2023)	Resource Allocation Modeling Framework to Refactor Software Design Smells.
47	(Sangeetha & Sengottuvelan, 2017)	Systematic Exhortation of Code Smell Detection Using JSmell for Java Source Code.
48	(Mohan et al., 2016)	Technical debt reduction using search based automated refactoring.
49	(Fernandes et al., 2024)	The Impact of a Live Refactoring Environment on Software Development.
50	(Palomba et al., 2018)	The Scent of a Smell: An Extensive Comparison between Textual and Structural Smells.
51	(Palomba et al., 2019)	Toward a Smell-aware Bug Prediction Model.
52	(Kiss & Mihancea, 2018)	Towards Feature Envy Design Flaw Detection at Block Level.

2.2.4. Information extraction and analysis

Table 3 provides a detailed classification of the studies selected in this research, organised according to the code smell factors evaluated and the methods employed for their analysis. The corresponding columns indicate whether corrective actions were implemented to address the identified issues and whether such corrections were automated, manually performed, or merely proposed. The process adopted in each case is also described, including the specific tools, techniques, and methodological approaches applied. Finally, the results obtained following the application of these methods are reported, highlighting key metrics that are commonly used to indicate reductions in code smells, such as improvements in software maintainability and other relevant quality indicators.

Table 3. Classification of Selected Studies

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
1	A Log-linear Probabilistic Model for Prioritizing Extract Method Refactorings	Long Method	x		x		Code fragments were automatically identified and refactorings were prioritized according to	The results showed that the approach achieves an effective balance between accuracy (up to

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
							probabilities assigned by the model.	30.3%) and recall (up to 62.1%).
2	A Novel Approach for Improving the Quality of Software Code using Reverse Engineering	Duplicated Code, Long Method, Large Class	x		x		Reverse engineering technique was used.	This technique improved the code without altering its external behavior.
3	A Perspective Approach for Refactoring Framework Using Monitor Based Approach	Duplicated Code, Long Method, Large Class, Long Parameter List, Divergent Change	x		x		A framework integrating monitoring, detection and refactoring was used.	The InsRefactor framework detected 548 of 654 code smell and resolved 500, outperforming tools such as JDeodorant.
4	A Preliminary Investigation of Self-Admitted Refactorings in Open Source Software (SARs).	Duplicated Code, Long Method, Large Class, Cyclomatic Complexity, Unused Imports, Too Many Methods	x			x	The study investigated the impact of SARs on Fastjson and JUnit4 using the static analysis tool Programming Mistake Detector (PMD) and structural quality metrics.	The results showed that 70.25% of the SARs did not worsen in Fastjson, and 67.86% in JUnit4, showing variable effectiveness according to project and developers.
5	A robust multi-objective approach to balance severity and importance of refactoring opportunities	Blob, Feature Envy, Data Class, Spaghetti Code, Lazy Class, Long Parameter List	x		x		The NSGA-II model was used with variations in severity and importance to simulate uncertainties.	The algorithm improved the code smell in 8 projects and one industrial case, minimizing quality loss.
6	A Support Vector Machine based Approach for Code Smell Detection	God Class, Data Class, Feature Envy, Long Method	x		x		The support vector model (SVM) with polynomial kernel was used to detect 4 types of code smell in ArgoUML and Xerces.	SVM excelled in detecting God Class and Long Method, showing efficiency and robustness in complete systems and classes.
7	A User Feedback Centric Approach for Detecting and Mitigating God	God Class	x			x	Metrics and expert feedback were used to detect God Class and apply Extract	The approach detected 100% of God Class with an accuracy of 95% and an F-

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
	Class Code Smell Using Frequent Usage Patterns						Class as refactoring.	measure of 96.2% in MobileMedia and 98.6% in HealthWatcher.
8	An Approach of Extracting God Class Exploiting Both Structural and Semantic Similarity	God Class	x		x		The approach combined structural and semantic analysis to detect cohesive methods in God Class, using automatic metrics and clustering.	The approach improved cohesion in Xerces and GanttProject, increasing C3 and significantly reducing LCOM after refactoring.
9	An Approach to Suggest Code Smell Order for Refactoring	Long Method, Feature Envy, God Class, Data Class	x		x		Key metrics were selected with machine learning to detect critical codes and assess their impact on quality.	Following the refactoring order (Data Class → Feature Envy → Long Method → God Class) was shown to reduce coupling and complexity, improving cohesion and abstraction.
10	An Automated Code Smell and Anti-Pattern Detection Approach	Data Class, Brain Method	x		x		The process used the Y-CSD tool to analyze software projects using structural metrics.	The approach achieved 83.3% accuracy for Data Class and 63.6% accuracy for Brain Method, beating iPlasma and Essere in Data Class detection.
11	An Efficient Design Smell Detection Approach with Inter-class Relation	Broken Hierarchy, Insufficient Modularization, Deficient Encapsulation	x		x		The process combined structural and semantic analysis with UML and neural networks (BiLSTM and R-GCN).	The results showed that Broken Hierarchy and Insufficient Modularization were detected with 69% accuracy.
12	An Empirical Investigation into the Nature of Test Smells	God Class, Internal Duplication, Blob Operation, Data Class, Distorted	x		x		The study analyzed the introduction, longevity and association of code smell in testing and	The results showed that code smell appeared in the first commits with an 88% probability and had an 80%

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
		Hierarchy, Message Chain					production in 152 projects, using HistoryMiner and metric rules for detection.	chance of not being corrected after 1,000 days.
13	An empirical study to improve software security through the application of code refactoring	Feature Envy, Message Chain, Un-Encapsulated Data Class	x		x		Java security code smell was identified with inFusion, its impact was validated and automatically refactored with IntelliJ IDEA.	Code smell was eliminated, reducing risks and improving metrics significantly (p-value < 0.05).
14	Analyzing Code Smell Removal Sequences for Enhanced Software Maintainability	Long Method, God Class, Feature Envy	x		x		The study analyzed 16 Java projects with JDeodorant, detected malformed code and calculated maintainability metrics. Refactoring was prioritized using a custom index (MCI) based on maintainability index (MI) and relative logical complexity (RLC).	The sequence Long Method → Feature Envy → God Class (LM-FE-GC) generated the greatest improvement in maintainability, increasing MI and reducing relative logic complexity (RLC).
15	Application of Naïve Bayes classifiers for refactoring Prediction at the method level	Long Method, Duplicate Code	x		x		The study analyzed 5 Tera-PROMISE projects, calculated 103 metrics and selected 8 key characteristics with the Wilcoxon test. It used 3 Naïve Bayes classifiers.	The Bernoulli Naïve Bayes classifier (BNB) obtained the best accuracy (84.65%) and AUC (78.78%). Significant feature selection (FS) improved accuracy and reduced time.
16	Are Smell-Based Metrics Actually Useful in Effort-Aware Structural Change-Proneness Prediction? An Empirical Study	Large Class, Feature Envy, Shotgun Surgery, Message Chain, Divergent Change	x			x	The study analyzed 6 Java projects, detected code smell and structural metrics, and used logistic regression to evaluate the correlation	The results showed that code smells metrics outperform traditional metrics in predicting structural changes, and

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
							between malformed code and structural changes.	their combination further improves accuracy in most projects.
17	Automated Detection of Code Smells Caused by Null Checking Conditions in Java Programs	Duplicated Code	x		x		The process used regular expressions to automatically identify patterns of null value check conditions in 13 open source Java projects.	The RegEx approach detected malformed code more efficiently than AST, increasing the detection of candidate variables to 90.52%.
18	Bug Prediction Model using Code Smells	God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling, Message Chain					The study used malformed and traditional code metrics to predict bugs in 13 Java projects, applying classifiers such as Naive Bayes, Random Forest and Logistic Regression.	The model with ill-formed code metrics outperformed the one based on traditional metrics. Random Forest performed the best, with 100% accuracy.
19	Code smell detection using multi-label classification approach	Long Method, Feature Envy	x				Multi-label classification methods that consider correlations between code smell were applied.	The methods, which consider correlations between ill-formed code, achieved an average accuracy of over 95%. A positive correlation was identified between Long Method and Feature Envy.
20	Code Smells Analysis Mechanisms, Detection Issues, and Effect on Software Maintainability	Long Method, Large Class, Feature Envy, Duplicated Code, Shotgun Surgery, Message Chain	x		x		The paper reviewed approaches to detect malformed code, combining metrics, machine learning and optimization. Tools with dynamic thresholds were	The combined approaches improved detection accuracy over metric-based methods. Tools with dynamic thresholds were more effective on larger projects.

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
							used to identify Large Class and Long Method.	
21	Code Smells and Detection Techniques A Survey	Duplicated Code, Long Method, Large Class, Feature Envy, Shotgun Surgery, Data Class, Message Chains, Lazy Class	x		x		The study reviewed code smell design, its impact on software quality and detection and correction techniques, analyzing tools such as PMD, JDeodorant and DECOR.	Current tools detect malformed code with high accuracy, but face challenges in large-scale projects with less well-defined code smell, such as Message Chains.
22	Code smells incidence: does it depend on the application domain?	Feature Envy, Large Class, Long Method, Data Class, Duplicated Code, Primitive Obsession					The study analyzed 118 Java systems in 6 domains, detecting code smell and complexity metrics with JDeodorant and CodePro AnalytiX.	There were no significant relationships between complexity and malformed code, except in cases such as Method size and Data Clumps.
23	Continuous quality assessment with inCode	God Class, Data Class, Feature Envy, Duplicated Code	x		x		The inCode tool, integrated in Eclipse, evaluated software quality, detected code smell with predefined metrics and provided contextual refactoring recommendations.	Developers with inCode solved more design problems in less time, reducing severity by up to 76% and improving design quality versus the Eclipse tool.
24	Deep Learning Based Code Smell Detection	Feature Envy, Long Method, Large Class, Misplaced Class	x		x		The approach used deep neural networks to detect code smell in Java, automatically generating data sets and training type-specific classifiers.	The learning approach improved code smell detection accuracy: 27.4% in Feature Envy, 15.11% in Long Method, 4.73% in Large Class and 48.18% in Misplaced Class.

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
25	Detecting and Managing Code Smells: Research and Practice	Long Method, God Class, Feature Envy, Shotgun Surgery, Duplicated Code, Data Class	x		x		He used metrics and rule-based approaches to identify code smell in real systems and described strategies to avoid them through refactoring and automated analysis tools.	The study concluded that the Designite and JDeodorant tools are effective in detecting code smell such as Long Method and God Class, but have difficulties with more complex ones, such as Shotgun Surgery.
26	Detecting Code Smells using Deep Learning	Brain Class, Brain Method	x		x		The study used 30 Java projects to generate data sets with metric-based rules.	The model achieved an accuracy of 97%-99% in Brain Class and 94%-95% in Brain Method.
27	Developer-Driven Code Smell Prioritization	God Class, Complex Class, Spaghetti Code, Shotgun Surgery					The study collected data from 9 open source projects, identifying code smell with tools such as DECOR.	The model with Random Forest achieved an F-Measure" of 72%-85%. Spaghetti Code obtained the highest accuracy.
28	Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection	Long Method, Feature Envy, Duplicated Code, Inappropriate Intimacy	x		x		The study proposed an automatic code smell detection threshold adjustment approach based on feedback from engineers, validating the results manually.	The approach improved accuracy over methods with static thresholds, reducing the gap between actual and objective accuracy by 80%.
29	Evolution of Code Smells over Multiple Versions of Softwares: An Empirical Investigation	Feature Envy, God Class, Long Method, Type Checking, Dead Code, Long Parameter List					The study analyzed the evolution of 3 projects ("Junit", "GCViewer", "Gitblit") using automated tools to detect code smell and design violations in 4 versions of each one.	The results showed that God Class and Feature Envy had the greatest impact, while Type Checking contributed little. Gitblit showed better quality than Junit and GCViewer.

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
30	Experiences on Managing Technical Debt with Code Smells and AntiPatterns	Long Method, Large Class, Shotgun Surgery, Divergent Change, Feature Envy, Primitive Obsession, Global Data	x		x		The process combined static analysis ("CodeMR", "IntelliJ IDEA") with manual inspections to detect code smell through metrics such as complexity, cohesion and coupling and to identify antipatterns from design problems.	2,690 lines of code smell were eliminated, critical classes were identified and prioritized for refactoring, and developers' understanding of the relationship between antipatterns and technical debt was improved.
31	Feature Envy Detection based on Bi-LSTM with Self-Attention Mechanism	Feature Envy					The study approached Feature Envy detection as a deep learning problem, combining textual information (names) and structural metrics (distances) to train a self-attentive Bi-LSTM network on 7 Java projects.	The approach outperformed JDeodorant, JMove and CNN, achieving an F1-score of 55.97% on Feature Envy and 82.29% accuracy. Self-attention and Bi-LSTM improved detection accuracy and reliability.
32	How Do Developers Select and Prioritize Code Smells? A Preliminary Study	Blob Class, Data Class, God Class, Schizophrenic Class					The study evaluated how developers selected and prioritized code smell in prefactoring, using data from JabRef and code smell lists detected by inFusion.	Developers prioritized code smell by relevance and module. The tools generated false positives, highlighting the need for improved filtering and prioritization methods.
33	How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?	Feature Envy, God Class	x		x		The study analyzed 5 Java projects and 353 incomplete refactorings, evaluating their impact on code quality.	The analysis revealed that 58% of the refactorings did not affect quality, 22% worsened it and 13% improved it.

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
34	Identifying Code Smells with Collaborative Practices: A Controlled Experiment	Long Method, Unused Field, Duplicated Code, Data Class, Feature Envy, Lazy Class, Intensive Coupling					A controlled experiment was designed with 28 novice developers divided into groups using 3 approaches: programming only, Pair Programming (PP) and Group Programming (CDR).	Collaborative practices improved inter-class malformed code detection by 40%. Pairs avoided more false positives, but large groups faced convergence and timing problems.
35	Impact of Code Smells on the Rate of Defects in Software: A Literature Review	God Class, Feature Envy, Shotgun Surgery, Data Class, Brain Class, Dispersed Coupling, Lazy Class					A systematic review was conducted where the relationship between code smell and software defects was analyzed, identifying 6 studies in systems such as Hadoop and Eclipse.	The results showed that there was a weak correlation between malformed code and defects. God Class and Dispersed Coupling showed positive correlation in some cases.
36	Improving Code: The (Mis)perception of Quality Metrics	God Class, Feature Envy, Shotgun Surgery, Long Method, Data Class, Dispersed Coupling, Lazy Class					1,282 registrations of changes in Java projects were analyzed on GitHub, comparing before and after metrics to validate improvements in quality that were stated.	In most cases, quality metrics did not reflect the improvements perceived by developers: only 26%-40% of classes improved in cohesion, and complexity increased.
37	Improving Machine Learning-based Code Smell Detection via Hyper-parameter Optimization	Data Class, Feature Envy					Eight Java projects were used, processing data with feature extraction tools. Machine learning models were trained and validated with hyperparameter optimization.	Hyperparameter optimization improved model performance. Data with lower code smell ratio allowed for more accurate detection.
38	Measuring Code Smells and Anti-Patterns	God Class, Duplicated Code, Feature Envy, Long					4,443 lines of code in 5 modules were analyzed,	The study found that some malformed codes were eliminated,

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
		Method, Type Checking					detecting code smell with JDeodorant and refactoring them with JSpIRIT.	others persisted or increased, with reductions of up to 60% in certain modules.
39	Metric and Rufe Based Automated Detection of Antipatterns in Object-Oriented Software Systems	God Class, Swiss Army Knife, Lava Flow					The study developed an automated detection system based on metrics and rules to identify antipatterns in object-oriented code.	The system achieved moderate accuracies: Good Class (84.6%), Swiss Army Knife (72.7%), Lava Flow (66.6%), with an average of 74.6%.
40	Metric Based Detection of Refused Bequest Code Smell	Refused Bequest					Metrics-based technique was used to identify "Refused Bequest" in object-oriented systems, analyzing Java projects to define thresholds and apply them to specific class hierarchies.	The technique correctly identified "Refused Bequest" in test hierarchies, with effective metrics to detect problematic classes, highlighting the ASM metric as a safe indicator.
41	MMRUC3: A recommendation approach of move method refactoring using coupling, cohesion, and contextual similarity to enhance software design	Feature Envy	x		x		Codes were analyzed to extract calls, attributes and context, calculating similarity between methods and classes with coupling, cohesion and context metrics.	The MMRUC3 approach achieved 18.91% accuracy and 69.91% recall, outperforming tools such as JDeodorant in accuracy and coverage.
42	MORE: Amulti-objective refactoring recommendation approach to introducing design patterns and fixing code smells	God Class, Feature Envy, Spaghetti Code, Shotgun Surgery, Long Parameter List, Lazy Class	x		x		The code was analyzed with tools to detect code smell and design patterns, generating optimal refactorings that correct defects, introduce patterns and improve quality metrics.	84% of the code smell was corrected, introducing an average of 6 design patterns. Significant improvements in cohesion, coupling and understandability.

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
43	Prioritization of Smelly Classes: A Two Phase Approach	God Class, Feature Envy, Duplicate Code, Long Method, Type Checking	x		x		The refactoring was carried out using automated methods provided by "JDeodorant", such as Extract Method and Extract Class.	Refactoring with prioritization reduced problems to 21% after 10 iterations, compared to the remaining 50% with random refactoring.
44	Recommendation of Move Method Refactorings Using Coupling, Cohesion and Contextual Similarity	Feature Envy	x		x		The approach analyzes the code to extract classes, methods and attributes, calculating similarities between a target method and other classes using metrics and contextual similarity (tf-idf and cosine).	The approach was evaluated on 3 open source projects, achieving 78% accuracy, significantly outperforming JDeodorant, which scored 45% and 47%, respectively.
45	Recommending Refactoring Solutions based on Traceability and Code Metrics	Feature Envy, God Class	x		x		Metrics such as traceability entropy and Entity Placement (EP) were calculated to evaluate design and traceability. The suggested solutions (Move Method and Extract Class) were virtually evaluated.	The solutions reduced entropy in 81% of cases, decreasing it by 6%, while traditional tools increased it by 3.6%.
46	Reducing the Large Class Code Smell by Applying Design Patterns	Large Class	x		x		A refactoring process was carried out to mitigate the code smell Large Class by applying design patterns.	The introduction of design patterns improved code maintainability, reusability and flexibility. Quality metrics showed improvements in more than 70% of the evaluated cases.

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
47	Refactoring of Code to Remove Technical Debt and Reduce Maintenance Effort	Redundant Code, Large Class, Long Method	x		x		Five open source projects were selected and code smell was identified using the ReSharper tool.	The elimination of code smell improved project efficiency by 7%. In projects with clean code, less effort was required to add new features and fix bugs.
48	Refactoring Debt: Myth or Reality? An Exploratory Study on the Relationship Between Technical Debt and Refactoring	Feature Envy, Large Class, Redundant Code, Long Method					77 open source Java projects were analyzed using SmartSHARK to extract data about refactorings. Operations such as Extract Method, Rename Variable and Change Variable Type were applied.	55.37% of the commits removing technical debt included refactoring operations. The most common refactorings were Extract Method (22.32%) and Rename Variable (10.89%).
49	Resource Allocation Modeling Framework to Refactor Software Design Smells	God Class, Feature Envy, Large Class, Duplicate Code, Spaghetti Code	x		x		An NHPP-based framework was developed to calculate code smell eliminated with specific resources, validated with data from the Azureus software (Vuze) classifying 33 defects in 7 categories.	During the first phase, 84.16% of the code smell defects were eliminated using 35,000 resource units. In the second phase, 50.35% of the remaining defects were removed with 25,000 resource units.
50	Systematic Exhortation of Code Smell Detection Using JSmell for Java Source Code	Data Class, Message Chain, Primitive Obsession, Speculative Generality, Duplicate Code	x		x		The analysis consisted of two phases: JSmell used ANTLR to extract data and then code smell was identified. Refactorization was suggested.	Automated refactoring reduced code comprehension time by 30% and the time to add new functionality was cut in half.
51	Technical debt reduction using search based automated refactoring	God Class, Feature Envy, Long Method	x		x		Evaluation functions were designed based on software metrics, optimized with	The results showed that the coupling functions showed significant improvements

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
							techniques such as Hill Climbing and Simulated Annealing, and evaluated in 6 Java programs.	while the abstraction and inheritance functions had little impact.
52	The Impact of a Live Refactoring Environment on Software Development	Long Method, Duplicate Code, God Class	x		x		The study developed an add-on for IntelliJ IDEA that analyzes code quality in real time, suggesting refactorings based on metrics.	The study showed that a real-time refactoring environment accelerates problem detection and correction, improving software quality and promoting best practices.
53	The Scent of a Smell: An Extensive Comparison between Textual and Structural Smells	God Class, Feature Envy, Long Method, Misplaced Class, Promiscuous Package	x		x		Twenty open source projects were analyzed to detect textual and structural code smell. A qualitative analysis with 19 developers and 5 experts evaluated the perception to refactor the detected codes.	Textual code smells were easier to identify and refactor than structural ones. Textual codes decreased over time due to maintenance activities, while structural codes increased.
54	Toward a Smell-aware Bug Prediction Model	God Class, Data Class, Brain Method, Shotgun Surgery, Message Chain, Dispersed Coupling					The study evaluated an error prediction model that included code smell as a predictor, analyzing 34 versions of 11 projects with data from PROMISE and JCodeOdor.	The model increased F-Measure by 47% over models based on structural metrics alone. Combining metrics, processes and ill-formed code, it achieved 13% more accuracy than state-of-the-art models.

No.	Article Name	Code Smell Factors	Correction Methods		Automatic Corrections		Process Used	Results
			Yes	No	Yes	No		
55	Towards Feature Envy Design Flaw Detection at Block Level	Feature Envy	x		x		The approach used an Eclipse plug-in called Method Defragmenter to identify Feature Envy fragments within methods.	The approach improved accuracy in detecting Feature Envy at the block level versus IPLASMA. It was more relevant in 9 out of 26 cases for complex methods, but less accurate in 10 cases because of excessive splits.

In some cases, the articles do not report corrective actions because their primary focus is on the detection and analysis of code smells rather than on the application of refactoring techniques. The absence of an "X" in the columns corresponding to method, automatic, or non-automatic correction may be due to the fact that the study only identifies problems without implementing corrections, does not specify the type of correction applied, or concentrates on the prediction and classification of code smells rather than on their modification. In addition, several of the selected articles correspond to systematic literature reviews and therefore do not involve direct intervention in software artefacts.

3. Results

In response to the research questions posed, the results obtained from the analysis of the 55 papers selected for the Systematic Literature Review are presented below.

Q1: What are the main design and/or programming factors identified in the literature that generate code smells, contributing to software entropy and technical debt?

A detailed analysis of the code smell factors that influence the generation of software entropy and technical debt was conducted. Table 4 summarises the processed data, in which 25 design-related practices were identified. These practices affect system architecture, modularity, and cohesion. In addition, 17 programming practices were identified, which influence code clarity and efficiency.

Table 4. Recurrence of code smell

No.	Code smell factors	Recurrence	Description	Classification	
				Design	Programming
1	Feature Envy	33	Methods that frequently access data of another class.	x	
2	Long Method	25	Extensive methods that make understanding difficult.		x
3	God Class	25	Classes that centralize too much logic.	x	
4	Data Class	17	Classes that only contain data and lack behavior.		x
5	Duplicated Code	15	Repeated code.		x
6	Large Class	13	Classes with too many methods or responsibilities.	x	
7	Shotgun Surgery	11	Changes in functionality require modifying many classes.	x	
8	Message Chain	8	Long sequences of calls between objects.	x	
9	Lazy Class	6	Classes with limited functionality.		x
10	Spaghetti Code	4	Code is disorganized and difficult to follow.	x	
11	Long Parameter List	4	Methods with too many parameters.		x

No.	Code smell factors	Recurrence	Description	Classification	
				Design	Programming
12	Brain Method	4	Methods that concentrate too much complex logic.		x
13	Dispersed Coupling	4	Units distributed in multiple locations.	x	
14	Divergent Change	3	Classes that need to be modified for multiple reasons.	x	
15	Primitive Obsession	3	Excessive use of primitive types to represent complex concepts.		x
16	Type Checking	3	Explicit type checks instead of using polymorphism.		x
17	Blob Operation	3	Methods that concentrate too much logic.	x	
18	Misplaced Class	2	Classes placed in incorrect packages.	x	
19	Brain Class	2	Classes that concentrate too much system logic.	x	
20	Redundant Code	2	Code that performs duplicate tasks.		x
21	Cyclomatic Complexity	1	Complex logic flow.		x
22	Unused Imports	1	Imports not used in the code.		x
23	Too Many Methods	1	Classes with too many methods		x
24	Broken Hierarchy	1	Poorly designed inheritance hierarchies.	x	
25	Insufficient Modularization	1	Lack of adequate separation in modules.	x	
26	Deficient Encapsulation	1	Internal details of a class unnecessarily exposed.	x	
27	Internal Duplication	1	Duplicate code within the same class.		x
28	Un-Encapsulated	1	Unprotected data within a class.	x	
29	Promiscuous Package	1	Packages with diverse and unconnected classes.	x	
30	Complex Class	1	Classes with overly intricate logic.	x	
31	Overcomplicated Expressions	1	Unnecessarily complex expressions.		x
32	Global Data	1	Globally accessible data.		x
33	Schizophrenic Class	1	Classes with multiple unrelated responsibilities	x	
34	Dead Code	1	Code not used.		x
35	Refused Bequest	1	Subclasses that do not use legacy functionality.	x	
36	Lava Flow	1	Obsolete code that remains in the system.		x
37	Inappropriate Intimacy	1	Classes that excessively access internal details of other classes.	x	
38	Unused Field	1	Fields declared but never used.		x
39	Distorted Hierarchy	1	Poorly designed inheritances.	x	
40	Intensive Coupling	1	Class that depends too much on the methods of another class.	x	
41	Swiss Army Knife	1	Classes with too many general functionalities.	x	
42	Speculative Generality	1	Code written for unconfirmed future use.	x	

Q2: What are the code smell factors that contribute to software entropy and technical debt but have received limited research attention and for which no solutions have yet been proposed?

Factors such as Inappropriate Intimacy, Blob Operation, Unused Field, Dead Code, Lava Flow, Broken Hierarchy, Schizophrenic Class, Promiscuous Package, Complex Class, Overcomplicated Expressions, Global Data, Distorted Hierarchy, Intensive Coupling, and Swiss Army Knife show a low incidence across the analysed studies, reflecting limited research attention. In addition, other factors, including Cyclomatic Complexity, Deficient Encapsulation, Internal Duplication, and Speculative Generality, are only superficially addressed in the literature. This leaves a significant gap in the identification of effective methods to mitigate their impact on software entropy and technical debt. Overall, this analysis highlights the need for more in-depth and focused studies to develop strategies and solutions in these critical areas.

In summary, Figures 1 and 2 present the analysis conducted using Pareto charts, which were used to quantify the relative frequencies of these factors.

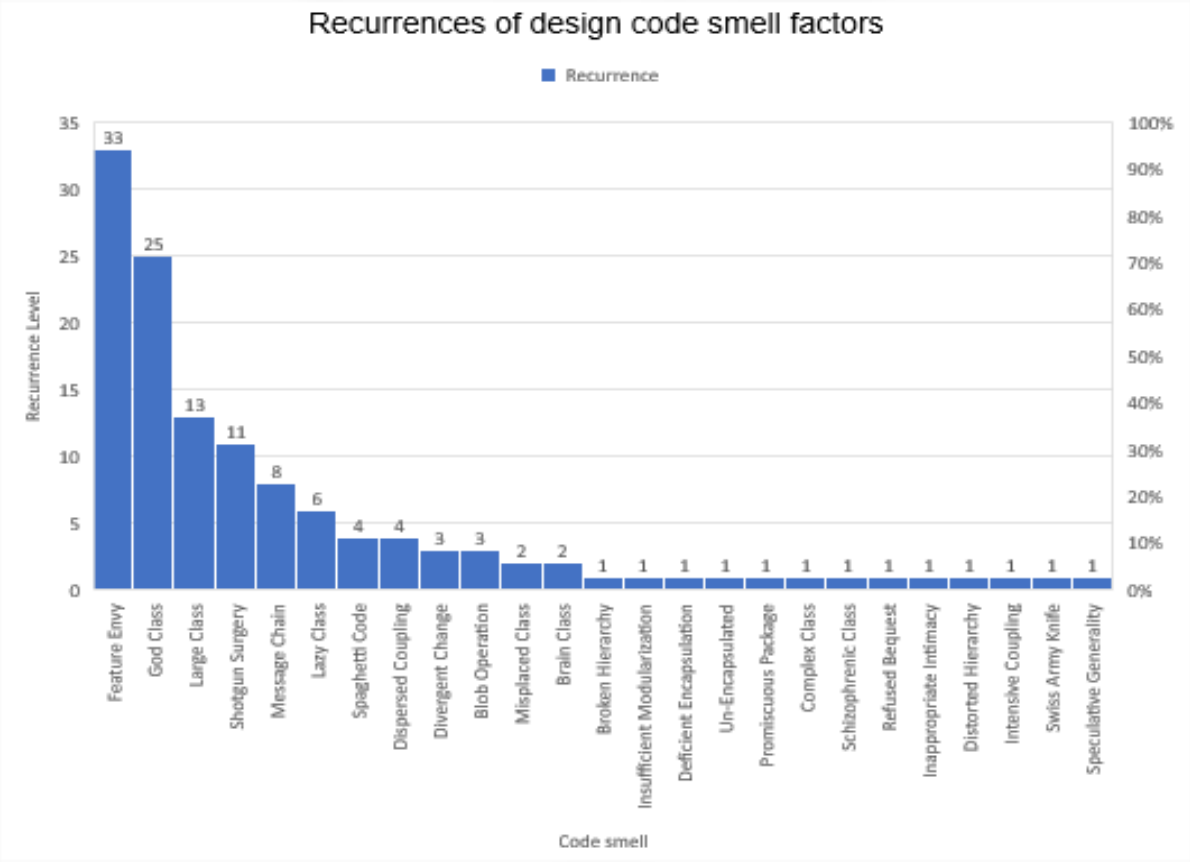

Figure 1. Recurrence of design factors

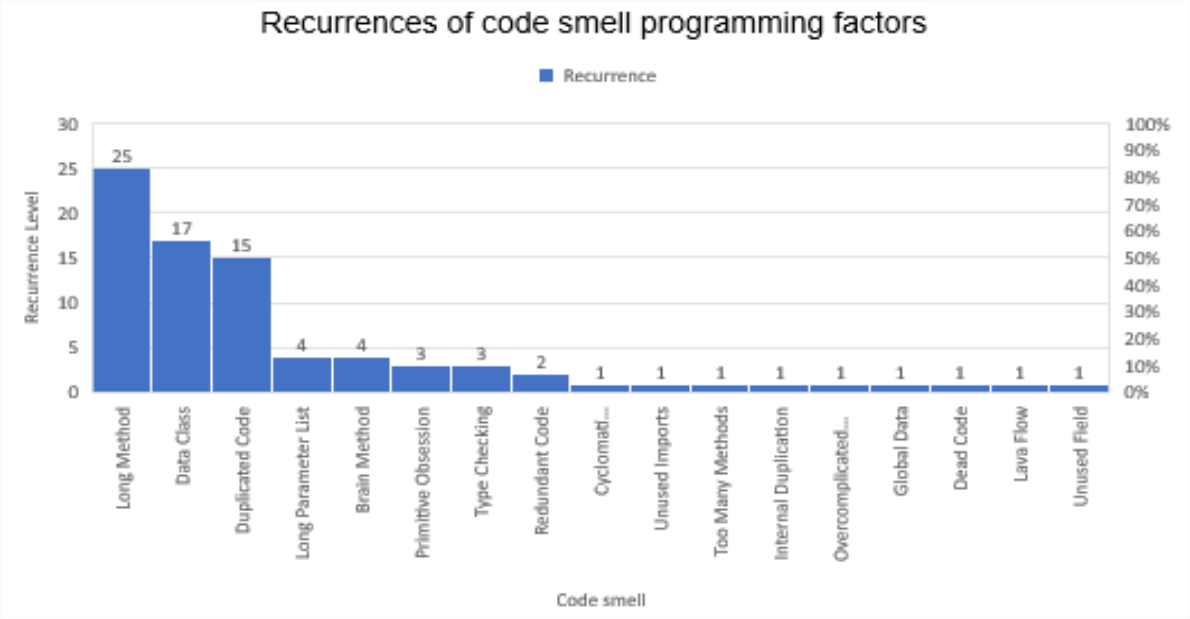

Figure 2. Recurrence of programming factors

The most frequently reported factors were Feature Envy, Long Method, God Class, Large Class, and Data Class. In contrast, those that have received comparatively less attention in the literature, and therefore represent potential avenues for future research, include Blob Operation, Brain Class, Brain Method, Broken Hierarchy, Deficient Encapsulation, Misplaced Class, and Dead Code. In addition, two factors with particularly low incidence, namely Refused Bequest and Lava Flow, remain largely unsupported by empirically validated or broadly generalisable methodologies.

4. Conclusions

The research presented in this paper has not only identified recurrent patterns in software quality factors, but has also highlighted significant gaps in the existing literature. This analysis provides evidence that can be used to prioritise efforts toward the development of automated tools and novel approaches aimed at addressing less explored code smells. In doing so, it may contribute to more effective control of the evolution of existing software systems by mitigating technical debt and structurally improving software in future projects, thereby reducing the need for continuous code manipulation and potentially extending system longevity.

5. References

Abdelaziz, T. M., Elghadhafi, H. A., & Maatuk, A. M. (2018, June 19). A novel approach for improving the quality of software code using reverse engineering. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3234698.3234729

Akash, P. S., Sadiq, A. Z., & Kabir, A. (2019). An approach of extracting God class exploiting both structural and semantic similarity. *ENASE 2019 - Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, 427–433. https://doi.org/10.5220/0007743804270433

Aras, R. (2022). Decision Support System (DSS) dengan Berorientasi -Solver. teknik, 2(1), 58-63. https://doi.org/10.55606/teknik.v2i1.917

Arif, A., & Rana, Z. A. (2020, December 16). Refactoring of Code to Remove Technical Debt and Reduce Maintenance Effort. *2020 14th International Conference on Open Source Systems and Technologies, ICOSST 2020 - Proceedings*. https://doi.org/10.1109/ICOSST51357.2020.9332917

Bibiano, A. C., Soares, V., Coutinho, D., Fernandes, E., Correia, J. L., Santos, K., Oliveira, A., Garcia, A., Gheyi, R., Fonseca, B., Ribeiro, M., Barbosa, C., & Oliveira, D. (2020). How does incomplete composite refactoring affect internal quality attributes? *IEEE International Conference on Program Comprehension*, 149–159. https://doi.org/10.1145/3387904.3389264

Fernandes, S., Aguiar, A., & Restivo, A. (2024). The Impact of a Live Refactoring Environment on Software Development. *Proceedings - International Conference on Software Engineering*, 337–338. https://doi.org/10.1145/3639478.3643100

Ganea, G., Verebi, I., & Marinescu, R. (2017). Continuous quality assessment with inCode. *Science of Computer Programming*, *134*, 19–36. https://doi.org/10.1016/j.scico.2015.02.007

Gradišnik, M., & Heričko, M. (2018, August 27). *Impact of Code Smells on the Rate of Defects in Software: A Literature Review* . http://www.webofknowledge.com/

Guggulothu, T., & Moiz, S. A. (2019). An Approach to Suggest Code Smell Order for Refactoring. *Communications in Computer and Information Science*, *985*, 250–260. https://doi.org/10.1007/978-981-13-8300-7_21

Guggulothu, T., & Moiz, S. A. (2020). Code smell detection using multi-label classification approach. *Software Quality Journal*, *28*(3), 1063–1086. https://doi.org/10.1007/s11219-020-09498-y

Gupta, P., Anand, A., & Mellal, M. A. (2023). Resource Allocation Modeling Framework to Refactor Software Design Smells. *International Journal of Mathematical, Engineering and Management Sciences*, *8*(2), 213–229. https://doi.org/10.33889/ijmems.2023.8.2.013

Jaber, K. Mohammad. (2019). *Code Smells Analysis Mechanisms, Detection Issues, and Effect on Software Maintainability*.

Kaur, A., Jain, S., & Goel, S. (2017). A Support Vector Machine Based Approach for Code Smell Detection. *Proceedings - 2017 International Conference on Machine Learning and Data Science, MLDS 2017*, *2018-January*, 9–14. https://doi.org/10.1109/MLDS.2017.8

Kiss, A., & Mihancea, P. F. (2018). Towards feature envy design flaw detection at block level. *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, 544–548. https://doi.org/10.1109/ICSME.2018.00064

Kitchenham, B. (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering*. https://www.researchgate.net/publication/302924724

Kumar Das, A., Yadav, S., & Dhal, S. (2019). *Detecting Code Smells using Deep Learning*. 1362.

Lahti, J. R., Tuovinen, A. P., & Mikkonen, T. (2021). Experiences on Managing Technical Debt with Code Smells and AntiPatterns. *Proceedings - 2021 IEEE/ACM International Conference on Technical Debt, TechDebt 2021*, 36–44. https://doi.org/10.1109/TechDebt52882.2021.00013

Lárez Mata, J. J. (2020). *Calidad de Software - Deuda Técnica*. https://saber.ucab.edu.ve/handle/123456789/647

Liu, H., Jin, J., Xu, Z., Zou, Y., Bu, Y., & Zhang, L. (2021). Deep learning based code smell detection. *IEEE Transactions on Software Engineering*, *47*(9), 1811–1837. https://doi.org/10.1109/TSE.2019.2936376

Liu, H., Liu, Q., Niu, Z., & Liu, Y. (2016). Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection. *IEEE Transactions on Software Engineering*, *42*(6), 544–558. https://doi.org/10.1109/TSE.2015.2503740

Liu, H., Yu, Y., Li, B., Yang, Y., & Jia, R. (2018). Are Smell-Based Metrics Actually Useful in Effort-Aware Structural Change-Proneness Prediction? An Empirical Study. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, *2018-December*, 315–324. https://doi.org/10.1109/APSEC.2018.00046

Mehta, Y., Singh, P., & Sureka, A. (2018). *Analyzing Code Smell Removal Sequences for Enhanced Software Maintainability*.

Merzah, B. M., & Selçuk, Y. E. (2017). *Metric Based Detection of Refused Bequest Code Smell*. 121.

Mkaouer, M. W., Kessentini, M., Cinnéide, M., Hayashi, S., & Deb, K. (2017). A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, *22*(2), 894–927. https://doi.org/10.1007/s10664-016-9426-8

Mohan, M., Greer, D., & McMullan, P. (2016). Technical debt reduction using search based automated refactoring. *Journal of Systems and Software*, *120*, 183–194. https://doi.org/10.1016/j.jss.2016.05.019

M.Sangeetha, & P.Sengottuvelan. (2017). *A Perspective Approach for Refactoring Framework Using Monitor based Approach*.

Mumtaz, H., Alshayeb, M., Mahmood, S., & Niazi, M. (2018). An empirical study to improve software security through the application of code refactoring. *Information and Software Technology*, *96*, 112–125. https://doi.org/10.1016/j.infsof.2017.11.010

Nyamawe, A. S., Liu, H., Niu, Z., Wang, W., & Niu, N. (2018). Recommending refactoring solutions based on traceability and code metrics. *IEEE Access*, *6*, 49460–49475. https://doi.org/10.1109/ACCESS.2018.2868990

Oliveira, R., Estácio, B., Garcia, A., Marczak, S., Prikladnicki, R., Kalinowski, M., & Lucena, C. (2016). Identifying code smells with collaborative practices: A controlled experiment. *Proceedings - 2016 10th Brazilian*

Symposium on Components, Architectures and Reuse Software, SBCARS 2016, 61–70. https://doi.org/10.1109/SBCARS.2016.18

Ouni, A., Kessentini, M., Ó Cinnéide, M., Sahraoui, H., Deb, K., & Inoue, K. (2017). MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal of Software: Evolution and Process*, *29*(5). https://doi.org/10.1002/smr.1843

Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., & De Lucia, A. (2018). *The scent of a smell: An Extensive Comparison between Textual and Structural Smells*. 740–740. https://doi.org/10.1145/3180155.3182530

Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A., & Oliveto, R. (2019). Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, *45*(2), 194–218. https://doi.org/10.1109/TSE.2017.2770122

Panigrahi, R., Kumar Kuanar, S., & Kumar, L. (2020). Application of Naïve Bayes classifiers for refactoring Prediction at the method level. *2020 International Conference on Computer Science, Engineering and Applications (ICCSEA)*.

Pantiuchina, J., Lanza, M., & Bavota, G. (2018). Improving code: The (mis) perception of quality metrics. *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, 80–91. https://doi.org/10.1109/ICSME.2018.00017

Paulk, Mark. (2016). *Code smells´ incidence does it depend on the application domain*.

Pecorelli, F., Palomba, F., Khomh, F., & De Lucia, A. (2020). Developer-Driven Code Smell Prioritization. *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, 220–231. https://doi.org/10.1145/3379597.3387457

Peruma, A., AlOmar, E. A., Newman, C. D., Mkaouer, M. W., & Ouni, A. (2022). *Refactoring Debt: Myth or Reality? An Exploratory Study on the Relationship Between Technical Debt and Refactoring*. http://arxiv.org/abs/2203.05660

Rahman, M. M., Riyadh, R. R., Khaled, S. M., Satter, A., & Rahman, M. R. (2018). MMRUC3: A recommendation approach of move method refactoring using coupling, cohesion, and contextual similarity to enhance software design. *Software - Practice and Experience*, *48*(9), 1560–1587. https://doi.org/10.1002/spe.2591

Rahman, M., Riyadh, R. R., & Rahman, R. (2017). *Recommendation of Move Method Refactorings Using Coupling, Cohesion and Contextual Similarity*.

Rani, A., & Chhabra, J. K. (2017a). *Evolution of Code Smells over Multiple Versions of Softwares An Empirical Investigation*.

Rani, A., & Chhabra, J. K. (2017b). *Prioritization of Smelly Classes A Two Phase Approach*.

Reeshti, Sehgal, R., Nagpal, R., & Mehrotra, D. (2019, November 21). Measuring Code Smells and Anti-Patterns . *4th International Conference on Information Systems and Computer Networks (ISCON)*.

Sae-Lim, N., Hayashi, S., & Saeki, M. (2017). How do developers select and prioritize code smells? A preliminary study. *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*, 484–488. https://doi.org/10.1109/ICSME.2017.66

Sangeetha, M., & Sengottuvelan, P. (2017). *Systematic Exhortation of Code Smell Detection Using JSmell for Java Source Code*.

Sharma, T. (2018). Detecting and managing code smells: Research and practice. *Proceedings - International Conference on Software Engineering*, 546–547. https://doi.org/10.1145/3183440.3183460

Shen, L., Liu, W., Chen, X., Gu, Q., & Liu, X. (2020). Improving machine learning-based code smell detection via hyper-parameter optimization. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, *2020-December*, 276–285. https://doi.org/10.1109/APSEC51365.2020.00036

Singh, R., Bindal, A., & Kumar, A. (2019). A user feedback centric approach for detecting and mitigating god class code smell using frequent usage patterns. *Journal of Communications Software and Systems*, *15*(3), 245–253. https://doi.org/10.24138/jcomss.v15i3.720

Sirikul, K., & Soomlek, C. (2016). *Automated Detection of Code Smells Caused by Null Checking Conditions in Java Programs*.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., & Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 4–15. https://doi.org/10.1145/2970276.2970340

Turkistani, B., & Liu, Y. (2019). *Reducing the Large Class Code Smell by Applying Design Patterns*.

Ubayawardana, G. M., & Karunaratna, D. D. (2018). *Bug Prediction Model using Code Smells*. 446.

Xu, S., Guo, C., Liu, L., & Xu, J. (2017). *A Log-linear Probabilistic Model for Prioritizing Extract Method Refactorings*.

Zhang, D., Li, B., Li, Z., & Liang, P. (2018). A preliminary investigation of self-Admitted refactorings in open source software. *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE*, *2018-July*, 165–168. https://doi.org/10.18293/SEKE2018-081

Zhu, H., Li, Y., Li, J., & Zhang, X. (2023). An Efficient Design Smell Detection Approach with Inter-class Relation. *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE*, *2023-July*, 181–186. https://doi.org/10.18293/SEKE2023-208