Code Smell Severity in Predictive Modeling for Software Quality Risk and Maintenance Effort: A Systematic Literature Review

Aamina Banu[1]*, Thenuri Hettiarachchi[2], Chaman Wijesiriwardana[3]
School of Computing, Asia Pacific Institute of Information Technology, Colombo, Sri Lanka[1,2]
Department of Information Technology, University of Moratuwa, Moratuwa, Sri Lanka[3]

***Abstract*—Code smells are structural indicators of poor software design that have been empirically associated with higher defect rates and increased maintenance effort. Although research on smell detection, software quality risk, and maintenance effort estimation has grown significantly, these areas have largely been studied separately. In addition, the role of smell severity in predictive modeling remains insufficiently synthesized. This Systematic Literature Review (SLR) investigates empirical studies published between 2010 and 2025, while also including selected foundational studies where appropriate. A total of 38 studies were selected for analysis through a structured search and screening process. The review addresses two research questions: 1) how smell type and severity influence software quality risk and maintenance effort, and 2) the extent to which current predictive approaches incorporate smell-related information in software defect prediction. The findings show that design-level smells are more consistently associated with fault-proneness than method-level smells. The review also finds that severity-aware models generally outperform binary detection models. However, the independent effect of code smells on directly measured maintenance effort remains inconsistent when confounding factors such as code size and code churn are taken into account. Furthermore, fully unified models that jointly predict defect risk and maintenance effort using smell severity as structured input remain largely absent. Overall, these findings highlight the need for standardized severity operationalization, integrated datasets, and interpretable unified modeling frameworks to advance software quality analysis.**

***Keywords*—Code smells; maintenance effort; software quality risk; smell severity; software quality**

I. Introduction

Software systems are being continuously evolved with the addition of new features, fixing defects, and with improvements in software performance and security. However, with software being large and complex, there still remains complexity in maintaining high quality of the software [1], [2], [3]. Two major concerns in software evolution are software quality risk and maintenance effort. Software quality risk means the possibility of bugs, errors, or system problems. Maintenance effort refers to the time and work required to fix, update, or improve the software after it is released [1], [4], [5].

The presence of code smells is one of the widely recognized signs of bad software quality [2], [3], [6]. Code smells do not cause failures immediately; instead, they show deeper structural or design-related problems caused by poor design implementation. Fowler was the first to describe the code smells as warning signs of a system which makes the code difficult to understand, modify, or extend [2]. Since then, multiple studies have shown that code affected by smells is more likely to contain defects and is harder to maintain [3], [7], [8].

Code smells can vary in both type and impact. Some smells mostly appear at design level, like God Class, Brain Class, and they usually point to deeper structural issues in the software. These problems usually happen when one single class has too many responsibilities or when classes are very closely connected [7], [9], [10]. As a result, they reduce overall system quality and make software maintenance difficult over time [7], [9], [11], [12]. Smells can also occur at method level like Long Method and Feature Envy. These indicate poor design in individual functions and make the code difficult to understand, update, or maintain in the long run [13], [14].

However, not all code smells have the same level of impact. For example, a slightly long method may not cause serious issues, but a very long and complex method can significantly increase defect risk and maintenance effort. Therefore, identifying the severity of code smells helps developers and researchers identify whether a smell is a minor issue or if it has the possibility to cause serious risks for the software quality [13], [15], [16]. In addition, when multiple code smells appear together in the same component, their combined effect can increase software quality risk further beyond what each smell would cause individually [17], [18].

Even though multiple research studies exist on code smell detection, software quality risk analysis, and maintenance efforts, these areas have mostly been studied independently. Many predictive models depend on traditional metrics like size and complexity, while maintenance studies usually focus on change frequency and technical debt without integrating software quality risks [19], [20], [4].

Furthermore, the direct connection between code smells and maintenance effort becomes less clear when factors like code size and code churn are taken into account, which makes it harder to isolate the independent effect of smells on effort [4], [21]. As a result, software quality is not examined in an integrated and comprehensive manner [20], [22], [23]. Prior secondary studies reflect this same separation. Existing reviews have mapped smell research broadly [24], synthesized the reported effects of smells [25], surveyed machine learning

*Corresponding author.

techniques for smell detection [26], examined the relationship between smells and quality attributes [27], and reviewed the use of smells in defect prediction [23]. However, these reviews largely address detection accuracy or single quality outcomes, and none of them synthesizes smell severity as a structured predictive input across both defect risk and maintenance effort. This review focuses on that combination. Another key challenge is that effort is defined inconsistently across studies and there is no standard way to evaluate unified predictive models, which makes it difficult to combine defect risk and maintenance effort prediction into a single framework [28]. To address this gap, this SLR aims to identify whether code smell type and severity can be used together in predictive and analytical models in order to explain software quality risk and long-term maintenance effort within a unified framework. The review is guided by the following research questions:

RQ1: What empirical evidence exists on how different code smell types and their severities impact software quality risk and long-term maintenance effort?

RQ2: To what extent have existing approaches integrated code smell information into defect prediction, and what methodological gaps prevent the development of unified predictive models?

These questions were made to understand how code smell type and severity can support more integrated, maintenance-effort-focused and risk-aware software quality analysis.

The remainder of this paper is organized as follows. Section II reviews existing studies on code smells. Section III discusses the research methodology, which also includes the search process and selection criteria. Section IV discusses the findings from the selected studies, which are structured according to each research question. Section V shows the overall outcomes of the review. Section VI outlines the threats to validity identified during this study. Section VII concludes the paper by summarizing the main findings, highlighting the limitations, and suggesting directions for future research.

II. Literature Review

Over the past two decades, there has been growing research on code smells, software quality risk analysis, and maintenance effort. However, these areas have often been studied independently, with limited integration between them. To provide a clear review that matches the research questions, this section discusses previous studies under five main themes: 1) code smell detection and severity assessment, 2) the relationship between code smells and software quality risk, 3) the effect of code smells on maintenance effort, 4) the use of machine learning in software quality prediction, and 5) recent studies on unified predictive models.

A. Code Smell Fundamentals and Detection

Code smells were initially introduced by Fowler in his work on refactoring as signs of poor design and structural problems in software systems [2]. Although code smells do not directly cause system failures, they can show deeper design and maintainability issues which can negatively affect the evolution of the software over time [29], [30]. Early detection methods mainly used fixed thresholds based on static code metrics, while recent studies have moved towards machine learning and deep learning based approaches which can provide better accuracy [29], [30], [31]. This shift shows a move from rule-based detection methods to a data driven approach. In addition to identifying whether a code smell is present, researchers have also started focusing on how severe the smell is. Instead of treating code smells as simply present or absent, many studies now use metric-based scores, ranking methods, and context-aware measures to better understand how serious a smell is within the code structure [13], [15]. Severity provides additional information about the potential impact of a code smell on software quality. It also helps developers prioritize refactoring more effectively by allowing them to focus on more serious code smells instead of treating all smells as equally important. These two lines of work differ in what they optimize. Metric-threshold and machine learning detectors are evaluated primarily on classification accuracy, whereas severity-based approaches are evaluated on how well they rank or prioritize instances. Existing reviews of detection techniques focus on detection accuracy [26], [24], so the value of severity estimation as an input to downstream quality models remains largely unexamined.

B. Code Smells and Software Quality Risk

Several empirical studies have shown a strong relationship between code smells and defect proneness [7], [23], [3], [32]. For example, Olbrich et al. carried out important longitudinal studies which found that code smells like God Class and Brain Class are strongly connected to higher defect rates [7]. These findings suggest that poor structural design can lead to more faults with time. Similarly, Piotrowski and Madeyski found that adding smell-related metrics as predictive features improved bug prediction accuracy by 9% to 16% compared to baseline models that did not use smell information at all [23]. More studies have shown that design-level smells are strongly associated with software defects. Components with these design issues are nearly two to three times more likely to contain defects than a well-designed component [3], [32].

In addition to smell type, severity also plays an important role in prediction and performance. Code smells that have higher severity are more likely to be associated with defect introduction than less severe ones. Studies by Palomba and others have shown that, among models that already incorporate smell information, those which additionally consider smell severity perform better than those that only identify the presence or absence of a smell, improving results by around 5% to 16% [15], [33]. These findings suggest that models that consider smell intensity can better capture deeper structural problems in the code when compared to the models that only identify whether a smell is present or not.

These studies differ methodologically in ways that affect how their results should be read. Repository mining studies such as Olbrich et al. [7] observe large numbers of releases but cannot control for confounding factors directly, whereas prediction studies such as Piotrowski and Madeyski [23] report performance gains without establishing whether the gain comes from severity itself or from the size and complexity signals correlated with it. Prior reviews of the code smell effect have noted this inconsistency across primary studies as an unresolved issue [25], [27].

Overall, the literature consistently shows that both smell type and severity can improve the accuracy of software quality risk prediction, highlighting the importance of smell-aware quality modeling.

C. Code Smells and Maintenance Effort

In addition to increasing defects, code smells also make software maintenance harder over time. Longitudinal studies show that fixing defects in classes with smells requires twice as many code changes compared to fixing defects in cleaner code [7], [12]. This indicates that structural weaknesses in software make corrective changes more complex and harder to implement. Research by Peters and Zaidman also showed that God Class and Brain Class smells tend to persist in systems for long periods and significantly increase maintenance effort [9]. These severe smells also impact multiple parts of the system, leading to increased technical debt [11].

When multiple code smells occur in the same class, maintenance becomes much harder. Such classes require more frequent changes and more effort compared to classes with only one or no smells [6], [34]. The findings indicate that co-occurring smells are associated with greater maintenance complexity than individual smells.

It should be noted, however, that these studies measure effort in different ways. Repository-based studies infer effort from the number or frequency of changes [7], [12], while controlled studies measure developer time directly [4]. This difference in operationalization is one reason why the reported strength of the smell-effort relationship varies across the literature.

Overall, the reviewed studies associate high-severity and co-occurring smells with increased maintenance effort and longer-term system instability.

D. Machine Learning in Software Quality Prediction

Machine learning has become a widely used approach for predicting software defects. Random Forest, in particular, has demonstrated strong performance in a recent code-smell-related prediction study [35]. These models are often preferred because they are robust and can handle large, complex sets of software metrics effectively. Deep learning approaches have further improved prediction performance. For example, methods such as Deep Forest can increase AUC by 5% to 8% compared with traditional machine learning techniques [15]. Ensemble methods that combine techniques such as SMOTE with optimized classifiers have achieved over 97% accuracy on benchmark datasets [36].

These reported gains are not directly comparable, as the studies use different datasets, validation procedures, and performance indicators. Accuracy figures obtained on balanced benchmark data, in particular, are not equivalent to AUC gains reported on imbalanced project data.

These findings indicate that predictive performance can improve when models incorporate code smell characteristics, including severity.

E. Unified Prediction Models

Recent research has started exploring unified modeling approaches which combine related software quality prediction tasks into a single framework, rather than handling each task separately. Multi-task deep learning frameworks that jointly predict, categorize, and repair defects within a single model have been shown to improve prediction accuracy by sharing information across tasks, resulting in about a 4% to 7% improvement [37]. However, these methods still face challenges, like lack of combined datasets, inconsistent ways of measuring severity, more complex models, and difficulty in understanding how models make decisions [22].

Although modern predictive techniques show strong potential, the research areas remain fragmented. Most models focus either on risk-aware quality prediction or on general code quality assessment, and few consider maintenance effort as a prediction target. The literature therefore lacks a unified framework that jointly considers smell type, severity, defect risk, and maintenance effort. Existing secondary studies do not close this gap either, as they review detection techniques [26], the smell literature as a whole [24], reported smell effects [25], or smells in relation to quality attributes [27] and defect prediction [23] separately. The present review differs in that it treats severity as the organizing concept and examines defect risk and maintenance effort together rather than in isolation.

III. Methodology

This paper reviews existing studies to understand how different types of code smells and their severity affect software quality risk and maintenance effort. The review follows a structured and transparent methodology, with explicit search, screening, and quality-assessment stages, to ensure that the process is systematic and reproducible. The aim is to carefully review existing research findings, compare the results, identify common trends and differences, and highlight the research gaps related to smell-aware software quality analysis.

The review was conducted in four main steps: first deciding the scope of the study, then searching for relevant research papers, selecting the most suitable studies, and then combining their findings. These steps helped to keep the review well-structured and closely aligned with the research goals.

A. Review Scope and Planning

The review focuses on studies that look at code smells as signs of software quality problems, mainly in relation to software quality risk and maintenance effort. However, more attention is given to research that identifies the type of code smell and how severe it is, as these factors help explain how much code smells influence the long-term evolution of software.

Before starting the search process, a review plan was developed in order to guide the methodology. The plan defined the research focus, identified relevant academic databases, outlined inclusion and exclusion criteria, and specified the type of data to be extracted from selected studies. Establishing this plan early ensured that the review remained focused on the central research questions and reduced the risk of including unrelated studies.

B. Keywords Utilized

A selected set of keywords was used for this literature search. The keywords were grouped into three main groups which were code smell concepts, software quality outcomes, and analysis context. Boolean operators (AND, OR) were applied to systematically combine these groups and refine the search results.

1) Code smell concepts: ("Code smell" OR "Design smell" OR "Software smell" OR "Smell severity" OR "Smell prioritization")

2) Software quality outcomes: ("Software quality risk" OR "Bug prediction" OR "Fault-proneness" OR "Maintenance effort" OR "Change proneness" OR "Maintainability")

3) Analysis context: ("Software quality" OR "Software maintenance" OR "Software evolution" OR "Predictive models" OR "Empirical software engineering")

These Boolean search strings were applied across multiple digital libraries which includes IEEE Xplore, ACM Digital Library, SpringerLink, ScienceDirect, and Wiley Online Library, to ensure comprehensive coverage of peer-reviewed literature relevant to smell-aware software quality analysis.

C. Literature Search Strategy

The literature search was carried out using well-known digital libraries that are commonly used in software engineering research. The search mainly focused on studies published between 2010 and 2025 to capture recent research developments, while also including a few earlier important studies when necessary. In addition to keyword-based searches, the references and citations of important papers were also reviewed to find additional relevant studies.

D. Study Selection Criteria

Clear inclusion and exclusion criteria were used to make sure that only relevant and reliable studies were selected. Studies were included if they were published in peer-reviewed journals or conference proceedings, reported empirical results or detailed analytical investigations, examined the relationship between code smells and software quality outcomes such as quality risk indicators, fault-proneness, change-proneness, or maintenance effort, and discussed smell severity, prioritization, or impact levels.

Studies were excluded if they focused only on detecting code smells without having a connection to quality outcomes, if they were not peer-reviewed or were grey literature, were written in languages other than English, or did not provide enough methodological detail. Applying these criteria helped ensure that the selected studies directly supported the objectives of the review. Table I summarizes these criteria together with the reason each one was applied.

TABLE I. INCLUSION AND EXCLUSION CRITERIA AND THEIR JUSTIFICATION

Type	Criterion	Justification
Include	Published in a peer-reviewed journal or conference proceedings	Ensures that the reviewed evidence has undergone independent scholarly assessment
Include	Reports empirical results or a detailed analytical investigation	The review synthesizes empirical evidence rather than opinion or position statements
Include	Examines the relationship between code smells and quality outcomes such as quality risk indicators, fault-proneness, change-proneness, or maintenance effort	Directly addresses the outcomes specified in RQ1 and RQ2
Include	Discusses smell severity, prioritization, or impact levels	Severity is the organizing concept of this review
Exclude	Detects code smells without any connection to quality outcomes	Detection accuracy alone does not inform the relationship under study
Exclude	Not peer-reviewed, or grey literature	Reduces the risk of including evidence of unverified quality
Exclude	Written in a language other than English	The review team could not reliably extract data from other languages
Exclude	Insufficient methodological detail	Data extraction requires a reported study design, dataset, and analysis method

E. Study Selection Process

The initial search identified 500 publications across the databases listed above. After removing 75 duplicate records, 425 records remained for screening. Title screening removed 250 records, leaving 175 abstracts to be assessed. Abstract screening removed a further 100 records, leaving 75 full-text articles assessed for eligibility. Of these, 37 were excluded for not meeting the inclusion criteria, providing insufficient empirical evidence, not being relevant to the research questions, or not reporting enough methodological detail. This process resulted in 38 studies being selected for detailed analysis. The complete flow of records through identification, screening, eligibility, and inclusion is shown in Fig. 1. The reference list also includes background and methodological sources, along with prior secondary studies, which are not counted among the selected primary studies. Screening was carried out consistently to maintain reliability throughout the study selection process.

F. Data Extraction and Analysis

Relevant information from each selected study was manually extracted and recorded in a data extraction table. This included details like the study context, the types of code smells examined, how smell severity was defined or measured, the quality outcomes considered (such as defects or maintenance effort), the analytical or predictive methods used, the main findings, and any reported limitations. Organizing the studies in this way made it easier to compare them and identify common patterns, differences, and research gaps.

G. Assessment of Study Quality

The methodological quality of the selected studies was evaluated using four criteria, each scored 0 (criterion not met), 0.5 (criterion partially met), or 1 (criterion fully met):

- QA1: whether the research design was clearly explained;
- QA2: whether appropriate datasets and analytical methods were used;
- QA3: how consistent the reported findings were;

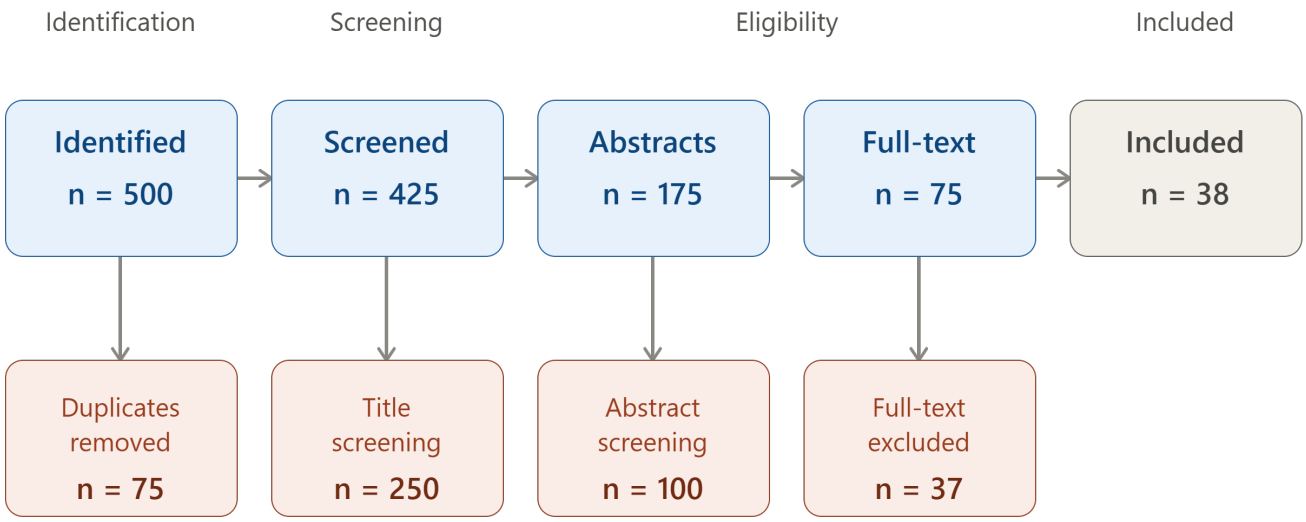

Fig. 1. PRISMA flow diagram of the study identification, screening, eligibility, and inclusion process.

- QA4: how openly the study discussed possible limitations or threats to validity.

The four scores were summed to give each study a quality score out of 4. These scores are reported for the key studies summarized in Table II and Table III, alongside their focus areas, key findings, and reported limitations.

Many of the reviewed studies were published in reputable venues, including conferences organized by IEEE and well-recognized academic journals, which further supports the reliability of the reviewed evidence.

H. Classification of Effect Strength

In addition to the QA1–QA4 quality score, Table II and Table III report an Effect Strength rating (Strong, Moderate, or Mixed, including intermediate labels such as Moderate to Strong where the evidence for a study fell between two categories) for each summarized study. This rating is a structured qualitative judgment formed as part of the descriptive synthesis approach described later in this section, not a statistical effect-size calculation, since the included studies were too heterogeneous in design, dataset, and outcome measure to support a quantitative meta-analysis.

Effect Strength was assigned by considering, for each study: 1) whether its findings were consistent with, or contradicted by, other reviewed studies addressing the same or a closely related question; 2) the scale and nature of the evidence reported, such as the number of systems, releases, or projects analyzed; and 3) whether the reported association held after the study accounted for potential confounding factors, such as code size or churn, where applicable. A rating of Strong was assigned where findings were consistent across multiple studies and supported by large-scale or well-controlled evidence; Moderate was assigned where findings were directionally consistent but based on more limited evidence or a narrower context; and Mixed was assigned where the reviewed evidence on that specific point was inconsistent or sensitive to confounding factors. This classification is descriptive rather than statistical, was not derived from a formal scoring rubric or inter-rater agreement procedure, and is intended only to help readers compare the relative weight of evidence across the summarized studies.

I. Synthesis of Findings

As the studies used different datasets, measures, and models, the results were not combined statistically. The findings were explained using a descriptive method and organized according to the research questions. If studies reported numbers like effect sizes or performance improvements, they were highlighted to show how strong the relationships were. This method helped present a clear overall view of how code smell type and severity are linked to defects and maintenance effort.

IV. Results

Empirical evidence consistently shows that code affected by smells is more likely to change and more prone to faults than code without smells, especially in large-scale studies that analyze many projects and releases [17], [8]. The evidence is even stronger when smell severity and co-occurrence of smells are taken into account. Many prediction studies show that including smell intensity improves performance metrics such as Area Under the Curve (AUC) and F-measure by about 5% to 16% compared to using only presence or absence of smells [17], [33], [14].

However, when studies focus on directly measuring the effort rather than the defect or change risk, the results are less clear. After accounting for factors like size and code churn, the presence of smells often shows only limited independent effect on maintenance effort [4], [21].

The literature also shows a clear progress in multi-task defect learning and effort-aware defect prediction. However, unified models that predict both defect risk and maintenance effort as equally important outputs are still rare. A major challenge is the lack of a standardized definition of effort and

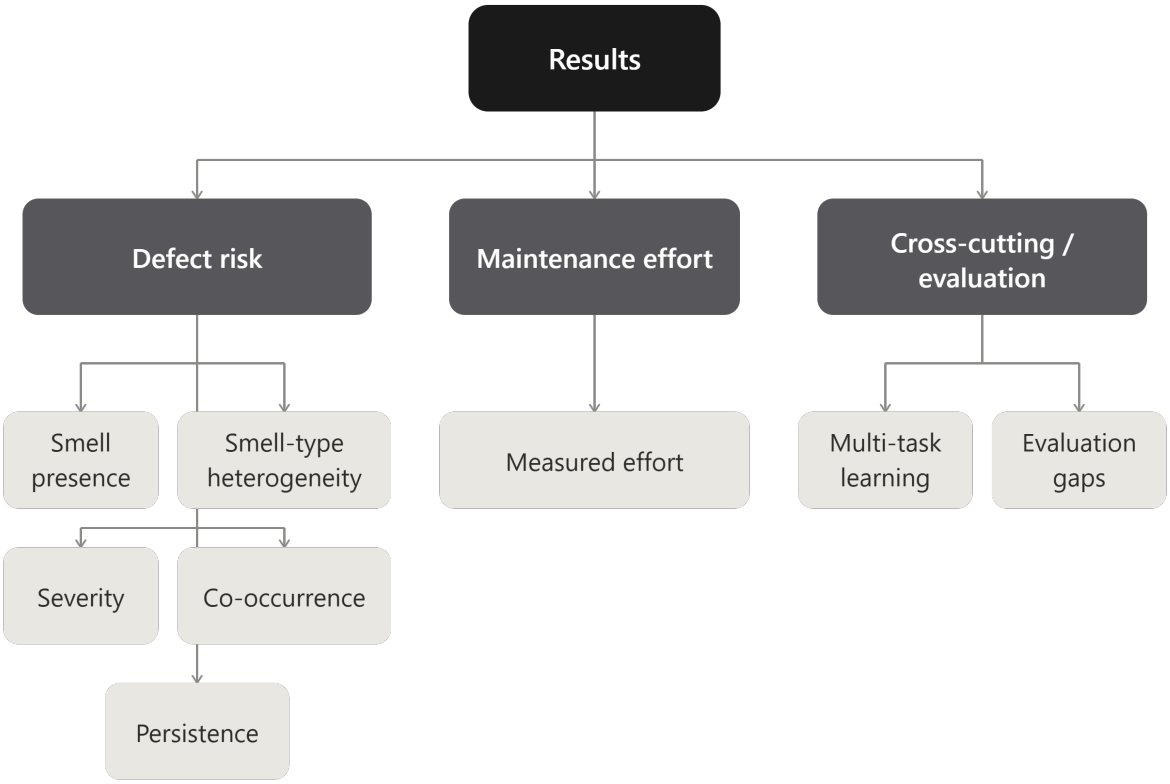

Fig. 2. Overview of results on code smell impacts, organized into defect risk (smell presence, smell-type heterogeneity, severity, co-occurrence, and persistence), maintenance effort (measured effort), and cross-cutting evaluation themes (multi-task learning and evaluation gaps).

consistent evaluation methods, making it difficult to compare results across studies [37], [28].

An overview of how these results are organized across the two research questions is shown in Fig. 2.

A. RQ1: Smell Types, Severity and their Impact on Software Quality Risk and Maintenance Effort

1) Smell presence and defect or change risk: Large-scale empirical studies consistently show that classes with code smells are more prone to changes and defects. This pattern holds in longitudinal analyses across multiple open-source systems and hundreds of releases, indicating that the findings are robust across different datasets and ecosystems [17], [8]. Because these findings are supported by large datasets, manual validation, and replication across multiple systems, the evidence linking code smells to defect and change risk is considered strong.

However, comparisons across studies also show that results are sensitive to confounding factors, especially size and code churn. When effects are adjusted for size, some results can even reverse, and certain smell types appear less change-prone than what the raw counts might suggest [7]. This shows that code smells often appear in larger, more complex components, which are naturally more prone to defects. From a synthesis perspective, therefore, although the association is strong at scale, it should be interpreted with caution rather than as a direct cause.

2) Smell type heterogeneity: The literature consistently shows that not all smell types are the same. Systematic reviews and large-scale studies indicate that certain smells, such as God Class, God Method, Message Chains, Blob, and other complexity-related smells, are consistently stronger predictors of defects [10], [38].

Importantly, some smells are strongly linked to faults even when they occur infrequently, suggesting that frequency alone is not a reliable measure of their impact [38]. Since this pattern is observed both in systematic reviews and in independent empirical datasets, the evidence for the differences in risk among smell types is strong [23], [38].

3) Severity and intensity as predictive amplifiers: A major shift in research after 2010 has been the move from simply detecting the presence of code smells to modeling their severity. Many prediction studies show that including smell intensity as a feature improves bug and change prediction models compared to using only binary smell presence [33], [14], [39].

Quantitatively, these studies usually report improvements in AUC and F-measures of about 5% to 16%, depending on the dataset and model design [33], [14], [39]. This shows that considering severity provides a meaningful improvement in prediction, not just a small or marginal gain.

Related work on severity classification using machine learning also supports the idea that severity is a useful concept for prioritizing work, even when defects themselves are not directly predicted [13], [16], [35]. Overall, evidence for severity

as a meaningful predictive amplifier is strong in prediction contexts.

4) Co-occurrence and agglomeration effects: Across different studies, software artifacts that contain more than one code smell tend to be riskier than those with just a single smell. Research also shows that code smells often appear together, and when they do, they are linked to a higher chance of faults and future changes [40].

Evidence from both experiments and real industry settings shows that relationships between multiple code smells can significantly increase maintenance effort, suggesting that their combined impact is greater than just the sum of individual smells [18]. Recent studies on code smell agglomerations further support this idea, showing that groups of related smells can reduce code stability across several Java systems [41].

Since these findings are reported in both software mining studies and maintainability-focused research, the evidence for clustering effects can be considered moderate to strong, depending on how the studies were conducted.

5) Smell persistence and survivability: Longitudinal studies show that smells frequently persist once introduced and are not consistently removed through explicit refactoring actions [42]. Survivability analysis further shows that smells and vulnerabilities tend to remain longer than bugs, suggesting they represent durable technical debt signals rather than transient noise [43]. Since lifecycle and survivability findings depend on detection thresholds and operational definitions, the strength of evidence here is moderate, but directionally consistent.

6) Smells and measured maintenance effort: When the dependent variable shifts from defect or change risk to directly measured maintenance effort in controlled experiments, results become less consistent. Controlled in vivo studies show limited independent smell effects after controlling for size and change characteristics [4]. Activity-based effort decomposition shows that smell impact may vary across reading, editing, searching, and navigation activities, meaning aggregated effort metrics can mask localized cognitive costs [21].

Thus, evidence linking smell presence to defect-proneness is strong, whereas evidence linking smell presence to direct measured maintenance effort is mixed to moderate, and highly sensitive to confounding and operationalization. These findings suggest that the distinction matters practically: teams relying solely on smell presence to forecast maintenance workload may find predictions unreliable unless size and churn are carefully controlled. This also implies that smell-aware effort models need to be designed with explicit confound management rather than treating smell counts as standalone predictors.

Overall, the empirical evidence suggests that smell type and severity are reliable indicators of software quality risk and change-proneness, while their independent effect on direct maintenance effort remains context-dependent and sensitive to confounding factors such as size and churn. That said, this does not diminish the value of smell-based analysis. Rather, it refines how practitioners should interpret smell signals: as early warning indicators of structural risk that tend to co-occur with costly maintenance contexts, rather than as direct and isolated causes of effort. Moving forward, research designs that separate structural risk signals from size-related noise will be essential for building more trustworthy and actionable smell-aware quality models.

To provide a concise overview of the empirical evidence discussed above, Table II summarizes key studies on the impact of different smell types and severity on software quality risk and maintenance effort. It highlights the focus of each study, the main findings, and the limitations reported by the authors, allowing readers to quickly compare and interpret the strength of evidence across multiple investigations.

B. RQ2: To What Extent have Existing Approaches Integrated Code Smell Information Into Defect Prediction, and What Methodological Gaps Prevent the Development of Unified Predictive Models?

1) Multi-task learning within the defect workflow: Recent work has tried combining defect tasks into single models rather than treating each one separately. Studies show that prediction, categorization, localization and repair can run together in one framework without badly hurting individual task results [37], [44]. Shared training helps because all these tasks draw from similar information about code structure anyway. When multiple related objectives are learned together, the model can pick up on patterns that would be harder to detect if each task were handled in isolation, and this tends to produce more robust representations of code quality overall.

But the scope of what gets combined is limited. Effort prediction is not part of it. These models identify and describe defects but they do not try to estimate how much work fixing them will require [37], [44]. From a synthesis perspective, that is a significant omission, because knowing a defect exists and knowing how expensive it is to address are both necessary for practical prioritization decisions. In real development settings, teams rarely have the luxury of addressing every issue that gets flagged, so understanding the likely cost of resolution is just as important as knowing where the problems are.

Code smells are similarly absent as deliberate inputs. Indicators like God Class, Long Method and Feature Envy have established connections to fault density and maintenance difficulty but they do not appear as intentional feature categories in multi-task defect architectures. At best they get bundled into general metric collections without any specific consideration of what they contribute beyond other complexity measures. Across the reviewed studies, this appears to be a missed opportunity, given how much empirical evidence already supports their predictive value.

2) Effort-aware defect prediction: The motivation for effort-aware prediction is straightforward. Inspection takes time and not every flagged module can be checked so models should reflect that constraint [28]. This direction has attracted considerable research and has produced evaluation methods built around inspection budgets. The underlying idea is sensible and practically grounded, which is probably why it has gained traction in the research community over the past several years.

A central difficulty is that effort has no agreed definition. Lines of code, churn and complexity proxies are all used and the choice affects results significantly [28]. Model rankings shift depending on which definition a study adopts, which

TABLE II. RQ1: COMPARATIVE SUMMARY OF KEY STUDIES ON SMELL TYPE AND SEVERITY IMPACTS

Author	Focus Area	Key Findings	Reported Limitations	QA (/4)	Effect Strength
Olbrich et al. [7]	Longitudinal historical analysis of design-level smells	Smelly code showed more changes and defects in raw analysis. However, after adjusting for size, the relationship could reverse, showing that smell effects are strongly influenced by class size.	Strong confounding effect of code size; only a small number of systems and smell types were examined.	3.5	Strong
Palomba et al. [17]	Large-scale empirical mining of smell diffusion and impact	Code containing long or complex smells was found to be more change-prone and fault-prone across 395 releases and 30 projects.	Mainly observational evidence; code size and churn may still influence the findings.	4	Strong
Palomba et al. [40]	Lifecycle and co-occurrence analysis of code smells	Multiple smells were found to commonly co-occur, and code with several smells showed higher risk than code with only one smell.	Findings depend on the detection tools and threshold values used to identify smells.	3.5	Moderate to Strong
Yamashita and Moonen [18]	Inter-smell relations and maintenance effort	Certain combinations of smells were shown to significantly increase maintenance effort, suggesting compounding effects between smells.	Results depend on project context and how maintenance effort was operationalized.	3	Moderate to Strong
Sjøberg et al. [4]	Controlled in vivo maintenance study with professionals	After controlling for size and number of changes, code smells showed only a weak independent effect on time-based maintenance effort.	Based on only a few systems and limited maintenance tasks.	3	Mixed to Moderate
Soh et al. [21]	Activity-based maintenance effort analysis	The effect of code smells differed depending on the maintenance activity being measured, such as reading, editing, searching, or navigation.	Study setting and measurement approach may limit generalizability.	3	Mixed to Moderate
Palomba et al. [14]	Prediction modeling using smell intensity	Adding smell intensity (severity) improved the performance of bug prediction models compared to simpler smell representations.	Results are highly dependent on the datasets and prediction models used.	4	Strong
Palomba et al. [33]	Smell-aware bug prediction using severity information	Models that incorporated smell severity outperformed basic binary smell detection models in bug prediction tasks.	Effectiveness depends on accurate smell detection and reliable severity measurement.	4	Strong
Tufano et al. [42]	Longitudinal smell lifecycle analysis	Once introduced, code smells were found to persist for long periods and were rarely removed directly through refactoring.	Cause-and-effect relationships are difficult to establish clearly.	3.5	Moderate
Zabardast et al. [43]	Survivability analysis of technical debt items	Code smells and vulnerabilities were found to survive longer in systems than bugs, indicating long-term persistence.	Results vary depending on how technical debt items are defined and categorized.	3	Moderate
Rahman et al. [38]	Smell-frequency and fault-proneness correlation study	Specific smells such as Anti-Singleton, Blob, and Class Data Should Be Private showed strong associations with fault-proneness.	Mostly correlational evidence; results depend on how smells and faults were mapped.	3.5	Strong

means comparisons across papers are unreliable. What looks like a strong model in one study may look weak in another that used different effort assumptions, and there is currently no standard to resolve the disagreement. This makes it genuinely difficult to know whether a proposed improvement reflects a better model or simply a more favorable effort definition.

Size conflation makes this worse. When effort is proxied mainly by lines of code the prediction task becomes partly about file size rather than actual correction cost [28]. These are related but not equivalent and treating them as equivalent distorts what the model is actually learning. Composite proxies combining churn and difficulty have been proposed as a fix [45], but adoption is inconsistent. Studies incorporating smell metrics face additional complications here, because smell density correlates with complexity, which in turn correlates with size-based effort proxies, making it difficult to isolate the smell contribution from size-related noise.

3) Direct effort prediction models: A separate research thread models effort directly as a prediction target. Work exists on estimating defect correction effort and maintenance effort using project history and code metrics [5], [46] and some defect prediction studies include effort metrics as predictor variables [47]. This body of work confirms that effort is measurable and predictable, which is a necessary foundation for any unified approach. The fact that reasonably accurate effort estimates can be produced from available project data is encouraging and suggests the raw material for integration already exists.

The two areas remain disconnected in practice though. Defect models and effort models are developed and evaluated independently. Smell features, when present, serve as inputs within one stream rather than as shared features across a joint model that outputs both defect risk and effort estimates simultaneously. Evidence for effort predictability is reasonably strong but evidence for integrated defect plus effort modeling using smell information as structured input is essentially absent from current literature. Bridging this gap would require not just combining the two streams technically but also agreeing on how smells should be represented and weighted within a joint prediction objective.

4) Evaluation standardization as prerequisite for unification: The sensitivity of results to effort definition further complicates matters [28]. Without shared operationalizations of effort and agreed baseline comparators, individual studies cannot build on each other reliably. For unified models targeting both defect risk and effort prediction this problem is especially damaging because inconsistency affects both components and their combination produces results that are even harder to interpret across different evaluation contexts. The reviewed evidence therefore indicates a need for more careful and consistent reporting practices, standardized effort definitions, and shared benchmark datasets across the community.

5) Synthesis and Identified Gap: Across these four areas the literature shows that joint defect learning works, effort-

TABLE III. RQ2: SUMMARY OF JOINT AND UNIFIED MODELING APPROACHES RELEVANT TO DEFECT RISK AND EFFORT

Author	Focus Area	Key Findings	Reported Limitations	QA (/4)	Effect Strength
Dai et al. [45]	Effort-aware just-in-time defect prediction and optimization	Proposed an improved effort-aware defect prediction approach by ranking defect-prone changes under effort constraints. Effort was explicitly modeled as *difficulty* × *churn*.	Findings depend heavily on how effort is defined and modeled; does not directly predict overall maintenance effort.	3.5	Moderate
Lavazza et al. [28]	Evaluation and metric analysis for effort-aware prediction	Showed that evaluation outcomes can change significantly depending on the effort model applied, highlighting the sensitivity of results to effort assumptions.	Lack of standardized effort definitions makes it difficult to compare models fairly across studies.	4	Strong
Hassouna et al. [46]	Bug-fix effort prediction using kNN	Introduced a framework for predicting defect correction effort, where effort was treated as the final outcome and measured in person-hours.	Focused only on bug-fix effort and not on predicting defects earlier in the software lifecycle.	3	Moderate
Haldar et al. [5]	Interpretable maintenance and support effort prediction	Identified the most influential drivers of maintenance and support effort using SHAP, improving interpretability of effort prediction models.	Applied mainly at project level and did not integrate defect risk prediction into a unified model.	3.5	Moderate
Haldar et al. [47]	Interpretable defect prediction using effort and static metrics	Demonstrated reliable cross-project defect prediction and model interpretability by combining project effort metrics with static code metrics.	Defect prediction and maintenance effort prediction were still handled separately rather than within a unified framework.	3.5	Moderate

aware evaluation is a meaningful framing, and maintenance effort is predictable from project data.

A unified model that uses code smell information as structured input to jointly predict defect risk and maintenance effort does not exist in any validated form. Smell data is relevant to both outputs but has not been embedded in a framework that targets both simultaneously. The barriers are definitional inconsistency in how effort is operationalized and the absence of standardized evaluation conditions that would allow such a model to be meaningfully compared against alternatives.

Table III summarizes representative effort-aware and unified defect prediction studies and their reported limitations.

V. DISCUSSION

The findings from RQ1 and RQ2 show that research on code smells has developed considerably over time, but there are still important gaps that prevent a fully integrated understanding of how code smells affect software quality risk and maintenance effort. Overall, the evidence suggests that the impact of code smells cannot be understood through a simple yes or no detection alone. Instead, factors like smell type, severity, co-occurrence, and the way in which software outcomes are measured all influence how strongly code smells are related to software quality problems.

To represent these relationships in a single view, the evidence synthesized in Section IV is summarized as a conceptual framework, as illustrated in Fig. 3.

The framework in Fig. 3 is a conceptual synthesis rather than a new empirical model, and each element of it corresponds to a finding already reported in Section IV. The three input categories reflect the distinction between design-level smells, method-level smells, and co-occurring smells, which the reviewed studies report as carrying different levels of risk [7], [17], [10], [38], [40], [41]. The severity scoring stage reflects the operationalizations used in the reviewed severity literature, namely metric-based intensity, ranking indices, and context-aware measures [13], [15], [16], [35]. The separation between binary detection models and severity-aware models, together with the reported 5–16% improvement, is taken from the prediction studies reviewed under RQ1 [33], [14], [39]. The two outcome nodes correspond to the two outcome types examined throughout this review, and the figures shown for them are those reported in the reviewed studies for defect density [3], [32] and for code changes required in smelly classes [7], [12]. The final node is not an empirical result. It represents the unified defect-effort model that RQ2 found to be absent from the current literature, and it is included to show where the reviewed evidence converges rather than what has already been validated. The arrows in the figure therefore indicate the direction of evidence reported across the reviewed studies, and should not be read as causal claims, particularly on the maintenance effort path, where Section IV shows the evidence to be mixed.

A. From Binary Detection to Severity-Aware Modeling

Prior studies mainly focused on whether a code smell was present or absent, and these studies helped in identifying that smelly code is generally more fault-prone than clean code [3], [6], [7]. However, this binary view is limited because it treats all smell instances as equally important, even though some are clearly more harmful than others. This becomes especially important in practice, where developers need to decide which smells should be fixed first.

The findings of this review show that when smell severity is included as a predictive feature rather than a simple binary indicator, prediction models generally perform better [33], [14], [39]. Across the reviewed studies, severity-aware models reported improvements of around 5–16% in AUC and F-measure. This suggests that the intensity of a smell provides useful information that binary detection alone cannot capture. For example, a slightly long method may not be as problematic as a highly complex long method inside a God Class [13], [16], [35].

At the same time, the review also shows that there is still no consistent way of defining or measuring smell severity across studies. Different tools and studies use different thresholds and operationalization, making it difficult to compare findings or

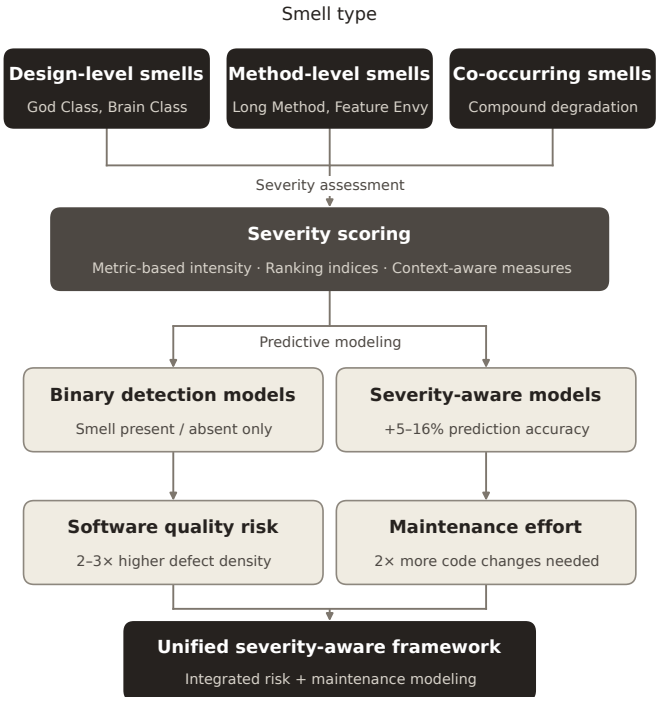

Fig. 3. Severity-aware code smell impact framework, derived from the findings of the reviewed studies.

apply severity consistently in practice. This shows that future research should move toward more standardized severity definitions if severity-aware models are to become more reliable and practically useful.

B. Smell Type as a Differential Risk Predictor

Another important pattern is that not all code smells carry the same level of software quality risk. The studies reviewed consistently show that design-level smells like God Class, Brain Class, and Blob tend to be stronger predictors of fault-proneness and change-proneness than method-level smells [7], [17], [10], [38].

A possible reason is that design-level smells usually show deeper structural problems in the system. For example, a God Class usually centralizes too many responsibilities and increases coupling, which can then spread risk across several parts of the codebase. In contrast, a smell such as Long Method may affect maintainability and readability, but its impact is often more localized. Olbrich et al. found that components affected by the God Class smell tended to evolve differently over time, showing different patterns in both code changes and defect occurrence [7], and Palomba et al. similarly found that different smell types have different levels of impact on software quality outcomes across a large number of releases [17].

This means that in practice, developers should not treat all smells as equally urgent. Some smell types, especially design-level smells, may deserve higher priority because they are more strongly linked to future software quality problems.

C. The Maintenance Effort Paradox

One of the most interesting findings of this review is the difference between software quality risk results and maintenance effort results. Large-scale repository-based studies often show that smelly code is changed more frequently and is associated with higher fault-proneness and structural degradation over time [7], [17], [40]. At first glance, this would suggest that code smells should also directly increase maintenance effort.

However, the evidence is less clear when it comes to actual maintenance effort. Sjøberg et al. observed that when factors like file size and the number of changes were taken into account, the effect of code smells on maintenance time was weaker [4]. This shows that smells do not always directly lead to higher maintenance effort. Instead, they usually appear with other challenging or costly areas of the software, and act more like warning signs than direct causes. Therefore, code smells are more useful as indicators of structural risk than as predictors of how much effort maintenance will require.

This pattern becomes even clearer when maintenance effort is examined more closely. Soh et al. showed that the impact of smells depends on the type of maintenance activity being performed [21]. For example, smells may create more difficulty during editing or navigation tasks than during reading or searching tasks. As a result, studies that use broad or aggregated effort measures may not fully capture how smells affect real maintenance work.

D. Compounding Effects and Smell Persistence

The findings also show that code smells often do not appear alone. Instead, they frequently co-occur with other smells, creating a more complex and risky structural context. Palomba et al. found that a large proportion of smelly classes contain more than one smell type [40], and other studies show that code with multiple smells tends to be more fault-prone and change-prone than code with only one smell [40], [18], [41].

The findings indicate that co-occurring smells may be associated with greater software quality risk than individual smells considered separately. Certain smell combinations may amplify software quality risk and make maintenance more difficult beyond what individual smells would predict on their own [18], [41]. Therefore, smell co-occurrence should not be treated as a secondary issue. It should be considered an important part of software quality analysis, especially when prioritizing refactoring or building predictive models.

Another related issue is smell persistence. Longitudinal studies show that once code smells are introduced, they often remain in the system for many releases and are rarely removed through direct refactoring [42]. Similarly, smells and other technical debt items may survive longer than bugs in software history [43]. This suggests that developers often respond faster to visible faults than to deeper structural quality problems. As a result, high-severity and co-occurring smells may continue to accumulate and influence software quality over time if they are not addressed early.

E. Towards Unified Defect-Effort Modeling

The findings from RQ2 show that although there has been progress in defect prediction, effort-aware modeling, and

interpretable machine learning, these areas are still not well integrated. Existing studies have explored effort-aware just-in-time defect prediction [45], multi-task learning [37], [44], and interpretable prediction models [5], [46], [47], but no validated model was found that jointly predicts both defect risk and maintenance effort using code smell severity as structured input.

One major reason for this gap is the inconsistency in how maintenance effort is measured. Some studies use developer time, some use lines modified, and others use effort proxies based on complexity or churn [4], [45]. Lavazza et al. showed that differences in effort definitions can significantly change evaluation outcomes [28]. This makes it difficult to compare models fairly or build unified prediction frameworks.

Another important gap is that current predictive approaches rarely incorporate the combined role of smell type, severity, and smell co-occurrence within the same model. This is important because the findings of RQ1 show that these factors do not operate independently. A severe design-level smell that appears together with other smells may carry much higher software quality risk than an isolated low-severity smell. Yet most existing models still simplify smell information into isolated features or binary indicators.

The sensitivity of evaluation outcomes to how effort is defined further highlights that, before unified models can be developed effectively, the field first needs more consistent evaluation practices, standardized effort definitions, and shared benchmark datasets [28]. Future work should therefore focus on building interpretable multi-output models that integrate smell type, severity, and co-occurrence in a way that can jointly support defect risk and maintenance effort prediction.

F. Limitations

This review has multiple limitations which have to be considered when interpreting the findings. Firstly, the reviewed large-scale empirical studies are based mainly on open-source Java systems, from Apache and Eclipse ecosystems [17], [38]. This can limit the generalizability of the findings to industrial systems, proprietary software, other programming languages, or different development environments. Secondly, smell detection itself is not fully consistent across studies. Different tools and detection rules use different thresholds in identifying the same smell type, which may affect the findings and reduce comparability between studies. This is important for severity-based analysis, where inconsistent thresholds can directly influence how severity is interpreted. Finally, this review provides a synthesis of the evidence rather than a formal meta-analysis. While this allows for broad comparison across study types and research designs, it also means that the findings should be interpreted as an evidence-based synthesis without a statistical aggregation of effect sizes.

VI. Threats to Validity

As with any literature review, there are a few limitations in the review process that should be considered. One possible issue is that some relevant studies may have been missed, even though the search was carried out using multiple academic databases and carefully chosen keywords. To reduce this as much as possible, the references and citations of important papers were also checked to identify additional studies.

Another possible limitation is publication bias, since this review mainly included peer-reviewed studies. This means that some useful findings from grey literature or unpublished work may not have been captured. However, focusing mainly on peer-reviewed sources helped improve the reliability of the studies included in the review.

There is also a chance of interpretation bias, as the selected studies were manually reviewed and summarized. To make this process more consistent, the relevant details from each study were recorded in a data extraction table and examined based on the research questions.

A further limitation concerns the reporting of the review process itself. The screening was carried out in the stages described in Section III, and the number of records remaining after each stage is reported in the PRISMA flow diagram (Fig. 1). The methodological quality of the selected studies was assessed using the QA1–QA4 criteria listed in Section III, with each criterion scored 0, 0.5, or 1 per study; the resulting quality scores for the key studies are reported alongside Tables II and III. To further support transparency, the screening stages and the inclusion and exclusion criteria were also stated explicitly, as summarized in Table I.

In addition, the reviewed studies did not all use the same definitions or measurements for code smells, smell severity, maintenance effort, and software quality outcomes. Since different datasets, tools, thresholds, and evaluation methods were used across studies, comparing the findings was not always straightforward. Because of this, the results of this review should be understood with these differences in mind.

VII. Conclusion

The review examined 38 empirical studies on understanding how code smell type and severity are linked to software quality risk and maintenance effort. The findings show that code affected by smells is more likely to contain faults and require more changes than clean code. Among the different types of smells, design-level smells such as God Class and Brain Class were found to be stronger indicators of risk than method-level smells. The review also found that models that consider smell severity perform better than those that only detect whether a smell is present or not. Another important finding is that many code smells often come together, and their combined effect increases software risk further. However, the direct connection between code smells and maintenance effort becomes less clear when other factors like code size and code churn are taken into account. This shows that code smells may be more useful as warning signs of poor software quality rather than direct causes of higher maintenance effort. Overall, the review shows that while code smells are important indicators of software risk, there is still no well-established unified model that predicts both defect risk and maintenance effort using smell severity. The primary barriers identified are inconsistent effort operationalization and the absence of standardized evaluation conditions.

Future research will focus on standardized severity operationalization, integrated datasets, and interpretable unified modeling frameworks to advance software quality analysis.

ACKNOWLEDGMENT

The authors gratefully acknowledge the financial support provided by the University of Moratuwa through the Senate Research Grant (Grant No. SRC/ST/2025/78).

DECLARATIONS

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper. All data analysed in this review were obtained from the published studies cited in the reference list.

DECLARATION ON GENERATIVE AI

Generative AI tools were used solely for language refinement and LaTeX formatting. All scientific ideas, analysis, results, and conclusions were developed by the authors, who critically reviewed all AI-assisted content and remain fully responsible for the manuscript's accuracy, originality, and integrity.

REFERENCES

[1] B. Boehm and V. Basili, "Software defect reduction top-10 list," in *Foundations of Empirical Software Engineering*, B. Boehm, H. D. Rombach, and M. V. Zelkowitz, Eds., Springer Berlin Heidelberg, 2005, pp. 426–431.

[2] M. Fowler, K. Beck, and J. Brant, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[3] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *J. Syst. Softw.*, vol. 86, no. 10, pp. 2639–2653, Oct. 2013.

[4] D. I. K. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.

[5] S. Haldar and L. F. Capretz, "Interpretable software maintenance and support effort prediction using machine learning," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, Lisbon, Portugal: ACM, Apr. 2024, pp. 288–289.

[6] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka, "Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains," in *2013 IEEE International Conference on Software Maintenance*, Eindhoven, Netherlands: IEEE, Sep. 2013, pp. 260–269.

[7] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, Lake Buena Vista, FL, USA: IEEE, Oct. 2010, pp. 390–400.

[8] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empir. Softw. Eng.*, vol. 17, no. 3, pp. 243–275, Jun. 2012.

[9] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *2012 16th European Conference on Software Maintenance and Reengineering*, Szeged, Hungary: IEEE, Mar. 2012, pp. 411–416.

[10] C. Politowski et al., "A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension," *Inf. Softw. Technol.*, vol. 122, p. 106278, Jun. 2020.

[11] N. Bessghaier, A. Ouni, and M. W. Mkaouer, "A longitudinal exploratory study on code smells in server side web applications," *Softw. Qual. J.*, vol. 29, no. 4, pp. 901–941, Dec. 2021.

[12] D. Cedrim et al., "Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, Paderborn, Germany: ACM, Aug. 2017, pp. 465–475.

[13] F. Arcelli Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowl.-Based Syst.*, vol. 128, pp. 43–58, Jul. 2017.

[14] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Raleigh, NC: IEEE, Oct. 2016, pp. 244–255.

[15] A. Gupta, R. Gandhi, N. Jatana, D. Jatain, S. K. Panda, and J. V. N. Ramesh, "A severity assessment of Python code smells," *IEEE Access*, vol. 11, pp. 119146–119160, 2023.

[16] A. Abdou and N. Darwish, "Severity classification of software code smells using machine learning techniques: A comparative study," *J. Softw.: Evol. Process*, vol. 36, no. 1, p. e2454, Jan. 2024.

[17] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empir. Softw. Eng.*, vol. 23, no. 3, pp. 1188–1221, Jun. 2018.

[18] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *2013 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA: IEEE, May 2013, pp. 682–691.

[19] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Trans. Softw. Eng.*, vol. 32, no. 4, pp. 240–253, Apr. 2006.

[20] A. Adewumi, S. Misra, N. Omoregbe, B. Crawford, and R. Soto, "A systematic literature review of open source software quality assessment models," *SpringerPlus*, vol. 5, no. 1, p. 1936, Dec. 2016.

[21] Z. Soh, A. Yamashita, F. Khomh, and Y.-G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities?," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Suita: IEEE, Mar. 2016, pp. 393–402.

[22] M. R. Islam, A. Al Maruf, and T. Cerny, "Code smell prioritization with business process mining and static code analysis: A case study," *Electronics*, vol. 11, no. 12, p. 1880, Jun. 2022.

[23] P. Piotrowski and L. Madeyski, "Software defect prediction using bad code smells: A systematic literature review," in *Data-Centric Business and Applications*, vol. 40, Cham: Springer International Publishing, 2020, pp. 77–99.

[24] 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 Trans. Softw. Eng.*, vol. 47, no. 1, pp. 17–66, Jan. 2021.

[25] 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," *J. Syst. Softw.*, vol. 144, pp. 450–477, Oct. 2018.

[26] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Inf. Softw. Technol.*, vol. 108, pp. 115–138, Apr. 2019.

[27] 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*, vol. 27, no. 4, pp. 1267–1296, Sep. 2020.

[28] L. Lavazza, G. Rotoloni, and S. Morasca, "Critical considerations on effort-aware software defect prediction metrics," in *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (EASE '25)*, Istanbul, Turkey: ACM, 2025, pp. 147–157.

[29] T. Sharma, "Multi-faceted code smell detection at scale using DesigniteJava 2.0," in *Proceedings of the 21st International Conference on Mining Software Repositories*, Lisbon, Portugal: ACM, Apr. 2024, pp. 284–288.

[30] A. Ho, A. M. T. Bui, P. T. Nguyen, A. Di Salle, and B. Le, "EnseSmells: Deep ensemble and programming language models for automated code smells detection," *J. Syst. Softw.*, vol. 224, p. 112375, Jun. 2025.

[31] D. Wu et al., "iSMELL: Assembling LLMs with expert toolsets for code smell detection and refactoring," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, Sacramento, CA, USA: ACM, Oct. 2024, pp. 1345–1357.

[32] G. M. Ubayawardana and D. D. Karunaratna, "Bug prediction model using code smells," in *2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer)*, Colombo, Sri Lanka: IEEE, Sep. 2018, pp. 70–77.

[33] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," *IEEE Trans. Softw. Eng.*, vol. 45, no. 2, pp. 194–218, Feb. 2019.

[34] V. Martins et al., "Eyes on code smells: Analyzing developers' responses during code snippet analysis," in *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software (SBES 2024)*, Brasil: Sociedade Brasileira de Computação, Sep. 2024, pp. 302–312.

[35] R. S. Rao, S. Dewangan, A. Mishra, and M. Gupta, "A study of dealing class imbalance problem with machine learning methods for code smell severity detection using PCA-based feature selection technique," *Sci. Rep.*, vol. 13, no. 1, p. 16245, Sep. 2023.

[36] D. Mahalakshmi, P. Kasinathan, D. Elangovan, C. R. Bhat, M. Balamurugan, and S. Sivakumar, "Code smell detection using hybrid machine learning algorithms," in *2023 5th International Conference on Inventive Research in Computing Applications (ICIRCA)*, Coimbatore, India: IEEE, Aug. 2023, pp. 633–638.

[37] C. Ni, K. Yang, Y. Zhu, X. Chen, and X. Yang, "Unifying defect prediction, categorization, and repair by multi-task deep learning," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Luxembourg: IEEE, Sep. 2023, pp. 1980–1992.

[38] Md. M. Rahman, T. Ahammed, Md. M. A. Joarder, and K. Sakib, "Does code smell frequency have a relationship with fault-proneness?," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, Oulu, Finland: ACM, Jun. 2023, pp. 261–262.

[39] G. Catolino, F. Palomba, F. A. Fontana, A. De Lucia, A. Zaidman, and F. Ferrucci, "Improving change prediction models with code smell-related information," *Empir. Softw. Eng.*, vol. 25, no. 1, pp. 49–95, Jan. 2020.

[40] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Inf. Softw. Technol.*, vol. 99, pp. 1–10, Jul. 2018.

[41] A. Santana, E. Figueiredo, and J. A. Pereira, "Unraveling the impact of code smell agglomerations on code stability," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Flagstaff, AZ, USA: IEEE, Oct. 2024, pp. 461–473.

[42] M. Tufano et al., "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1063–1088, Nov. 2017.

[43] E. Zabardast, K. Ebo Bennin, and J. Gonzalez-Huerta, "Further investigation of the survivability of code technical debt items," *J. Softw.: Evol. Process*, vol. 34, no. 2, p. e2425, Feb. 2022.

[44] H. Gupta, T. G. Kulkarni, L. Kumar, L. B. M. Neti, and A. Krishna, "An empirical study on predictability of software code smell using deep learning models," in *Advanced Information Networking and Applications (AINA 2021)*, L. Barolli, I. Woungang, and T. Enokido, Eds., Lecture Notes in Networks and Systems, vol. 226, Cham: Springer, 2021, pp. 120–132.

[45] H. Dai, J. Xi, and H.-L. Dai, "Improving effort-aware just-in-time defect prediction with weighted code churn and multi-objective slime mold algorithm," *Heliyon*, vol. 10, no. 18, p. e37360, Sep. 2024.

[46] A. Hassouna and L. Tahvildari, "An effort prediction framework for software defect correction," *Inf. Softw. Technol.*, vol. 52, no. 2, pp. 197–209, Feb. 2010.

[47] S. Haldar and L. F. Capretz, "Interpretable software defect prediction from project effort and static code metrics," *Computers*, vol. 13, no. 2, p. 52, Feb. 2024.