	Journal of Computing Innovations and Emerging Technology	JCIET
	Volume 1 | Issue 2 | December (2025)	Journal of Computing Innovations and Emerging Technology
	Published by **Novamind Press**	

How Code Smell And Refactoring Affect The Software Product Line Maintainability

Maryam Mehmood[1*], Asad Ijaz[2]

[1] *Software Engineering Department, National University of Sciences and Technology (NUST) and National University of Modern Languages, Pakistan*

[2] *Mechanical Engineering Department, University of Science and Technology, MUST AJK, Pakistan*

**Corresponding author: maryam.mehmood@numl.edu.pk*

Abstract* — Code cloning remains a significant challenge in modern software development, particularly within the Object-Oriented paradigm and advanced methodologies such as the Software Product Line (SPL) approach. In this context, code smells and refactoring can be seen as two sides of the same coin—one representing the symptoms of poor design, and the other offering systematic strategies for improvement. Among the various software quality attributes, maintainability stands out as a critical factor in determining the long-term success of SPL-based systems. However, the presence of cloned code directly impacts this maintainability, making the detection and mitigation of such clones an essential concern. Although multiple quality models exist to assess the relationship between code cloning, refactoring, and maintainability, most lack the granularity to accurately capture the specific effects of code cloning within SPL environments. This research undertakes a systematic literature review to consolidate and analyze findings from existing surveys, with a particular focus on identifying software metrics capable of evaluating the impact of refactoring on SPL maintainability. Refactoring serves as a deliberate means to eliminate code smells, and numerous tools and techniques have been developed to support this process. By synthesizing the current body of knowledge, this study provides a foundation for researchers and practitioners to better understand, select, and apply effective practices and tools to reduce code smells, improve maintainability, and ultimately enhance the overall quality of SPL-based software systems

***Keywords*—Code Clone, Code Smell, Maintainability, Refactoring, Software Product Line**

Manuscript received 14 Sep. 2025; revised 24 Sep. 2024; accepted 25 Sep. 2025. Date of publication 15 Des. 2025.

I. INTRODUCTION

Maintenance is very important for every software system. Authors [1] and [2] reported that about 50-80 % software costs are due to maintenance tasks, like fixing of faults related to software design and implementation, plat form changes in terms of hardware or OS and addition of new capabilities or alteration of existing functionalities. The definition of software architecture as described by Bass in 1999 is as follows: “The software architecture of a program or computing system refers to its structural design, encompassing the software components, their externally observable characteristics, and the interconnections or relationships that exist among them [3]-[4]. In recent passing years a new approach of software reuse has emerged and become popular in industry and among academicians SPL. The fundamental concept of SPL is to segregate the mutual parts of a product’s family from those parts of the product which are different. The common parts create a platform which is used to serve as mutual baseline for all the products related to same product family. Software Product Line (SPL) refers to the methodologies, techniques, and tools employed to develop a collection of similar software systems, built from a shared set of core attributes and produced through common development practices. With the development of source code, the rise in probability of duplication of the code becomes more noticeable and spreads throughout the various segments of the program. Such duplication of codes is recognized as Code

Clones or Smells [5]-[6]. Code smells violate the design principles of codin. They increase technical debt [7], affecting software maintenance [8]-[9], and evolution [10]. This is one of the major problems developers have to face during development of Software Product Lines which ends up affecting the software quality factors such as Maintainability, these further effects development costs negatively resulting in high cost and less efficient software [11]-[12]. Therefore, it is highly imperative to take necessary actions, right from the start of the SPL development, to address and control the code clones. One popular way to tackle the code clones is that of refactoring which effectively eliminates the code clones. Code clones help in the identification process of SPL refactoring. Refactoring is the process of altering the code of the software in a delicate way that its internal structure is improved while its external behavior remains exactly the same [13]-[15]. The three main steps of refactoring process [16] are (i) identification of refactoring candidates (bad smells or copied code), (ii) validation of refactoring effect (validation of refactoring candidates), and (iii) application of refactoring [17]-[18] at Fig 1. Refactoring normally causes minimal changes to the software; however, a refactoring can involve more refactoring. There are numerous benefits of refactoring, some of them are improvisation while designing the software, help improve the understandability of software, speeding up the programming process by helping locate the bugs easily, and minimize code duplication [19]-[20]. It becomes very clear that code clones largely effect the SPL quality which then adversely effects the factors affecting quality of software products, resulting in deterioration of performance of software and bugs/errors in software which increase cost due to ineffective maintainability. Various parameters for assessment of software product quality [21] particularly Maintainability are: Changeability, Testability, Stability, Analyzability and Maintainability Conformance. In such a case, it becomes imminent to utilize Quality Models as we know that they have become a de-facto and widely acceptable means to describe and manage the quality of software. There are a total of five methods which are used for quantifying software's maintainability [22]-[24]. They are: Hierarchal multi-dimensional assessment model: views software maintainability as hierarchy of source code elements, polynomial regression model: utilizes regression analysis to determine the association between software maintainability and metrics, aggregate complexity measure: analyzes maintainability of software as a function of its entropy, principal Component Analysis: is a technique that uses statistics to decrease co-linearity between common complexity metrics to identify and decreases the total number of components used to form the regression model, factor Analysis: is a statistical technique. In this technique the metrics are orthogonal zed so that they become unobservable factors, which are thereafter used for modeling the system maintainability. Even though there are many other quality models which are capable of quantifying maintainability aspect of software's development but all of them obviously lack the capability of assessing and establishing the relationship between refactored code and SPL maintainability [25]-[26].We have particularly chosen the quality factor of maintainability for software because it is one of the most crucial factors and in the long run the code clones in the source code inevitably affect the maintenance of the software [27]-[29]. Consequently, we pitch a model which would firstly analyze the impact of code clones on SPL maintainability and then after the refactoring has been performed, it would analyze the relationship and impact of altered code on the SPL maintainability, which will help us to draw our conclusion [30].

The word *smell* indicates some in depth issues in the software either at code level or design level. Code smell indicates the **violation** of fundamentals **of developing** a **software** that results in decreased quality of code. It is different from an error or bug. In other words, it is a *clue* that indicates something *might* get wrong or may lead towards negative consequences, and affect the software maintenance and evolution. A composite code or design smell is derived from one code smell that is connected with other code smell [22]. Categorizing a piece of code as code smell or not is subjective in nature and depends on different parameters like: language used for development, developers and development methodology [22]. **Error! Reference source not found.** shows some common code smells.

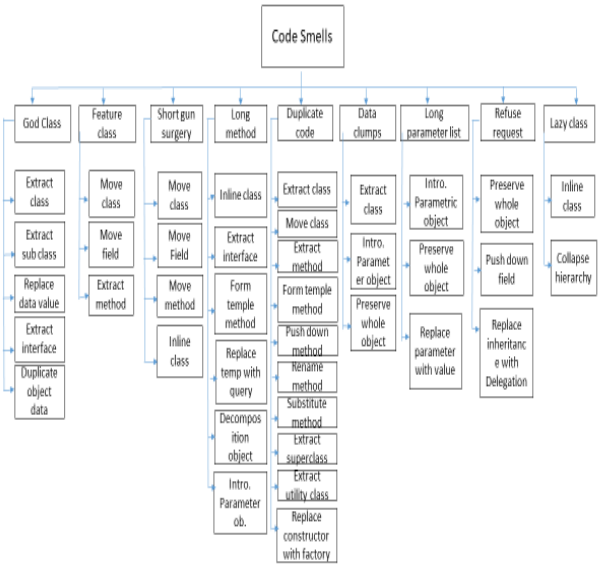

Fig 1. Code Smells

In Fig 1 there are some code smells are visualized and also there are some treatments to remove code smells from the code. As it can be seen that the problem of God Class can be removed with the help of extract class, extract sub class, extract interface, replace data value and duplicate object data. Same is the case for all the code smell types seen in 1st column and techniques to remove these smells in 2nd column. Code smells and refactoring are associated, since refactoring is crucial for removal of code smells by improving quality of code in terms of clarity, comprehension and simplicity. Also, if refactoring process is not followed properly, then it may produce new code smells and degrade the quality consequently. Together, these both impact the software quality. Developing quality software is very essential, but retaining or increasing the quality of software in maintenance is equally important. As code smells results in quality issues like high coupling, low cohesion, problems of encapsulation which effects maintenance and design decisions.

The remaining paper is divided into following sections: Section 2 presents a literature review related to Software Product Lines, Refactoring, Code Cloning and a few Quality

Models. The format we adopted for SLR, along with RQs, identification of related work, selection criteria, quality assessment, data extraction, and execution are described in Section 3. Section 4 demonstrates the motivation of this work performed. The findings and the proposed model, where relationship is established between refactoring and code smell on maintainability is discussed in Section 5. In Section 6 the main threats to validity of our study are discussed. Finally, we will conclude our findings and discuss possible future work in section 7.

D. Simon et al highlighted the importance of Software Product Lines (SPL) in managing large-scale software development across the industry. They argued that the advantages of SPL over traditional methods have significantly reshaped the perspectives of software companies, motivating them to adopt SPL practices even for their legacy products. In their study, the authors introduced a lightweight iterative process aimed at facilitating the gradual integration of product line principles into existing systems by applying feature analysis to legacy software. However, their approach deliberately avoids architectural reconstruction of legacy systems, which, although potentially beneficial, would result in higher costs.

R. Mitschke et al state that the main aim of SPLs is to promote reusability and the evolutionary process of common features to multiple products. To achieve this traceability is a necessity and therefore the authors have provided a proposal that each version of an artifact must be associated with a specific feature version and the dependence of the feature should also be managed explicitly. However, there is a big challenge which is faced by development team when developing SPL and that is to allow controlled evolution of SPLs.

J. Bosch et al elucidate that the Software Product Line (SPL) approach enhances software reusability while reducing development costs. This paradigm enables the sharing of architecture and reusable components across multiple products within a family. However, the authors emphasized that the evolution of SPLs is considerably more complex than that of traditional software development. This complexity arises from the continuous emergence of new and sometimes conflicting requirements, both from existing products within the SPL and from newly integrated ones. As these requirements grow and change, the number of features expands, ultimately increasing the overall complexity of the product line.

Gacek. C et al discussed that due to large members of SPLs it is imminent that they should support the description of SPL as a whole and also the instances of individual products derived from the product lines. They outlined that there are scalability issues in SPL approach and issues arise when combining and supporting various techniques which should be addressed and catered for properly and efficiently.

S. Apel et al discusses that Feature Oriented Software Development is a combo of differing ideas, methods, tools and techniques and not just a single development method. They state that it is a characteristic of product which is used for distinguishing software from a family of related software. The authors have presented an overview of Feature Oriented Software Development in their work and they have summarized various works done in the approach of Feature Oriented Software Development and have established relationships between different approaches of Feature Oriented Software Development

C. Kastner et al carried out an empirical study on Feature-Oriented Software Product Lines with a focus on code cloning. Their findings revealed that, although the Feature-Oriented Programming paradigm is intended to mitigate the cloning issues commonly found in Object-Oriented approaches, certain limitations result in a considerable presence of code clones within feature-oriented SPLs. Most of these clones are directly linked to feature-oriented programming itself. While refactoring techniques can be applied to eliminate such clones and improve the quality of the product line, the underlying factors that give rise to code cloning remain difficult to quantify, making it challenging to address them proactively

S. Schulze has primarily focused on code cloning analysis and their removal and on the reasons due to which code cloning occurs. He has also proposed a refactoring technique for preserving the variant nature of compositional SPL, in order to aid in removal of code clones. He concluded that although there is a prevalent problem of code clones in SPLs, the degree of harm that these clones cause in SPLs has not been accurately realized.

S. Thummalapenta et al have proposed in the study they published, an automatic approach for categorization of the evolution of code clone segments. Their study also attempts to inquire the reason due to which code clones continue to consistently propagate or evolve independently.

Heitlager argued that the effort required to maintain a software system is directly proportional to the quality of its source code. Furthermore, he noted that despite the extensive body of literature, no definitive methods exist for quantifying software metrics that can reliably assess software quality

G. Aldekon et al have used an SPL case study for measuring the maintainability index of each feature of the software developed using that particular SPL. The have also discussed ways of improving feature maintainability index and optimization of maintainability index for making design decision that will enhance global maintainability.

O. Panchenko et al have proposed the research of matric based quality indicators in order for assessing the most significant maintainability characteristics of software. The model used for quality was obtained from the goal questionnaire metric approach. The literature research was followed by expansion of quality model using standard metrics also some specially defined metrics. A few selective were validated by authors to foretell maintainability of software through experiments. They concluded with a note stating that metrics are very accurate indicators for

assessment of maintainability of a software. And it isn't necessary for all metrics to result either and still it is possible to describe the various aspects of maintainability using indicating metrics.

D. Coleman et al discussed how automated software maintainability can be used for guiding the process of software related decision making. They also applied, to 11 industrial software systems, metrics-based software maintainability models and used the results produced to improve the fact-finding method and selection of processes.

T. Thumm et al proposed a method to preserve the variants in SPLs so that the validity of all SPLs can be ensured after the refactoring process. The authors also presented the generalizability of this method for the annotative SPLs.

M. Kuhlemann et al introduced the concept of Refactoring Feature Modules of RFMs which basically provide extension through refactoring to feature modules. The study also concluded that RFMs decrease the number of incompatibilities and facilitate the reusability of modules. RFM also reshapes the program structure, which are composed of feature modules, automatically. To facilitate the decomposition and reusability of features and refactoring in RFMs a tool by the name of VAMPIRE is used. He argued that the effort required to maintain a software system is directly proportional to the quality of its source code. Furthermore, he noted that despite the extensive body of literature, no definitive methods exist for quantifying software metrics that can reliably assess software quality

Bashir et al implemented the MOMOOD quality model suggested by Rizvi et al. [13]. The model defines the formula for maintainability, which takes understandability and modifiability as arguments. The authors do not state how the metrics required for maintainability assessment were measured.

Wijayanayake et al worked on sub characteristics of maintainability. They measured the analyzability, changeability, resource utilization and time behavior for each participant in their experiment doing the refactoring. They also calculated the maintainability index, WMC, CBO, DIT, and LOC (see Table1.) for refactored code and code without refactoring.

Mehta et al focused on the maintainability index. The authors proposed an approach for the improvement of software quality attributes by elimination of relevant code smells from the source code of observed software project. To analyze the effectiveness of this, work the Maintainability Index (measured by the JHawk tool) and Relative logical complexity (measured by the Eclipse Metrics plug-in) are measured. According to the authors, the combination of maintainability index and relative logical complexity does better at estimating the maintainability of a software system than the maintainability index itself.

Sz″oke et al measured the refactoring impact on the software projects by the Columbus QM probabilistic software maintainability model proposed by Bakota et al. [30]. Quality characteristics mentioned in the ISO/IEC 25010 standard, are the basis of this model. The maintainability of the software project was measured by a tool named Source meter, developed by the authors of this work.

Reis et al reviewed 83 primary studies, showing that the most common detection approaches were search-based (30.1%) and metric-based (24.1%). The most studied code smells were God Class (51.8%), Feature Envy (33.7%), and Long Method (26.5%); only about 20% of studies incorporated visualization to support detection.

Zakeri-Nasrabadi et al analyzed 45 existing datasets and revealed critical limitations: datasets often suffer from imbalance, lack of severity levels, and a strong bias toward Java, with only commonly addressed smells like God Class, Long Method, and Feature Envy covered; several smells from Fowler & Beck's catalog remain unsupported.

Ali et al conducted an SLR highlighting evolving techniques in code smell detection and refactoring. They observed a shift from classic object-oriented paradigms toward approaches tailored for cloud, web, and mobile applications, underlining a growing need for automated, efficient detection and refactoring methods.

Lacerda et. al in a tertiary SLR explored relationships between code smells, detection approaches, refactoring tools, and quality impacts. It reported that refactoring tends to improve quality more effectively than merely detecting smells, mapped the top smells to their detection and refactoring strategies, and spotlighted 13 open challenges and unanswered questions in the field.

Recent 2025 studies in code smell detection have shown a remarkable shift toward transformer-based models, nuanced code representations, and domain-specific tools. Ali, Rizvi, and Adil introduced a Transformer-based approach for code smell detection, demonstrating how these models can significantly enhance detection accuracy by learning deep contextual features from source code.

In a complementary effort, researchers examined how code representation techniques including tree-based, token-based, and embedding formats affect machine learning detection of the God Class smell; their findings showed marked improvements in F_1-scores on the MLCQ dataset, paving the way for intelligent IDE plugins [31]

Parallelly, the EnseSmells framework, combining structural features with pre-trained language models, achieved 5.98% to 28.26% detection improvements across multiple smell types, underscoring the benefit of multi-faceted feature fusion [32]

Expanding detection beyond code, Oztas et al. (2025) tackled inline code comment smells using both augmented datasets and classifier models (notably Random Forest yielding 69% accuracy), providing a solid baseline for future comment-quality tools [33]

Addressing ML-specific project challenges, MLScent emerged as a novel static-analysis tool detecting anti-patterns and smells in ML codebases, including frameworks like TensorFlow, PyTorch, and scikit-learn—elevating code quality in data science projects [34].

An innovative study on test smells employed small open-source LLMs in multi-agent workflows, achieving near-guaranteed detection (96%) and enabling real-world refactoring; notably, pull requests generated via these workflows were accepted in open-source repositories, showcasing practical applicability [35].

II. METHODOLOGY

Code Smells Taxonomy

The term "**code smells**" was first introduced by Fowler and Beck in 1999, who proposed a list of 22 distinct code anomalies, often referred to as code fragrances. These can broadly be classified into two categories: dependence-based and similarity-based smells. At the class level, code smells are further categorized as those within a class and those that extend beyond a class.

In contrast, a **code clone** is a duplicate or near-duplicate fragment of code, often introduced through copy paste reuse. Clones are commonly classified into four types, ranging from exact copies (Type-1) to semantically similar fragments with different syntax (Type-4).

Code Smell Detection Techniques

Over the years, a variety of techniques have been developed to identify different types of code smells. These detection approaches rely on either raw source code or compiled code representations, which are analyzed for structural or semantic characteristics. Software metrics whether object-oriented or otherwise are commonly employed to correlate measurable properties of code with known code smell patterns. The required metric values are typically obtained through third-party tools or via static source code analysis.

Detection tools then process these metrics to determine whether certain code-smell conditions are met, providing the identified smells as output. However, static analysis alone cannot capture all instances of code smells, as certain issues emerge only at runtime (e.g., due to dynamic dispatch). To address such limitations, some approaches adopt hybrid detection techniques that combine static and dynamic analysis. Moreover, historical information about software evolution has also been leveraged to enhance the accuracy of smell detection.

Mapping study Process

To summarize the current research of assessing quality models for analysis of impact of code refactoring on software product line maintainability, we have performed a systematic literature review (SLR) [11], [36], [37]. SLR guidelines. Our review was performed in five stages (**Error! Reference source not found.**): Defining goal and Research Questions, Identification of Relevant Research Articles, Selection criteria, Quality assessment and then Data extraction at the end.

Fig. 2. SLR Design

Research questions:

RQ1: Are there any established software metrics available for the analyzing the impact of Code Refactoring on SPL Maintainability?

A software metric can be used to measure the code cloning problem as code cloning has an impact on maintainability of software quality and causes an increase in amount of work required. Multiple software metrics are used to measure different aspects of the system, before and after refactoring

RQ2: What are the Top ten refactoring techniques and their effects on the quality attributes?

Practically, it's difficult for the developers to spot refactoring opportunities, that is; to work out which sort of refactoring should be applied to mitigate a code smell. Studies reported that the association between refactoring and smells isn't a 1 to 1 relationship. This article presents top studies which have been more cited on refactoring techniques. The techniques which are more frequently used are the extraction techniques (method, variable, class) [6].

RQ3: What refactoring tools have been identified in literature?

Refactoring is performed by using some tools. It becomes difficult for developer to select the appropriate refracting tool. For this purpose, intense literature surveys are conducted by the developers. To overcome this issue, we have presented a precise survey which helps developer to select the best tool.

RQ4: What is the impact of refactoring and code smell on maintainability of software Product line?

To understand the impact of refactoring and code smells on maintainability, it is important to understand about the associations between refactoring and code with quality attributes and also available refactoring techniques used on quality attributes. Code smells and refactoring are associated, since refactoring is crucial for removal of code smells by improving quality of code in terms of clarity, comprehension and simplicity. Also, if refactoring process is not followed properly, then it may produce new code smells and degrade the quality consequently.

III. RESULT AND DISCUSSION

Refactoring on SPL Maintainability?

A software metric can be used to measure the code cloning problem as code cloning has an impact on maintainability of software quality and causes an increase in amount of work

required. Multiple software metrics are used to measure different aspects of the system, before and after refactoring. There are several techniques for finding code clones, some utilize tokens, strings and some use parse trees. Which technique is used depends on the goal of measurement.

Existing studies have not yet succeeded in quantifying the underlying causes of code clones in Feature-Oriented Programming, nor have they identified the factors leading to code clones in Delta-Oriented methodologies. Nevertheless, it is evident that code cloning adversely impacts the quality of Software Product Lines (SPLs), and these effects, in turn, propagate to the quality of the software products derived from such SPLs. The consequences of code cloning include; Downgraded efficiency, Creeping of bugs and errors into the software, Deteriorated performance, and Increase in cost due to poor maintainability.

RQ2: What are the Top ten refactoring techniques and their effects on the quality attributes?

We have selected top ten studies which have been more cited on refactoring technique. The techniques which are more frequently used are the extraction techniques (method, variable, class). Extract Class is used to detect smells like applied Duplicated Clones, Divergent Change, Data Clumps and God Class. Same refactoring can be used for detection of more than one smells, by taking context under consideration.

Extract Method, Move Method and Extract Class are the most commonly used than other refactoring techniques. The high interest of researchers in these techniques indicated the significant importance of these in the software industry. Extract Superclass technique is infrequently used. Although Add Parameter, Rename Field, Inline Temp, and Rename Method are commonly used techniques. But we have not found any studies which report opportunities and applications of these techniques. Instead, Rename Method is often used as automatic refactoring technique.

Practically, it's difficult for the developers to spot refactoring opportunities, that is, to work out which sort of refactoring should be applied to mitigate a code smell. Studies reported that the association between refactoring and smells isn't a 1 to 1 relationship.

Effects of Refactoring on the Quality Attributes:

Literature reports a process of refactoring for analyzing the effect on software quality attributes. Refactoring can be performed by following some basic steps. These steps are: (i) identification of suspected pieces of code that contains bad smells, (ii) determine refactoring methods that can be applied on the suspected code, (iii) selected refactoring method must not compromise on the software behavior, (iv) perform refactoring at required places, (v) examine the impact of refactoring on the software quality attributes, and finally (vi) perform comparison of code quality before and after refactoring in order to maintain quality.

Many studies have been performed to analyze the effect of various refactoring techniques (Move Method, Extract Class, and Extract Method) on the quality of code. It is reported that Extract Class has positive impact on some internal quality attributes such as: cohesion, inheritance, size and also have negative effect on the internal attributes like coupling and complexity. Extract Subclass impacts negatively on complexity and showed inconsistent impact on coupling and cohesion.

Inline Class method has negative impact on inheritance, and positive impact on cohesion, coupling, and complexity.

Extract Method affect cohesion, complexity, and size positively, and remains neutral on inheritance and coupling. Move Method has an opposite effect on complexity and coupling, and positive impact on cohesion. The Move Field refactoring technique effects cohesion in positive manners while coupling in negative. Complexity is positively and coupling and cohesion are negatively impacted in Encapsulate Field method. Replace Data Value with Object shows positive affects for cohesion and inverse impact for coupling. Lastly, coupling impacts positively by use of Replace Method with Method Object..

Based on the above research it can be concluded that the positive and negative impacts of refactoring on different quality attributes, helps the practitioners to select appropriate refactoring technique for elimination of bad smells of codes along with improvement of quality attributes.

RQ3: What refactoring tools have been identified in literature?

Refactoring is performed by using some tools. To answer this research question we explored different studies on tools of refactoring and among these studies different tools presented.

1) **JDeodorant** is an Eclipse plug-in that uses metrics and AST8 to automatically detect bad smells in Java code like Type Check, Switch Statement, Feature Envy, Long Method and God class. This tool is frequently used in studies to help the users to perform refactoring. The study analyzed that JDeodorant (by using default configuration) detects more bad smells as compared to PMD and inFusion tools. However, JDeodorant in detection of smells like: Long Method, God class and 8 Abstract Syntax Tree, has achieved low results in terms of precision and recall (about 14%). It is also observed that JDeodorant performs smell detection along with other applications of refactoring. This is the strength of this tool, which made it very popular even having some limitations

2) **TrueRefactor** is an automatic tool of refactoring. It uses Genetic Algorithm for selecting the optimal sequence which eliminates maximum code smells from the source code. For identification of code smells, source code is first parsed. Then structure of the software is shown by creating control flow graphs. For classification of code to explicit code smells, different metrics are calculated. An example program is discussed which contains artificially inserted code smells in

order to analyze the effectiveness of TrueRefactor. It measures (i) the no. of distinct code smells over specified iterations, and (ii) different quality attributes. Comparison between initial artifact and final revealed that this tool successfully eliminated significant number of bad smells. And also maintained important quality attributes with improvement. This tool can perform refactoring very well, but instead of this, its frequent current use is in the area of UML modification. Both JDeodorant and TrueRefactor, are the frequently cited and discussed in literature.

3) **Eclipse** is a popular tool used to support developers in process of automation of refactoring. The process starts with verification of prerequisites, then in depth analysis is performed and finally code is rewritten with the help of guidelines, with no compromise on the structure of AST. The benefit of using Eclipse is to make sure the application of refactoring. As Eclipse supports about twenty refactoring techniques, so it's up to developer to detect code smells and select appropriate refactoring technique. The authors have reported that on the basis of developer's habits, it is not an unimportant process. Now tools are considering these factors and recommending the developers different refactoring techniques.

4) **IntelliJ IDEA** supports about 40 refactoring techniques. It uses a lexical and syntax parser, namely Program Source Interface, to transform the source code into Abstract Syntax Tree. The parser validates the source code. After conversion, for verification of scope of changes, indentation adjustments in the code, insertion of blank-lines, change of qualifiers names and inclusion of libraries in the source code is the responsibility of Formatter. However, this tool uses built in Domain-Specific-Language in order to detect fragments in the Program Source Interface by using a distinct notation.

5) **Wrangler** is the tool used for refactoring of clones. It is implemented in Erlang and integrated with Eclipse and Emacs, with the help of ErlIDE plugin. For the programs of Erlang, this tool provides interactive refactoring. Wrangler supports different types of refactoring, detection of code smells, and mainly detection and elimination of code clones.

RQ4: What is the impact of refactoring and code smell on maintainability of software Product line?

To understand the impact of refactoring and code smells on maintainability, it is important to understand about the associations between refactoring and code with quality attributes and also available refactoring techniques used on quality attributes. First the process of refactoring is explained through a flow chart in Fig 3.

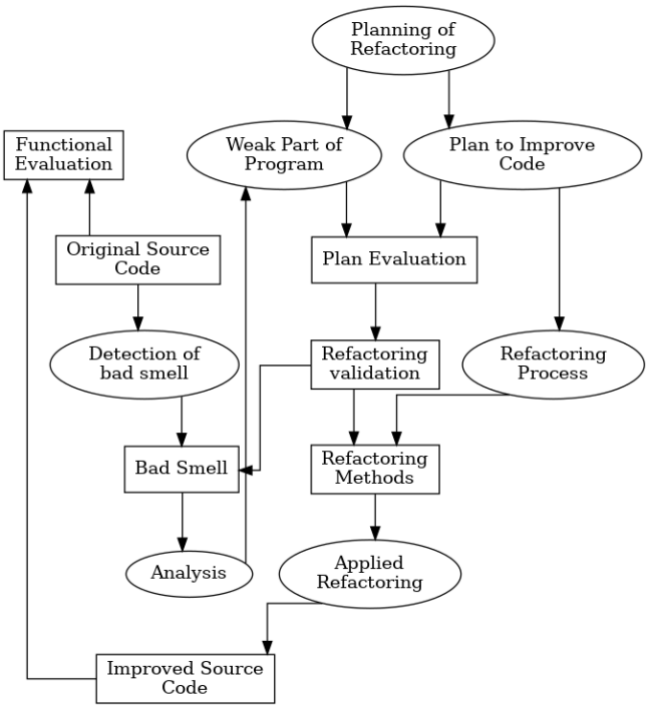

Fig. 3. Refactoring Process

Association between refactoring and the code external and internal quality attribute:

Different software quality models ISO/IEC 9126, FURPS, and McCalls Factor Model are reported in the literature and cited in the studies [[17], [38], [39]. Every model consists of different software quality attributes which are common in different quality models. These quality attributes are classified as internal and external attributes. The examples of internal attributes are coupling, complexity, cohesion, inheritance and size, while the maintainability, reusability, and understandability are the frequently studied external quality attributes. By using the combination of internal quality attributes (cohesion, inheritance, coupling, etc.) external attributes can be quantified. Thus, it is important to analyze the effect of refactoring on a single attribute rather than the combination of internal quality attributes. But the researchers conducted more studies to analyze the impact of refactoring on the external quality attributes as the experts are more interested in these attributes. The next most investigated issue of refactoring is the selection of code smells to be corrected based on its relative importance. Also, some studies identified and explored the relation of code smells type with the quality attributes. The type of identified relationship is different from author to author. Many code smells mentioned by Fowler et alcan affect multiple quality attributes, like maintainability, understandability and complexity, at the same time. These quality attributes have major influence on the software maintenance costs. Hence, the code smells which are related with these quality attributes will be given a highest priority for elimination from the software.

Similarly, some studies investigated that refactoring may produce negative effect on different software quality attributes. Doing changes in the code that doesn't need to be refactor, may result in low quality of the code instead of improvement. Therefore, refactoring doesn't guarantee the improvement of all software quality attributes.

Maintenance is one of the most essential features for software products. So far, we have seen quite a lot of researches about code smells, metrics, tools and techniques to remove these smells from Software Products line. There are many researches available regarding code metrics, techniques and tools. With the help of this review, we are able to identify that different tools are showing different results in different cases. They are incompatible for some scenarios. This systematic literature review helps us identify tools and techniques to minimize code smells. SPL (software Product Line) helps us build software products using software engineering methods, techniques, metrics and tools using collection of similar software systems from software assets using common production lines. This is one of the best approaches to reuse software products. It reduces cost and effort by reusing existing features and managing the variability between the different products with respect of particular constraints. With the help of this technique, we can reuse core assets of a company working on software product line. Code smells are the main issue when it comes to reusing of assets. To overcome the above-mentioned problem, we can move towards refactoring that helps to improve the internal structure of source code without disturbing the external behavior of the software product. Purpose of refactoring is to reuse software without the issue of code smells also it increases maintainability and helps improve quality of software product. It is basically restructuring the code by applying basic refactoring keeping in mind not to disturb internal structure of code so that it has no impact on the external behavior of software. There are so many code refactoring techniques available in literature. Ten most important techniques are described in this review article; Extract Class, Extract Subclass, Extract Method, Inline Class, Move Method, Move Field, Encapsulation Field Method, Replace Field, Replace Method, Rename Method. By removing code smells using refactoring we take source code having any sort of issues as an Input and produce a good quality code as an output. This output code can be reused in future software product development. We can identify code smell with the help of these refactoring techniques. We can also restructure code in order to remove code smell.

Software Product Line (SPL) refers to a collection of related software systems that share common features while also supporting variability, with the primary objective of maximizing reusability [20]. The SPL paradigm enhances software productivity and quality by exploiting similarities among systems and managing them within a reuse-driven framework. Inspired by industrial product line practices, SPL aims to reduce development costs and effort while improving overall efficiency. In software engineering, code smells are widely recognized as indicators of poor design choices or undesirable code characteristics. Similar to traditional systems, SPL artifacts can also exhibit various code anomalies. If these anomalies, or code smells, are not systematically addressed, the maintainability and quality of the SPL may deteriorate over time, particularly as the system evolves. Moreover, anomalies at the SPL model level can propagate across derived products, compounding the problem. While code smells are well-studied in conventional single-system development, their presence in SPLs represents a relatively new area of research. For instance, introduced the notion of "Variability Smells" specific to SPLs. Several refactoring tools have been developed to address such issues, including **J-Deodorant, True Refactor, Eclipse Refactoring, IntelliJ IDEA,** and **Wrangler**. This review highlights how practitioners can evaluate software maintainability by mapping widely used metrics to the tools that compute them. It also provides guidance for researchers and developers aiming to design or extend tools for emerging programming languages. Furthermore, the review identifies tools that support the calculation of popular maintenance metrics, outlines their language support, and points to open-source solutions that can serve as practical references for developing equivalent tools for diverse programming environments.

Threats to validity

This section is about the threats and some mitigations about the threats. The search string should be very well defined to get the precise results. First main issue in this SLR type research is the use of keywords to find the relevant topics. We have tried to select the best possible synonyms to get the max output. The chosen databases for the research is also a major concern. There is quite a big change that there are many relevant topics in other electronic repositories as well. Which is also a threat to validity to mitigate this threat we performed snowballing, where the citations of the selected papers are verified through a list of references to find other related studies not included initially on our search. All the topics selected for this SLR are written in English language, that doesn't mean there is no relevant topic in other languages.as we know the primary venue of scientific research is written in English language. So, we assume that our selected literature is enough to conclude a result. An investigation can be rehashed with similar outcomes. Our research can easily recreate following the means depicted and using the search strings.

IV. CONCLUSION

After the text edit has been completed, the paper is ready for the template. Duplicate the template file by using the Save As command, and use the naming convention prescribed by your conference for the name of your paper. In this newly created file, highlight all of the contents and import your prepared text file. You are now ready to style your paper; use the scroll down window on the left of the MS Word Formatting toolbar.

With the rapid expansion of Software Product Lines (SPLs), a significant number of code clones inevitably find their way into the source code. This not only leads to performance degradation but also increases the likelihood of bugs, errors, and higher maintenance demands. Code cloning is particularly prevalent in SPL methodology due to its emphasis on reusability. However, excessive cloning complicates maintenance efforts and directly undermines overall software quality.

To address this challenge, we have proposed a visualization-based approach to illustrate the effects of code cloning and the role of refactoring in enhancing SPL maintainability.

One limitation of this study is that the search strategy was limited to five databases (IEEE Xplore, ACM Digital Library, SpringerLink, ScienceDirect, and Google Scholar). Although these sources cover a substantial portion of the software engineering literature, additional databases such as Scopus and Web of Science may yield further relevant studies. Incorporating them would require re-executing the entire review protocol, which was beyond the scope of the current manuscript. We consider this an important extension for future work.

Our findings aim to provide researchers with deeper insights into effective practices and tools for mitigating code smells, thereby improving quality. Furthermore, we recommend future studies to explore the real-world objectives of refactoring as employed by industry professionals, assess its measurable impact on quality, and advance the development of intelligent refactoring tools capable of tracking and evaluating refactoring activities and their benefits over time.

ACKNOWLEDGMENT

The authors would like to thank the Department of Software Engineering, NUST/NUML, for providing support and resources during the course of this research.

REFERENCES

[1] W. LÖWE and T. PANAS, "RAPID CONSTRUCTION OF SOFTWARE COMPREHENSION TOOLS," *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 06, pp. 995–1025, Dec. 2005, doi: 10.1142/S0218194005002622.

[2] A. Telea and L. Voinea, "Visual software analytics for the build optimization of large-scale software systems," *Comput Stat*, vol. 26, no. 4, pp. 635–654, Dec. 2011, doi: 10.1007/s00180-011-0248-2.

[3] K. Alkharabsheh, Y. Crespo, E. Manso, and J. A. Taboada, "Software Design Smell Detection: a systematic mapping study," *Software Quality Journal*, vol. 27, no. 3, pp. 1069–1148, Sep. 2019, doi: 10.1007/s11219-018-9424-8.

[4] P. Hegedűs, I. Kádár, R. Ferenc, and T. Gyimóthy, "Empirical evaluation of software maintainability based on a manually validated refactoring dataset," *Inf Softw Technol*, vol. 95, pp. 313–327, Mar. 2018, doi: 10.1016/j.infsof.2017.11.012.

[5] M. Lafi, J. W. Botros, H. Kafaween, A. B. Al-Dasoqi, and A. Al-Tamimi, "Code Smells Analysis Mechanisms, Detection Issues, and Effect on Software Maintainability," in *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*, IEEE, Apr. 2019, pp. 663–666. doi: 10.1109/JEEIT.2019.8717457.

[6] 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," in *Proceedings of the 40th International Conference on Software Engineering*, New York, NY, USA: ACM, May 2018, pp. 482–482. doi: 10.1145/3180155.3182532.

[7] M. Misbhauddin and M. Alshayeb, "UML model refactoring: a systematic literature review," *Empir Softw Eng*, vol. 20, no. 1, pp. 206–251, Feb. 2015, doi: 10.1007/s10664-013-9283-7.

[8] 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 Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013, doi: 10.1109/TSE.2012.89.

[9] 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)*, IEEE, May 2013, pp. 682–691. doi: 10.1109/ICSE.2013.6606614.

[10] J. Erickson, *Impala Performance Update: Now Reaching DBMS-Class Speed*. Cloudera Inc, 2014.

[11] P. Kouros, T. Chaikalis, E.-M. Arvanitou, A. Chatzigeorgiou, A. Ampatzoglou, and T. Amanatidis, "JCaliper Search-based Technical Debt Management," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, New York, NY, USA: ACM, Apr. 2019, pp. 1721–1730. doi: 10.1145/3297280.3297448.

[12] Y. Mehta, P. Singh, and A. Sureka, "Analyzing Code Smell Removal Sequences for Enhanced Software Maintainability," in *2018 Conference on Information and Communication Technology (CICT)*, IEEE, Oct. 2018, pp. 1–6. doi: 10.1109/INFOCOMTECH.2018.8722418.

[13] O. Poy, M. Á. Moraga, F. García, and C. Calero, "Impact on energy consumption of design patterns, code smells and refactoring techniques: A systematic mapping study," *Journal of Systems and Software*, vol. 222, p. 112303, Apr. 2025, doi: 10.1016/j.jss.2024.112303.

[14] D. Ogenrwot, J. Nakatumba-Nabende, J. Businge, and M. R. V. Chaudron, "Empirical Investigation of the Relationship Between Design Smells and Role Stereotypes," Jun. 2024.

[15] X. Han *et al.*, "Code Smells Detection via Modern Code Review: A Study of the OpenStack and Qt Communities," *Cornell University*, May 2022, [Online]. Available: http://arxiv.org/abs/2205.07535

[16] R. Malhotra, A. Chug, and P. Khosla, "Prioritization of Classes for Refactoring," in *Proceedings of the Third International Symposium on Women in Computing and Informatics*, New York, NY, USA: ACM, Aug. 2015, pp. 228–234. doi: 10.1145/2791405.2791463.

[17] S. Vidal, I. berra, S. Zulliani, C. Marcos, and J. A. D. Pace, "Assessing the Refactoring of Brain Methods," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 1, pp. 1–43, Jan. 2018, doi: 10.1145/3191314.

[18] Y. Mehta, P. Singh, and A. Sureka, "Analyzing Code Smell Removal Sequences for Enhanced Software Maintainability," in *2018 Conference on Information and Communication Technology (CICT)*, IEEE, Oct. 2018, pp. 1–6. doi: 10.1109/INFOCOMTECH.2018.8722418.

[19] R. Malhotra, A. Chug, and P. Khosla, "Prioritization of Classes for Refactoring," in *Proceedings of the Third International Symposium on Women in Computing and Informatics*, New York, NY, USA: ACM, Aug. 2015, pp. 228–234. doi: 10.1145/2791405.2791463.

[20] O. Poy, M. Á. Moraga, F. García, and C. Calero, "Impact on energy consumption of design patterns, code smells and refactoring techniques: A systematic mapping study," *Journal of Systems and Software*, vol. 222, p. 112303, Apr. 2025, doi: 10.1016/j.jss.2024.112303.

[21] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow, "Code Smells Detection and Visualization: A Systematic Literature Review," *Archives of Computational Methods in Engineering*, vol. 29, no. 1, pp. 47–94, Jan. 2022, doi: 10.1007/s11831-021-09566-x.

[22] K. Borowski, B. Balis, and T. Orzechowski, "Semantic Code Graph An Information Model to Facilitate Software Comprehension," *IEEE Access*, vol. 12, pp. 27279–27310, 2024, doi: 10.1109/ACCESS.2024.3351845.

[23] N. R. Ravari, R. Latih, and A. Mohd Zin, "Multi-Language Program Understanding Tool," *Int J Adv Sci Eng Inf Technol*, vol. 13, no. 4, pp. 1554–1560, Aug. 2023, doi: 10.18517/ijaseit.13.4.18019.

[24] S. H. S. Almadi, D. Hooshyar, and R. B. Ahmad, "Bad Smells of Gang of Four Design Patterns: A Decade Systematic Literature Review," *Sustainability*, vol. 13, no. 18, p. 10256, Sep. 2021, doi: 10.3390/su131810256.

[25] G. Kaur and B. Singh, "Improving the quality of software by refactoring," in *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, IEEE, Jun. 2017, pp. 185–191. doi: 10.1109/ICCONS.2017.8250707.

[26] N. R. Ravari, R. Latih, and A. Mohd Zin, "Multi-Language Program Understanding Tool," *Int J Adv Sci Eng Inf Technol*, vol. 13, no. 4, pp. 1554–1560, Aug. 2023, doi: 10.18517/ijaseit.13.4.18019.

[27] 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," in *Proceedings of the 40th International Conference on Software Engineering*, New York, NY, USA: ACM, May 2018, pp. 482–482. doi: 10.1145/3180155.3182532.

[28] A. Rathee and J. K. Chhabra, "Restructuring of Object-Oriented Software Through Cohesion Improvement Using Frequent Usage Patterns," *ACM SIGSOFT Software Engineering Notes*, vol. 42, no. 3, pp. 1–8, Sep. 2017, doi: 10.1145/3127360.3127370.

[29] K. Solanki and S. Dalal, “Analysis of Research Trends Towards Types of Code Clone Detection Techniques,” *Indian J Sci Technol*, vol. 16, no. 7, pp. 468–475, Feb. 2023, doi: 10.17485/IJST/v16i7.2219.

[30] G. Kaur and B. Singh, “Improving the quality of software by refactoring,” in *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, IEEE, Jun. 2017, pp. 185–191. doi: 10.1109/ICCONS.2017.8250707.

[31] D. Shanmugasundaram, P. Arivukkarasu, H. Chen, and H. Cai, “Deep Learning Representations of Programs: A Systematic Literature Review,” *ACM Comput Surv*, vol. 58, no. 5, pp. 1–37, Apr. 2026, doi: 10.1145/3769008.

[32] E. Jabrayilzade, A. Yurtoğlu, and E. Tüzün, “Taxonomy of inline code comment smells,” *Empir Softw Eng*, vol. 29, no. 3, p. 58, May 2024, doi: 10.1007/s10664-023-10425-5.

[33] Md. A. Hossain, J. Jiang, J. Han, M. A. Kabir, J.-G. Schneider, and C. Liu, “Inferring data model from service interactions for response generation in service virtualization,” *Inf Softw Technol*, vol. 145, p. 106803, May 2022, doi: 10.1016/j.infsof.2021.106803.

[34] B. Zhang *et al.*, “A Comprehensive Evaluation of Parameter-Efficient Fine-Tuning on Code Smell Detection,” Jun. 2025.

[35] R. Melo *et al.*, “Agentic LMs: Hunting Down Test Smells,” Oct. 2025, doi: 10.1109/MS.2025.3621356.

[36] M. Lafi, J. W. Botros, H. Kafaween, A. B. Al-Dasoqi, and A. Al-Tamimi, “Code Smells Analysis Mechanisms, Detection Issues, and Effect on Software Maintainability,” in *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*, IEEE, Apr. 2019, pp. 663–666. doi: 10.1109/JEEIT.2019.8717457.

[37] S.-C. Necula, F. Dumitriu, and V. Greavu-Șerban, “A Systematic Literature Review on Using Natural Language Processing in Software Requirements Engineering,” *Electronics (Basel)*, vol. 13, no. 11, p. 2055, May 2024, doi: 10.3390/electronics13112055.

[38] R. Malhotra, A. Chug, and P. Khosla, “Prioritization of Classes for Refactoring,” in *Proceedings of the Third International Symposium on Women in Computing and Informatics*, New York, NY, USA: ACM, Aug. 2015, pp. 228–234. doi: 10.1145/2791405.2791463.

[39] V. Mehta, S. Bawa, and J. Singh, “WEClustering: word embeddings based text clustering technique for large datasets,” *Complex & Intelligent Systems*, vol. 7, no. 6, pp. 3211–3224, Dec. 2021, doi: 10.1007/s40747-021-00512-9.