DOI: https://doi.org/10.15276/hait.08.2025.11
UDC 004.4'2:004.451.44

Analysis of existing approaches to automated refactoring of object-oriented software systems

Mykola A. Hodovychenko[1)]
ORCID: https://orcid.org/0000-0001-5422-3048, hodovychenko@od.edu.ua. Scopus Author ID: 57188700773
Dmytro D. Kurinko[1)]
ORCID: https://orcid.org/0000-0001-8304-3257, dmitrykurinko@gmail.com
[1)] Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine

ABSTRACT

Automated refactoring plays a crucial role in the maintenance and evolution of object-oriented software systems, where improving internal code structure directly impacts maintainability, scalability, and technical debt reduction. This paper presents an extended review of current approaches to automated refactoring, emphasizing methodological foundations, automation levels, the application of artificial intelligence, and practical integration into CI/CD workflows. We examine rule-based, graph-based, machine learning–based (CNNs, GNNs, LLMs), and history-aware (MSR) techniques, along with hybrid systems incorporating human-in-the-loop feedback. The taxonomy of refactoring types is aligned with established terminology – particularly Fowler's classification – distinguishing structural, semantic (architectural), and behavioral transformations, all grounded in the principle of behavior preservation. Formal models are introduced to describe refactorings as graph transformations governed by preconditions and postconditions that ensure semantic equivalence between program versions. The paper provides a concrete example of a transformation generated by the DeepSmells tool, demonstrating the «before/after» change and explaining the rationale behind the AI-driven recommendation. The study also addresses the challenges of explainability and semantic drift, proposing mitigation strategies such as SHAP-based analysis, attention visualization in transformer architectures, integration with formal verification tools (e.g., SMT solvers, symbolic execution), and explainable AI recommendations. Special attention is given to the limitations of automated refactoring in dynamically typed languages (e.g., Python, JavaScript), where the lack of static type information reduces the effectiveness of traditional techniques. Generalization to multilingual systems is supported through models like CodeBERT, CodeT5, and PLBART, which operate over token-level, syntactic, and graph-based representations to enable language-agnostic refactoring. The paper also discusses real-world integration of automated refactoring into CI/CD environments, including the use of bots, refactoring-aware quality gates, and scheduled transformations applied at commit or merge time. Practical examples illustrate the verification of behavior preservation through regression testing or formal methods. This work targets software engineers, researchers, and tool developers engaged in intelligent software maintenance and automated quality assurance. By offering a consolidated classification, tool selection criteria, and practical scenarios, the paper delivers applied value for designing custom refactoring solutions or adopting existing technologies across diverse project constraints–ranging from safety-critical systems to large-scale continuous delivery pipelines.

Keywords: Automated refactoring; object-oriented programming; deep learning; software maintenance; code smells; graph neural networks; software engineering; code transformation; source code analysis; semantic code modeling; software metrics

***For citation*: Hodovychenko M. A., Kurinko D. D. "Analysis of existing approaches to automated refactoring of object-oriented software systems". *Herald of Advanced Information Technology*. 2025; Vol. 8 No. 2: 179–196. DOI: https://doi.org/10.15276/hait.08.2025.11**

INTRODUCTION, FORMULATION OF THE PROBLEM

Given the history of changing software and growing complex systems, the maintainability and adaptability of software source code have emerged as deciding factors for whether projects succeed or fail in the long run. As systems get larger, they accumulate technical debt (inefficient, redundant, or overly complicated code), which not only makes it less readable, reusable and scalable. Degraded internal software quality results in higher development costs, prolonged debugging time, and enfeebled efficiency [1].

Refactoring – transforming the internal quality of a code base without changing its function – has become a Linus for dealing with such concerns. Regular and disciplined refactoring ensures that developers are able to keep the code as clear as possible code smells and align its implementation to the latest design goals and architectural rules [2].

Nevertheless, traditional, manual refactoring in big object-oriented systems is time-consuming, prone-to-error and has high demand on developers' knowledge. In the agile and continuous delivery world, this challenge is further aggravated, as bug fixes are propagated with the next features, and there is usually no time to do deep work on the code. Accordingly, the need for automatic and semi-automatic refactoring tools and techniques is increasing continuously [3].

Automated refactoring speeds up the maintenance and guarantees consistency across different code bases, decreases human factor error rate, and promotes best practices. In the era of intelligent systems, machine learning and big code analysis, there are new opportunities to make refactoring more adaptive, context-sensitive, and developer-centric. Therefore, to re-evaluate and examine all existing automated refactoring approaches and to determine their strengths and weaknesses and to also forecast future promising research topics is important for the future software engineering community.

Refactoring automation is key for the efficiency, uniformity and long-term sustainability of software evolution tasks. Classical manual refactoring is based largely on developer experience and subjective reasoning whereas automated refactoring 'techniques can provide systematic, reproducible and scalable ways to improve the quality of code [4].

Among the main advantages of refactoring automation is the time savings that occur when the burden of locating and materializing code improvements is dramatically reduced, particularly where large and complex object-oriented systems are concerned. Automated refactoring tools like can easily identify structural problems (code duplication, long methods and deep inheritance trees), and propose suitable transformations based on sound software engineering principles.

In addition, automated refactoring guarantees the coherence of the codebase by removing the human element and forcing conventional best practices. It becomes a benefit for your continuous integration/continuous deployment (CI/CD) pipeline in no time by fitting nicely into your day-to-day workflow and giving you confidence in changes that you want to move safe and validated with minimal impact on your production systems [5].

Due to machine learning combined with intelligent code analysis, trends like context-awareness are pulling more and more into today's refactoring tools and allow refactoring tools to intelligently reason about code structure. Such evolution improves their capability to produce suggestions preserving the original program semantics and improving their maintainability.

Automated refactoring not only saves time and energy for developers, but also helps projects maintain health and long-term success with clean, modularity, and adaptability [6].

Thus, **the purpose of this research** is to analyze existing approaches to the automation of refactoring in object-oriented software systems, identify their strengths and limitations, and outline promising directions for further development and improvement.

To achieve this purpose, the following objectives have been defined:

1) to explore the theoretical foundations of software refactoring and its role in improving code quality;

2) to classify and analyze current tools and methods used for automated refactoring;

3) to evaluate the advantages and limitations of different refactoring automation approaches, including rule-based, heuristic, and machine learning-based methods;

4) to identify common challenges and open issues in the implementation of automated refactoring systems;

5) to formulate key trends and future directions in the development of intelligent and context-aware refactoring tools.

By addressing these objectives, the study aims to contribute to a better understanding of the current landscape of automated refactoring and provide a foundation for the design of more effective and intelligent solutions.

This survey paper is targeted at a wide readership consisting of both the academia as well as the industry. As a comparison analysis for the methods and tools available, the work supports software engineers and design tool developers to choose refactoring strategies according to system size, language, and project restrictions. For academics, the review organizes the trends in machine learning based refactoring, current open challenges (e.g., explainability and behavior preservation), and future directions of investigation.

The practical impact of the research is that it can help to design intelligent refactoring assistants, enable better integration in CI/CD pipelines, and shape the application of hybrid techniques that mix a rule-based precise approach with a learning-based more adaptable one. By combining a variety of techniques – from classical syntactic transformations to large language model–based suggestions–the paper also works towards narrowing the gap between research results from theory and practical software maintenance practises.

1. THEORETICAL FOUNDATIONS OF SOFTWARE REFACTORING

Refactoring is the function of restructuring code that already exists, but not to change its external behavior. Refactoring supports the process of

modifying software to correct these so that the resulting use of a software product is not compromised by the presence of remaining defects.

Martin Fowler is credited for the popularization of the concept of refactoring, which he defined as «a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior» [7]. This progress consists of a series of small transformations that do not change the behavior of the system but raise the level of design.

Refactoring is used to eliminate code smells (i.e., duplicated code, long methods, large classes), to reduce technical debt and to make way for future code extensions or code optimizations. It's a cornerstone of agile software development, where it's all about ongoing code quality maintenance in order for a system to keep high levels of quality as time passes by [8].

By relentlessly optimizing code structure, refactoring leads to better understanding of the codebase, easier debugging capabilities and more confident code modifications whilst forming the basis of a much healthier and scalable software architecture.

1.1. Goals and principles of refactoring

The refactored software keeps its external functionality, but internal structure is improved. This renovation is aimed to help make better readable code and to make it easier for other developers to bugifx and develop and/or support the code base and also serves to remove duplication and future proof the system for future development or extension [9].

Refactoring is not about fixing bugs or adding functions; it’s about improving the design of existing code to make it easier to understand and modify. By constantly refactoring a code base, developers can forestall the rot and technical debt which would otherwise accumulate resulting in deterioration in maintainability and longevity of the software product [10].

The major objectives of refactoring are:

1) maintain free + improved maintainability: the ability to make the code easier to read, understand and change;

2) increased re-usability: separating concerns into cohesive modules;

3) simplified complexity: simplification of control structures, breakdown of large methods and removal of needless dependencies;

4) greater uniformity: code hews to coding standards and architectural principles.

There are several basic principles behind the act of refactoring:

1) behavior preservation – should retain the outer (syntaxialogical) behavior of the program as is for each transformation;

2) tiny, safe transformations – refactoring happens in tiny, tested steps;

3) iterative development – refactoring should not be an add-on to the development process; however it should rather be a continuation of the development process;

4) automated TDD test suite – allowing me to refactor mess, without breaking everything.

By following these goals and principles, refactoring is a disciplined engineering practice that enables a cleaner, more maintainable and higher-quality software system.

1.2. Classification of refactoring types

Refactoring methods can be divided into different categories depending of their relative scope and type of edits. Structural, behavioral and semantic refactorings are the most typical classification. These categories are complementary for addressing specific facets of software quality and they have different objectives within the overall process of software maintenance [11].

Structural Refactoring. Structural refactoring is considered to enhance the organization and the appearance of the elements of the code preserving the syntactical and the working mechanism of the pro- gram. These changes focus on the structural properties of the codebase (e.g., modularity, encapsulation and dependencies).

Examples include:

1) picking out methods or classes to increase the level of the internal cohesion and diminish the code duplications;

2) migrating methods or fields between classes for better spread of responsibility;

3) we name variables, methods and classes, so that we can understand what you wrote.

Behavioral Refactoring. Behavioral refactoring is the process of modifying internal behavior of a component without changing its observable behavior to the rest of the software system [12]. They are frequently used as refactoring in order to achieve performance improvements, greater robustness, or the implementation of particular design patterns.

Examples include:

1) taking conditionals out of the equation with polymorphism;

2) changing the iteration routine preserving the output;

3) adding lazy init or cache for optimization.

Semantic Refactoring. Semantic refactoring are broader changes that have a direct impact on the meaning or semantics of the code at an higher level, usually with interactions across several companies or layers. Such changes maintain the systems' external behavior but potentially make a big difference for the semantic and design of code [13].

Examples include:

1) generalizing a class to extract a common superclass or interface;

2) removing ad-hoc implementations with domain-specific abstractions;

3) refactoring of imperative code to object oriented design.

This taxonomy is useful to realize impact and risk of various refactoring types, and can be used by software developers to choose the right techniques, given the goals and context of the software project.

1.3. Recognizing when to refactor: code smells and beyond

Identifying where and when refactoring are to be applied is an important aspect for retaining software quality. The reason is that since refactoring does not add new functionality or repair known bugs, it must be motivated by some indication of a design flaw or code rot. There are two main mechanisms for detecting refactoring such as code smells and code quality measures (Table 1).

In practice, the discovery of refactoring opportunities is typically driven by so-called code smells –observable characteristics of poor design. Based on the work of Fowler and Beck [7], these smells are heuristic instead of formal defects. They are most valuable in large codebases where you cannot search and read enough manually.

Typical smells include:

- Long Method – methods that have too much logic;
- God Class – classes that are doing too much;
- Feature Envy – methods stealing too much from other classes;
- Copy Paste, Shotgun Surgery, and others.

Though these categories are still fundamental, modern research indicates some drawbacks:

- Subjectivity and Developer bias – just because one developer sees it as a smell doesn't mean others won't see that as an acceptable design choice. The lack of domain context leads to false positives;
- Context-Free Detection – many static smell detection tools (e.g., PMD, Checkstyle) are context-agnostic and tend to report symptoms rather than diagnoses. For instance, performance-sensitive code might not mind a long method at all;
- Absence of Prioritization – traditional smell taxonomies do not provide measurements for seriousness and refactoring priority. This stands in the way of automated triage in largescale systems.

To deal with these problems recent approaches have suggested the integration of metric-driven and learning-based methods. Smells can be scored with structural indicators (e.g., cyclomatic complexity, coupling metrics) and contextualized using machine learning models developed on labeled datasets of «smelly» vs «clean» code.

Table 1. **Identifying the Need for Refactoring (Code Smells vs. Code Quality Metrics)**

Criterion	Code Smells	Code Quality Metrics
Definition	Symptoms in code that may indicate deeper problems	Quantitative indicators of internal code characteristics
Granularity	Typically, method or class level	Various levels – class, package, system
Detection Method	Heuristic or rule-based analysis (e.g., bad smells catalog)	Static analysis tools compute numerical thresholds
Interpretability	High – human-readable descriptions	Medium – often requires interpretation of numeric values
Tool Support	Supported in IDEs like IntelliJ, Eclipse	Broad tool support (e.g., SonarQube, Checkstyle)
Automation Potential	Moderate – requires human judgment for confirmation	High – can be automated in pipelines
Context Sensitivity	Low – does not consider overall design intent	Medium – reflects structural complexity but not intent
Main Limitation	Subjective and may vary between developers	Thresholds may not generalize across projects

Source: **compiled by the authors**

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

Furthermore, we are witnessing a move to behavior-informed smells where traces of runtime, history of code churn, and information on which code is more/less bug-prone can be used to correlate smells need with refactoring. For instance, a practice used for common bugs or poked at in every release might also seem deserving for analysis.

For this paper, we take a middle-of-the-road view: pre-existing smells are a good starting point, but refactoring decisions should be made by incorporating per-project quality signals, historical context, and the expected impact on maintainability. Automated systems should therefore transition from catalog odor detection to adaptive, learning-informed prioritization.

2. MODERN APPROACHES TO AUTOMATED REFACTORING

The challenge of maintaining code quality via manual refactoring is exacerbated by the growth in complexity and size of today's software systems, which has made it more and more difficult to keep code coherent, efficient, and free from defects. Facing this challenge, researchers and industry practitioners have proposed various techniques aiming at automating refactoring activities on object-oriented programs [15]. Those approaches differ by theoretical background, automation level, and capability of considering contextual and sematic information of code. This section structurally summarizes the four main classes of automated refactoring techniques: tool-based, algorithmic, AI-driven, and history-based (MSR–Mining SoftwareRepositories) approaches.

Tool-Based Approaches. Tool-based refactoring tools are typically built into modern IDEs e.g. IntelliJ IDEA, Eclipse and Resharper. These tools provide a set of predefined refactoring operations, e.g. renaming, method extraction, and class moving, that can be performed either by graphical menus or using shortcuts [16]. While useful, these tools are generally confined to surface, syntax-level refications and are heavily dependent on the developer's guidance to abstract and confirm changes. They offer inferior type of context reasoning and are not able to self-suggest structural reorganization.

Algorithmic Approaches. The goal of algorithmic approaches is to automatically refactor code according to rules which might be around code smell, design patterns, graph representations and static analysis. Such methodologies generally consider the structural aspects of the code using concepts such as abstract syntax trees (ASTs), software metrics, and design smells to recognize the opportunities for transformation. Graph representation allows reasoning about structural properties, and it became possible to find refactoring candidates at scale. Although deterministic and interpretable, these frameworks are limited when it comes to adaptation and semantics because they are grounded into a rule-based representation.

AI-Driven Approaches. With the advent of large code repositories and advanced machine learning models, AI based approaches have started becoming an alternative to intelligent refactoring AI techniques [17]. Such approaches (meta-) learn from supervised and unsupervised learning models such as deep neural networks, graph neural networks (GNNs), transformers, and code embeddings to detect code smells, predict refactoring operations, and recommend transformations with low-level of human effort. Tools like DeepSmells, MoveRec, and frameworks built upon CodeBERT have been successful at discovering relevant patterns in source code and evolutionary traces. However, the strategies are often used as the black-box methods and it is hard to interpret one's recommendation or to guarantee the correctness of behavior.

History-Based Approaches: The MSR Challenge. Another important class is mining software repositories (MSR) and finding patterns from previous refactoring activities. Through reviewing commit history, pull requests, and version control logs, these techniques are able to recognize recurring sequences of transformations and to draw conclusions from experienced developers based on the transformations they performed. RefDiff and RefactoringMiner are examples of this kind of tools that identify performed refactorings to guide next recommendations. These approaches offer utility in reality and explainability but are very sensitive to the availability and quality of version history [18].

Combined, these four classes of mechanisms provide complementary strengths: tool-oriented approaches are strong in terms of usability, algorithmic ones offer structural reasoning, AI-based techniques promise adaptivity and intelligence, and history-based approaches anchor their recommendations in empirical developer behavior (Table 2).

Table 2. **Modern approaches to refactoring automation**

Criterion	Tool-Based	Algorithmic	AI-Driven	History-Based (MSR)
Foundation	Manual or semi-automated transformations via IDEs	Rule-based, pattern-driven, or graph-based algorithms	Machine learning and deep learning models	Mining software repositories for historical changes
Automation Level	Partial – requires developer interaction	High – can automatically detect and suggest changes	High – learns from data to suggest transformations	High – learns patterns from actual refactoring commits
Context Awareness	Low – limited to local code structures	Medium – some structural reasoning via graphs	High – can incorporate multiple contextual signals	Medium – based on past developer behavior
Semantic Understanding	Low – lacks deep semantic analysis	Medium – syntax-level or structural inference	High – models can infer semantics from examples	Medium – indirectly captures semantics via evolution
Explainability	High – user can inspect changes before applying	Medium – defined rules or patterns can be traced	Low to Medium – often black-box behavior	Medium – examples from history are interpretable
Scalability	Moderate – dependent on IDE capabilities	High – algorithms can handle large codebases	High – scalable with sufficient computing resources	High – can scale with project size and history depth
Adaptability	Low – static and rule-based actions	Low – rigid rules, difficult to adapt to new contexts	High – models can generalize to new cases	Medium – tied to availability of version history
Tool Support	IntelliJ IDEA, Eclipse, ReSharper	RefactoringMiner, Designite	DeepSmells, MoveRec, CodeBERT-based systems	RefDiff, RefactoringWatcher
Main Limitation	Limited to predefined operations, not project-aware	Hard-coded rules reduce flexibility and learning	Lack of transparency and need for large training data	Dependent on quality and completeness of repository data

Source: **compiled by the authors**

3. AI-BASED AUTOMATED REFACTORING OF OBJECT-ORIENTED CODE

3.1. AI in Code Refactoring: Background and Trends

Refactoring is the process of restructuring of an existing computer program without altering its external behavior.

The traditional refactoring tools depend on manually-developed heuristics, and static analyses, where as in recent years, the rapid progress has been observed in AI-based approaches to automate or help refactoring in object-oriented (OO) systems [19].

From 2020 to 2025, machine learning (ML)/deep learning and large language models (LLMs) have gained more attention from researchers for analyzing code for «code smells» (design weaknesses/bugs) and for recommending (or even applying automatically) refactoring changes. This overview provides a comprehensive picture of the current state-of-the-art of such AI-based refactoring techniques in terms of key contributions, followed methodologies, evaluation strategies, and outstanding challenges.

Key Trends: Initial AI refactoring research has targeted supervised ML for the detection of code smells or the classification of refactoring types. For instance, some researchers apply ensemble ML (e.g. with XGBoost) for classification of the specific kind of refactoring (like Rename Method or Move Method) to the commit message text used in the refactoring operation [20].

With the availability of data (e.g., mining Git commits for known refactorings) deep learning became more popular. With the recent advancements in the NLP community, researchers started looking at neural models that are trained on code syntax and

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

structure rather than handcrafted features. Deep learning architectures, such as convolutional (CNN), recurrent (RNN/LSTM) and very recently graph neural networks (GNN), have been used to identify code smells (e.g., Long Method, God Class, Feature Envy) and recommend code refactoring recommendations. By 2023-2024 the arrival of strong code-based LLM applications (e.g. CodeT5, GPT models) prompted new directions: fine-tuning pretrained models to produce refactored code or utilizing prompt-driven LLMs for refactoring proposals.

Hybrid AI approaches that combined deep learning with search-based optimization or human-in-the-loop feedback were also investigated by researchers. The key new findings in these areas are presented below.

3.2. Machine Learning for Code Smell Detection and Refactoring Recommendations

Some recent research works are using ML classifiers for detecting code smells and suggesting remediation change patterns [21]. A bad smell like Feature Envy or Long Method in the code can be a sign of respect for refactorings (ie moving method, extracting method). It also replaced traditional detection which was based on metric thresholds or rules of heuristics. Recently, researchers in the community proposed to instead train ML models on labeled examples as smelly vs. clean code to detect such patterns. The smell detection accuracy has also been enhanced with ensemble learning, for instance, some authors combines boosting and bagging classifiers (with feature selection and balancing) led to 97–99 % accuracies on smell datasets such as Blob Class and Long Parameter List [22].

The use of ensemble methods is motivated to handle class imbalance and overfitting problems frequent in smell datasets. In a more refined manner, advanced models use the deep learning approach for automatic feature extraction code. Some authors introduced DL approach for code smell detection. Such model obtained higher precision than traditional ML models [23]. Similarly, some authors proposed DeepSmells, it combines a CNN and an LSTM network to detect various types of code smells in Java code snippets [24]. The CNN is used to learn local structural patterns from the source code and the LSTM captures longer-range sematic context with a final dense network determining whether or not a snippet is “smelly”. DeepSmells model achieved better F1-scores for many smells with respect to previous techniques according empirical results.

Another promising direction is utilizing graph-based learning. The code itself has natural graphs (ASTs, control-flow, and call graphs), GNNs can encodes significant relational information for smell detection. Authors of paper [24] used a GNN to detect Long Method and Blob classes and found that their approach achieved higher accuracy than treating source code as flat text.

A more sophisticated example in terms of Feature Envy is paper [25], where authors introduce two GNN-based approaches: SCG and SFFL.

SCG (SMOTE Call Graph) models have the envy detection task to frame as a binary edge classification on a method-call graph, based on predicting the “calling strength” of method intersection class relationships; a method “belongs” in another class it is recommended to be moved to if that edge weight is highest.

SFFL (Symmetric Feature Fusion Learning) model uses heterogeneous graphs to model various relations (method–method calls, method–class ownership, etc.) and applies link prediction to recommend a new class for a greedy method such that a refactored ownership graph is directly constructed.

These GNN methods not only identify where the smell occurs but also suggest an actual refactoring. Experiments on real-world open-source projects show that our approach not only achieves better detection precision, but also can provide developers meaningful refactoring suggestions over existing heuristic methods.

Example of refactoring generated by DeepSmells. To make the practical application and cleverness of the AI-enabled refactoring more concrete and self-explanatory, we consider an example of a change proposed by a powerful DeepSmells tool, whose behavior is based on a CNN + LSTM deep learning model implementing a code smell detection and correction by analyzing an existing code that needs to be arranged.

DeepSmells Suggestion: Extract Method

```
public void generateReport(List<Employee>
employees) {
    DecimalFormat df = new
DecimalFormat("#.##");
    double totalSalary = 0.0;
    for (Employee e : employees) {
        System.out.println("Name: " +
e.getName());
        System.out.println("Position: " +
e.getPosition());
        System.out.println("Salary: $" +
df.format(e.getSalary()));
        totalSalary += e.getSalary();
    }
    System.out.println("Total salary: $" +
df.format(totalSalary));
}
```

Upon its learned representation of common refactorings, DeepSmells therefore suggests extracting the body of the loop into a new method, increased read- ability and cohesion and less cognitive load for the calling method.

Code after Refactoring

```
public void generateReport(List<Employee>
employees) {
    double totalSalary =
printEmployeeDetails(employees);
    DecimalFormat df = new
DecimalFormat("#.##");
    System.out.println("Total salary: $" +
df.format(totalSalary));
}

private double
printEmployeeDetails(List<Employee> employees) {
    DecimalFormat df = new
DecimalFormat("#.##");
    double total = 0.0;
    for (Employee e : employees) {
        System.out.println("Name: " +
e.getName());
        System.out.println("Position: " +
e.getPosition());
        System.out.println("Salary: $" +
df.format(e.getSalary()));
        total += e.getSalary();
    }
    return total;
}
```

Satisfying important behavioral invariants for such transformation are:

- output remains unchanged;
- internal state (salary sum) is preserved and returned correctly;
- the code is now modular: logic for presentation and aggregation is encapsulated in a single-purpose method.

Although DeepSmells does not provide human-readable justifications for what suggestion is being made by the models, the motivation for suggesting that lies within the range of established thresholds for method length and fan-in complexity, which the neural model implicitly learns during training data.

3.3. Deep Learning and Refactoring Recommendation Systems

In addition to finding the smell of dead code, researchers have now also built recommendation systems that are able to suggest directly where to refactor old code and specifically which refactorings to apply.

For example, in object-oriented (OO) programs, it is a common requirement to move methods from class to another. This refactoring is even provided as an automated suggestion by JDeodorant when it finds «envy» in one class.

There are already a number of works trying to automate MMR recommendation using deep-learning methods.

For example, *MoveRec* is an approach that combines deep neural networks with knowledge derived from LLM. MoveRec uses a hybrid model that combines CNN, RNN and GRU to judge when and where to move methods.

To cope with the unique shape of this NLP output, MoveRec augments input features with textual summaries generated by an LLM (GPT-based, in this case) so that the method's meaning and surrounding context are recorded. Static code metrics are taken into account, such as the similarity of domains between source and target files. Early trained on 12,475 examples out of 58 Java projects, MoveRec achieved an average F1score of 74 %, far surpassing earlier heuristics like JDeodorant and previous machine-learning tools (which typically improved F1 by 9-53 %). This shows that combining deep learning with LLM insights can significantly improve refactoring proposals' accuracy.

RMove and Other Moving Methods (2021-2023) – former attempts to recommend Move Method were mainly represented by simple ML or program analysis. For instance, authors of papers [26] used a random forest to analyze code dependency graphs, and authors of work [27] employed path-based embeddings (Code2Vec) to rank candidate moves.

These projects showed their feasibility but not their quality. With the advent of deep models such as MoveRec, which take semantic understanding into consideration, however, this situation has been drastically changed and is producing a much more efficient way for MMR recommendation.

Refactoring in Multi-Objective Context – some research groups have integrated refactoring with other purposes, such as performance. These were the kind of issue authors addressed in their paper "MovePerf: efficient detection of refactoring performance faults using deep learning" [28]. They constructed a specialized deep learning model with historical data to predict a program's execution time after refactoring move to method level. For MovePerf, the architecture consists of a deep feedforward neural network and a factorization machine (FM), able to detect both low-level and high-order feature interactions among code metrics. Training data was obtained by refactoring Java microbenchmark projects and benchmarking their performance using JMH (Java Microbenchmark Harness). With a feature set of 32, the model had a mean error (MRE) rate around 7.7 %, and it outperformed baseline predictors such as CNN or DeepFM.

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

Other Refactoring Types – there are less AI-based systems for refactorings such as Extract Class or Extract Method, but researchers have been studying them. The 2022 article [29] proposes an automatic Extract Method refactoring technique handling a long method. Their method searches for what blocks of code to pull out using program slicing (not ML), in order to avoid introducing new smells. Turning to the field of machine learning, some scholars have created models to predict refactoring sequences – that is, which series of refactorings should be applied.

For instance, in a 2025 review paper [30], a number of researchers optimized refactoring schedules by genetic algorithms or even deep reinforcement learning for multiple smells.

Such meta-heuristic methods search over refactoring as a problem of selection (balancing various quality metrics), and if combined with learning-based models or heuristics they can indicate a sequence of haven-introduced rather than just giving you one.

3.4. Large language model and automatic refactoring

Large language models trained on code have opened up new space for automated refactoring. Two main approaches have emerged: LLMs (based on ChatGPT, for example) code modifications in some ways that make specially refactored code prompt and general LLMs become refactoring engines.

Fine-Tuning and RL for Refactoring: Authors of [31] recently introduced a model to generate refactored code using sequence-to-sequence and reinforcement learning. They made paired examples of pre- and post-refactored code using actual refactoring edits mined from Git (e.g., before and after an Extract Method refactoring), of which there are thousands.

Based on these examples, they fine-tuned code generation models like CodeT5 and CodeGPT and then reinforced them to learn transformations.

The model learns to output the refactored version of a given code fragment. Preliminary results indicate that Refactoring Engines can correctly apply numerous Extract Method refactorings. However, this approach must guarantee the behavior remains the same, something traditional refactoring engines generally do with robust precondition checks.

ChatGPT and Interactive Refactoring: Another line of research views refactoring as an interactive AI-assisted task. Authors of the paper [32] conducted an empirical study on ChatGPT's capability to refactor code using 40 Java code segments and prompts that pinpoint particular quality characteristics.

They requested that ChatGPT would code refactor in order to raise different quality aspects (readability, efficiency). Afterward they assessed the results. The study found that is 39 out of 40 cases, ChatGPT produced improved code. Improvements might be substantial – say clearer naming, removing redundancies, or even sometimes changing data structures to make the design better.

Even more impressively, ChatGPT was able to maintain the same general behavior as the original code did in 311 of 320 cases. 90% percent of the changes it made, the system produced accurate self-explanatory commit messages. But the authors comment that there are some caveats: ChatGPT's output is not consistent; indeed, from one prompt several possible refactoring solutions result, with only a few optimal. It made various modifications that were not just incomplete or incorrect (due to lack of the full context) but also had no counterpart in the original input. Given this, their results stress how powerful LLMs might be as refactoring assistants but, especially because of the occasional anomalous solution they suggest, you still need some human review [33].

ChatGPT as a collaborator is given some attention in subsequent studies: prompting it, for example, to produce the test cases that cause refactoring engine bugs or to perform specific code smell refactoring, such as Data Clumps.

For the Data Clumps refactoring, authors of paper [34] designed an AI-driven pipeline which employed ChatGPT to locate groups of variables that should be encapsulated into new objects. Their pipeline locates such code smells automatically and suggests the introduction of a new data structure (a kind of "Extract Class" refactoring) by code changes that are provided by the LLM.

In the end, LLMs bring a level of understanding and creative power, complementing traditional refactoring techniques. They are particularly good at making global improvements and justifying their changes (in commit messages or comments), but expecting uniformity, safety, and adherence is still an open question.

3.5. Cross-Language Challenges and Dynamic Typing Constraints in Refactoring Automation

While most existing solutions for automated refactoring have been conceived and tested in the realm of statically typed object-oriented languages, predominantly Java, the extension of these

techniques to dynamically typed or multi-language settings opens unexplored territories in software product lines research.

In languages like Python and JavaScript, which have dynamic typing, many of the assumptions that classical refactoring holds don't stand? No explicit type, late binding, reflection, not to mention metaprogramming all made static analysis process, code smell detection, and transformation precondition construction complicated. For example, identifying feature envy and method movement opportunities may be unreliable, because of the instability of the call graph and the ambiguity of dynamic dispatch.

In addition, refactorings like Rename Method, Extract Class, and Move Function need more context in their runtime environments to work correctly. Dynamic analysis, test-based inference and runtime instrumentation are techniques that are being used more and more in such environments as complementary strategies. However, they tend to suffer from performance overheads and low coverage that impede the applicability to large-scale codes.

Recent work tries to alleviate these problems by using probabilistic reasoning and learned type inference. For instance, code-corpora-trained neural models (e.g., CodeT5+, GraphCodeBERT) may estimate type information or data flow relations in untyped scripts that are not exactly the same as in typed scripts, facilitating partial transfer of refactoring techniques from typed to dynamic languages.

In the context of software development, modern software projects tend to combine libraries written in many different languages (e.g., Java + Kotlin + Javascript). In this regard, language-independent materializations, such as a sequence of tokens, an abstract syntax tree or a graph-based intermediate form (e.g., universal abstract syntax graph) have been proposed as candidate representations for cross-language refactoring tool support.

Multiple models (e.g., CodeBERT, PLBART, and RefT5) have been pre-trained on multilingual data covering Python, JavaScript, C++, etc. These models can be used to recognize refactoring patterns for multiple languages and even suggest consistent changes across modules written in different languages. Yet, idiomatic refactorings need to be reconciled across language boundaries and coherent behavior must be guaranteed across various runtime systems.

To conclude, the scope of automated refactoring techniques beyond the Java-centric ecosystems must continue to develop methodologies to work at the levels of dynamic semantics, multilingual representation, and hybrid static–dynamic analysis pipelines. We argue that addressing these limitations is essential for creating refactoring systems that are really general-purpose and not biased towards any language or language set.

3.6. Challenges and open issues

Though it has made strides, AI-driven refactoring faces several challenges and unsolved research questions.

Training data quality and bias. Supervised approaches require good datasets of existing smelly code or refactorings. Mining git histories could bring bias (e.g., different types of refactoring only happened to be recorded in commits). Class imbalances remained an insurmountable problem– compared with non-refactoring changes, examples of refactoring are few – until tens of thousands had been artificially created using methods like data augmentation, SMOTE and transfer learning.

Semantic Preservation and Validation. ML/LLM approaches don't deliver a priori guarantee about behavior-preserving changes as traditional tooling does. ChatGPT's ability to preserve behavior (97 %+ in tests) reassures people, but there are costs when things go wrong. A small, seemingly innocuous change could bring in subtle bugs. Integrating verification steps (such as running test suites & otherwise using formal checks) into the AI refactoring pipeline is an open research direction. Some works propose keeping a "human-in-the-loop" – the human decision maker, for example, might ensure that LLM-suggested changes comply with safety and regulatory requirements [35].

Tellability and Developer Confidence. Developers are more likely to trust AI recommendations if the tool can explain why a refactoring is necessary or how this will help. While some ML models are black boxes, others try to produce explanations (e.g., by outputting a rationale or by generating commit messages in the way that ChatGPT does. More work is under way on making refactoring recommenders more interpretable; perhaps by highlighting the code smell symptoms they detected, the expected improvement in metrics.

Here is a general problem that learning-based refactoring tools have so far overlooked: *most of them merely focus on one particular bad smell or refactoring operation* (moving methods around, making functions longer etc.). How do we construct a unified framework that is capable of suggesting all

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

sorts of adjustments to any given codebase, or deciding between different proposed fixes for a particular smell? Some optimization-based approaches solve this by evaluating multiple options (e.g., a GA might choose to either extract a class or move methods when dealing with big God Class smells). Combining multi-step reasoning (which LLMs can do a bit already) with fine-grained low-level transformations points to important future directions for our research efforts [36].

Cross cultural and multilingual support. Most early studies focused on Java (the «canonical» object-oriented language in refactoring research). Later work, such as RefT5, explicitly targeted multilingual projects (Java and Python). This model demonstrates that you can train a network to recognize refactoring activity in one language, and it will then automatically generalize to others. This finding is hopeful, in that real systems often combine different languages. Nevertheless, more work is needed for languages with greatly different paradigms and fewer refactoring datasets (e.g., C++ or dynamic languages).

Integrating these AI-based tools into a development environment and continuous integration and development process is another practical problem to be solved. Some research prototypes like Stack-IDE have IDE plugins that suggest refactorings as you code. How smoothly such tools run, and the extent to which they can provide nonintrusive, helpful suggestions at appropriate times, will determine their acceptance into industry.

In summary 2020-2025 has been a pivotal period for this kind of refactoring of object-oriented code, ever more so as AI makes it happen. The shift from simple «data clubbing» experiments pure and simple logistic models of software input/output, to the development of high-abstraction event-oriented systems which are largely automatic is soon well under way.

3.7. Models, Invariants and Refactoring in Automation of Refactoring

One of the most important criteria to look for in both automated and manual approaches to refactoring is maintenance of program behavior. The meaning of the program must be preserved under transformation. This principle, first articulated by Fowler, and in the ascendancy with AI-backed refactoring tools and IDEs, holds rule that there shall be no regression from a structural improvement.

Formal representation of refactoring transformations. Refactorings can be described by formal graph transformation systems or rewrite rules on/from abstract syntax trees.

A software system S can be defined as a tuple:

$$S = (C, D, F \tag{1}$$

where C denotes the set of code components (e.g., classes, methods); D is collection of data dependencies;

F is the collection of control flows.

A refactoring is a transformation function:

$$T: S \to S', \tag{2}$$

such that the semantic preservation constraint is satisfied:

$$\forall x \in I: [\![S]\!](x) = [\![S']\!](x), \tag{3}$$

where I is the input space; $[\![S]\!](x)$ represents the observable behavior (output trace, or side effects) of the system S on input x;

S' is the refactored system.

Thus, a refactoring is good if it induces semantically equivalent behavior on any legal input.

Graph-Based Refactoring Model. Let $G = (V, E)$ denote a graph representation of the program where:

V is the number of program entities, such as functions or variables;

$E \subseteq V \times V$ denotes control or data flow edges.

A refactoring can be formulated as a graph transformation operation:

$$T = (L, R, \phi), \tag{4}$$

where L is the pattern to match (left hand side); R is the replacement graph on the right; ϕ is a set of preconditions (e.g., type constraints, visibility, or usage bounds).

The refactoring is possible if there is a substructure $G_L \subseteq G$ such that:

$$G_L \vDash L, \qquad \phi(G_L) = true, \tag{5}$$

So, an updated graph is created:

$$G' = (G \setminus G_L) \cup R, \tag{6}$$

Semantic correctness implies:

$$[\![G]\!] = [\![G']\!]. \tag{7}$$

Invariants and safety conditions. In order to maintain behavioral safety, there is a set of invariants I_b that must always be respected by a code refactoring.

Input-output invariant:

$$\forall x \in D: f_S(x) = f_{S'}(x). \tag{8}$$

State space invariant (for systems with state):

$$\forall x \in D, s_0 \in \Sigma: \delta_S(x, s_0) = \delta_{S'}(x, s_0). \quad (9)$$

Pre-/post-condition preservation:

$$Pre_S(x) \Rightarrow Post_S(x) \Leftrightarrow Post_{S'}(x), \quad (10)$$

where δ is the transition function and Σ the set of internal program states.

Probabilistic guarantees with respect to test suite coverage. In the lack of correctness proofs, practitioners employ the test suite adequacy to approximate correctness.

Given a finite test set $T = \{x_1, x_2, \ldots, x_n\}$, we say a transformation is empirically correct if:

$$\forall x_i \in T: [\![S]\!](x_i) = [\![S']\!](x_i). \quad (11)$$

Another option is to learn confidence by approximating a probabilistic correctness bound:

$$P(correct) \geq 1 - \epsilon, \quad (12)$$

$$\epsilon = \frac{1}{|T|} \sum_{x \in I \setminus T} Pr[[\![S(x)]\!] \neq [\![S'(x)]\!]]. \quad (13)$$

This enables integration with CI/CD pipelines and works with reinforcement learning–based refactoring agents.

Semantic Distance Metrics. A different formalism to characterise the impact of a refactoring operation is semantic distance:

$$d_{sem}(S, S') = \int_I D([\![S]\!](x), [\![S']\!](x))dx, \quad (14)$$

where D is a divergence measure (e.g., Hamming distance, output hash mismatch).

A valid refactoring should satisfy:

$$d_{sem}(S, S') = 0. \quad (15)$$

As for approximate or lossy refactoring (eg. performance optimization), we can force an upper bound:

$$d_{sem}(S, S') \leq \delta. \quad (16)$$

4. RECOMMENDATIONS FOR PRACTICAL ADOPTION OF AUTOMATED REFACTORING TECHNIQUES

The scope of the surveyed methodologies is very broad in terms of complexity and automation degree, as their applicability in real projects is significantly determined by the matching between refactoring solutions and the constraints imposed by the development context. We make the following practical recommendations from the analysis.

1. Old-school, rule-based and IDE-integrated tools (such as IntelliJ IDEA, Eclipse, ReSharper), which are good for small-to-medium object-oriented systems with the testing coverage already in place continue to work well here. NameSteal/KP tools provide high explainability, instant developer control, and apply well-known transformations like Rename Method, Extract Method or Move Class.

2. For a big system in design erosion or with legacy code base, graph-based algorithms and the tool related to smell like design like detectors (Designite, RefactoringMiner) are preferable. They offer a structural view and may flag refactoring opportunities that span entire hierarchies or packages.

3. For new, data-rich projects that follow modern development stacks, AI-powered tools (e.g., DeepSmells, MoveRec, CodeBERT-based systems) can help in identifying complex smells and offering refactoring suggestions when conventional rules are insufficient. These systems can be especially handy when refactoring in uncertain circumstances or when taking on cross-cutting concerns.

4. In multilingual or dynamic typed (eg: python, javascript) settings, a good combination of dynamic analysis, runtime instrumentation and pretrained multilingual models (eg: PLBART, CodeT5+) is pursued. These are, however, cures for the symptom, they do not solve the problem of the missing static typing and support only limited generalisation of refactoring methods.

5. For projects with stringent reliability or regulatory requirements, other than only semantically validated transformations using formal verification tools or test-suite equivalence checks should be considered. In such cases, hybrid pipelines which mix AI-based advice and symbolic analysis (such as, SMT solvers, test case generation) are best suitable.

In the case of the continuous delivery environments, instruments that favor integration of CI/CD, non-blocking execution and refactoring-aware quality gates (OpenRewrite, RefactoringBot, SonarQube extensions and the like). These enable teams to make safe, incremental adjustments with minimal interference to deployment pipelines.

Hence, the choice of an appropriate refactoring should take into account:

- system size and complexity;
- language and typing model;
- availability of testing or specifications;
- organizational maturity in CI/CD;
- tolerance for automation vs human oversight.

By aligning tool features with project attributes, one to balance the automation efficiency, the improvement of the software quality and the operational safety of the online applications.

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

5. FUTURE DIRECTIONS

Although much effort has been devoted to automated refactoring, existing techniques suffer from several limitations, which undermine their effectiveness in large-scale real-world software systems.

These challenges include [37]:

- the poor quality of semantic understanding of code;
- lack of context awareness;
- the poor generalizability of machine learning models to different domains;
- the lack of trust, due to opaque decision making;
- the poor scalability in continuous integration environments.

Recently, researchers have proposed several promising research directions that may lead to the next generation of intelligent refactoring systems.

Welding of Deep Learning Models for Refactoring Prediction. One such promising line of work is applying deep learning methods like CNNs, RNNs, and transformers, to predict refactoring opportunities from raw code representations. Such models can be trained on large-scale datasets that are extracted from version control systems and thus capture patterns from common refactoring operations (e.g., Extract Method, Move Class). Furthermore, pre-trained code models such as CodeBERT and GraphCodeBERT have proved to capture the syntactic and semantic aspects of source code, which makes them suitable for learning deep refactoring strategies.

Semantic Analysis and Graph-based Code Representation. Another important line of research is the adoption of graph-based representations (e.g., ASTs, PDGs, heterogeneous code graphs). Such structures enable refactoring tool support for reasoning about semantic (or abstract) relationships and structural dependencies throughout the code. Using GNNs in combination with symbolic program analysis can greatly increase the precision of automated transformations and minimize the chance of affecting program behavior.

Knowledge-based Recommendation Systems. For enhancing the explainability and the confidence of developers, other works are pushing for creating knowledge-based refactoring systems. Such systems are based on encoded best practices, design patterns and domain ontologies, and suggest context-relevant transformations. Unlike pure data-driven systems, knowledge-based systems are capable of justifying recommendations based on traceable rules or conceptual models, which help in human-in-the-loop workflows and also suit regulatory needs in safety-critical domains.

Human-in-the-Loop and Developer Feedback Incorporation. Due to the complexity and context dependency of software refactoring, it is more and more clear super-organic level not always desirable or applicable. Hence, including developer input in the refactoring pipeline via interactive interfaces, ranking algorithms, or adaptive learning can aid in achieving a symbiosis of automation and human supervision. Human-in-the-loop approaches keep the system in line with project-specific style guidelines, coding conventions and shifting team preferences.

Scalable and CI/CD-Ready Architectures. A second important direction is to develop more scalable and modular refactoring architectures, which can be easily embedded into contemporary CI/CD pipelines. This involves the need for an efficient incremental analysis, distributed processing, and support for diverse programming languages and frameworks. These kinds of architectural solutions could even serve to underpin refactoring advice while committing/pulling code or testing, essentially integrating quality enhancements straight into the code lifecycle itself.

Explainable AI and Trust on Refactoring Decisions. Finally, the use of explainable AI (XAI) methods is crucial to narrow the gap between developers and AI-based refactoring tools. Justifying recommendations, pointing affected code regions and suggesting an alternative refactoring plan can increase the trust of developers working in the target software and accelerate the adoption of these tools in production. On the level of modeling and validating the different levels of uncertainty, future systems should include a way to estimate and validate the accuracy to avoid unpredictable ICHT behavior.

Improving explanation and semantic assurance of automated refactoring. Despite the improving precision of the AI-based refactoring tools, their actual use is still limited by two main problems: (i) explainability and (ii) semantic drift cost. Solving these challenges implies combining interpretable mechanisms which interpret the decisions of the model and offer formal guarantees on behavioral correctness.

One promising avenue is the usage of SHAP (SHapley Additive exPlanations), a model-agnostic framework for explaining feature attributions (e.g., cyclomatic complexity, method length, fan-in) on

the output of a given model. SHAP values make clear why in the recommendation engine particular structural properties are used for classifying a source code fragment as potential refactoring target.

Attention maps act as a built-in interpretability tool in transformer-based models (e.g., CodeBERT and CodeT5). By visualizing attention weights over tokens or AST nodes, one can track how the model gives attention to semantically cb meaning tokens in the code while making a decision. This level of self reflection is both helpful in solving model misbehavior and in making it easier for developers to make changes.

To avoid semantic drift – undesired behavioral changes introduced by the transformations – more recent work suggests the use of formal verification methods within the refactoring tooling. Symbolic execution engines or SMT solvers (e.g., z3) can be used to verify that post-refactoring code is functionally equivalent to its original. Instead, one may use automatic test generation frameworks for empirical evaluation of behavior-preserving transformations.

This convergence of explainability and verification techniques opens the door to the next generation of intelligent refactoring assistants – systems that do not just present high-quality transformations, but can justify them, and ensure them to be correct. We believe that hybrid architectures of this kind are necessary for the integration of refactoring tools into industrial, regulated, and safety-critical software development environments.

Integration of Automated Refactoring into CI/CD Pipelines. The value of CI/CDs integration support is often recognized in literature of automatic refactoring, however, little detail on how the integration can be done was documented. Yet now recent research, and also new commercial and open-source guidelines, show that integrating refactoring in the development process even further, into continuous workflows, results in more maintainable code that requires less work to create, and more time for developing new features.

In today’s era of software development, automatic refactoring is becoming part of the daily quality assurance in the continuous integration pipeline. The common way of running lightweight refactoring tools (like OpenRewrite, Designite, or ReSharper CLI) is to call them as standalone build steps or as part of the static code analysis stages. These tools analyze the code that has been committed and propose or deliver structural change, often based on pre-configured rules or learned patterns (e.g., method extraction, code deduplication or class decomposition).

A very powerful model is that of refactoring agents or bots that work with version control automate refactoring concurrently. These agents keep an eye on code changes and make pull request with proposed refactorings automatically. The changes tend to be rationalized with metric-based reasoning, and face the typical resistance validation (unit tests, linters, manual reviews). This asynchronous environment for interaction ensures minimal impact on developer workflow, while relentless increase in quality of code.

More sophisticated pipelines integrate refactoring-aware quality gates, which measure the structural quality of the code base and impose improvement thresholds. For example, SonarQube, can have alerts set or deployment blocked if technical debt/code smells exceed certain limits. The refactoring prescription coming from such AI-supported tools can then automatically fix the violations, fully closing the loop from detection to transformation.

Furthermore, the introduction of semantic checks (for example, by regression test suites or light-weight behavioral checks) leads to a high confidence that transformations do not change program behavior (i.e., they are preserving). For safety critical systems, this process of preservation of semantic invariants can potentially also be studied using formal verification modules or property-based test generators post-refactoring.

From a procedural context, three fundamental principles must be observed for such integrations to be successful:

- non-destructiveness – refactoring must not disrupt the deliverable or undo the developer's work;
- configurability – a project-specific rules, naming conventions and, architectural guidelines should be supported;
- traceable and explainable – automated transformations need to be visible, reviewable, and adhere to team coding conventions.

In this way, by effectively integrating automated refactoring into CI/CD pipelines, these tools cease to be an optional utility and instead become a proactive and context-aware part of the software lifecycle. In this way they enforce sustainable code quality management and by that aid in the long-term evolution of complex systems with low overhead.

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

CONCLUSIONS

This paper has presented an overview of the extant work on the topic of automatic refactorings in OO systems. Refactoring, an important activity aiming at enhancing internal code quality while preserving the external behavior of software, is one of the key practices in software evolution and maintenance. As contemporary applications become larger and more complex, it has become difficult for developers to apply such manual refactoring, opening the path to the development of automated tools.

We have surveyed a large range of refactoring approaches, such as tool-based, rule-based, graph-based, pattern-based, machine learning based, and history-aware. Each class employs a different approach that has its own advantages (e.g., speed, consistency, scalability) and weaknesses (lack of semantic understanding, difficulty of generalization, integration into industrial development pipelines), which provide the inspiration for techniques with complementary advantages and fewer weaknesses.

We also found certain benefits of automation like productivity, consistency of the mappings and ability to realize hidden defects in the legacy code. On the other hand, we stressed the main drawbacks, such as the risk of semantic drift, a low context awareness and an absent developer interpretability of the choices that can be made by AI.

A thorough exploration of open issues, including model limitation on generalization and lack of scalability, has been provided for paving the way of future research. These involve among others the fusion of deep learning for refactoring prediction, the exploit of graph-based code representation for semantic reasoning, and the creation of knowledge-based and explainable systems while considering human-in-the-loop model to balance between automation and control.

Although progress on automated refactoring has been made, there are tremendous opportunities for the improvement of its precision, flexibility, visibility, and usability. Through the exploration of hybrid and context aware approaches, the field can progress towards the realization of intelligent refactoring systems that can enable sustainable software engineering, both within academia and industry.

REFERENCES

1. Sharma, T., Efstathiou, V., Louridas, P., et al. “Code smell detection by deep direct-learning and transfer-learning”. *Journal of Systems and Software*. 2021; 176: 110936. DOI: https://doi.org/10.1016/j.jss.2021.110936.

2. Aniche, M., Maziero, E., Durelli, R., et al. “The effectiveness of supervised machine learning algorithms in predicting software refactoring”. *IEEE Transactions on Software Engineering*. 2020; 46 (9): 848–866. DOI: https://doi.org/10.1109/TSE.2020.3021736.

3. Desai, U., Bandyopadhyay, S. & Tamilselvam, S. “Graph neural network to dilute outliers for refactoring monolith application”. *Proceedings of AAAI*. 2021; 35 (3): 16079–16087. DOI: https://doi.org/10.48550/arXiv.2102.03827.

4. Mens, T. & Tourwe, T. “A Survey of Software Refactoring”. *IEEE Transactions on Software Engineering*. 2004; 30 (2): 126–139. DOI: https://doi.org/10.1109/TSE.2004.1265817.

5. Ratzinger, J., Sigmund, T., Vorburger, P., et al. “Mining Software Evolution to Predict Refactoring”. *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, Madrid, Spain, 2007. p. 354–363. DOI: https://doi.org/10.1109/ESEM.2007.9.

6. Sharma, T. & Mishra, P. “Designite: a software design quality assessment tool”. *IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities (BRIDGE)*. Austin, TX, USA. 2016. p. 1–4.

7. Fowler, M. “Refactoring: Improving the design of existing code”. *Addison-Wesley*. 1999. ISBN: 978-0-201-48567-7.

8. Liu, J., Huang, Y. & Zhang, L., et al. “CodeTrans: pre-trained LLM for automated restructuring suggestions”. *ESEC/FSE ’23*. 2023; 1: 123–134. DOI: https://doi.org/10.1145/3583645.3585432.

9. Martins, L., Bezerra, C., Costa, H. & Machado, I. “Smart prediction for refactorings in the software test code”. *Proceedings of the XXXV Brazilian Symposium on Software Engineering (SBES '21)*. 2021. p. 115–120. DOI: https://doi.org/10.1145/3474624.3477070.

10. Vimaladevi, M. & Zayaraz, G. “Stability Aware Software Refactoring Using Hybrid Search Based Techniques”. *2017 International Conference on Technical Advancements in Computers and Communications (ICTACC)*. 2017. p. 32–35. DOI: https://doi.org/10.1109/ICTACC.2017.18.

11. Ivers, J., Nord, R. L., Ozkaya, I., Seifried, C., Timperley, C. S. & Kessentini, M. “Industry's Cry for Tools that Support Large-Scale Refactoring”. *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2022. p. 163–164. DOI: https://doi.org/10.1145/3510457.3513074.

12. Arcelli, D., Cortellessa, V. & Di Pompeo, D. “Performance-Driven Software Architecture Refactoring”. *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2018. p. 2–3. DOI: https://doi.org/10.1109/ICSA-C.2018.00006.

13. Techapalokul, P. & Tilevich, E. “Programming environments for blocks need first-class software refactoring support: A position paper”. *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 2015. p. 109–111. DOI: https://doi.org/10.1109/BLOCKS.2015.7369015.

14. Li, Z., Lei, H., Ma, Z. & Zhang, F. “Code similarity prediction using GNNs.” *Entropy*. 2024; 26 (6): 505. DOI: https://doi.org/10.3390/e26060505.

15. Shrivastava, S. V. & Shrivastava, V. “Impact of metrics based refactoring on the software quality: a case study”. *TENCON 2008–2008 IEEE Region 10 Conference*. 2008. p. 1–6. DOI: https://doi.org/10.1109/TENCON.2008.4766459.

16. LeClair, A., Haque, S., Wu, L. & McMillan, C. “Improved Code Summarization via a GNN.” *arXiv preprint*. 2020. DOI: https://doi.org/10.48550/arXiv.2004.02843.

17. Baumgartner, N., Iyenghar, P., Schoemaker, T. & Pulvermüller, E. “AI-Driven refactoring pipeline for data clumps”. *Electronics*. 2024; 13 (9): 1644. DOI: https://doi.org/10.3390/electronics13091644.

18. Nyirongo, B., Jiang, Y., Jiang, H. & Liu, H. “A survey of deep learning based software refactoring”. arXiv. 2024. DOI: https://doi.org/10.48550/arXiv.2404.19226.

19. Konakanchi, S. “Artificial intelligence in code optimization and refactoring”. *Int. J. Sci. Res. Comput. Sci. Eng. Inf. Tech.* 2025; 11 (2): 1197–1211. DOI: https://doi.org/10.32628/CSEIT25112463.

20. Polu, O. R. “AI-driven automatic code refactoring for performance optimization” . *Int. J. Spec. Res. Comput. Sci. Eng. Inf. Tech*. 2025; 14 (1): 1316–1320. DOI: https://doi.org/10.21275/SR25011114610.

21. Tahsin, N. & Sakib, K. “Refactoring Community Smells: An Empirical Study on the Software Practitioners of Bangladesh”. *29th Asia-Pacific Software Engineering Conference (APSEC)*. 2022. p. 422–426. DOI: https://doi.org/10.1109/APSEC57359.2022.00055.

22. Alizadeh, V., Ouali, M. A., Kessentini, M. & Chater, M. “RefBot: Intelligent Software Refactoring Bot”. *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019. p. 823–834. DOI: https://doi.org/10.1109/ASE.2019.00081.

23. Alizadeh, V., Kessentini, M., Mkaouer, M. W., Ó Cinnéide, M., Ouni, A. & Cai, Y. “An Interactive and Dynamic Search-Based Approach to Software Refactoring Recommendations”. *IEEE Transactions on Software Engineering*. 2020; 46 (9): 932–961. DOI: https://doi.org/10.1109/TSE.2018.2872711.

24. Almogahed, A., Mahdin, H., Rejab, M. M., et al. “Code Refactoring for Software Reusability: An Experimental Study”. *2024 4th International Conference on Emerging Smart Technologies and Applications (eSmarTA)*. 2024. p. 1–6. DOI: https://doi.org/10.1109/eSmarTA62850.2024.10638872.

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

26. Coelho, F., Massoni, T. & Alves, E. L. G. “Refactoring-Aware Code Review: A Systematic Mapping Study”. *IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*. 2019. p. 63–66. DOI: https://doi.org/10.1109/IWoR.2019.00019.

27. Arcelli, D., Cortellessa, V. & Di Pompeo, D. “Automating Performance Antipattern Detection and Software Refactoring in UML Models”. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019. p. 639–643. DOI: https://doi.org/10.1109/SANER.2019.8667967.

28. Rahman, M. M. & Satter, A. “A Context Based Approach for Recommending Move Class Refactoring”. *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. 2020. p. 515–516. DOI: https://doi.org/10.1109/APSEC51365.2020.00070.

29. Pandimurugan, P., Parvathi, M. & Jenila, A. “A survey of software testing in refactoring based software models”. *International Conference on Nanoscience, Engineering and Technology (ICONSET 2011)*. 2011. p. 571–573. DOI: https://doi.org/10.1109/ICONSET.2011.6168034.

30. Almogahed, A., Mahdin, H., Omar, M., Zakaria, N. H., Muhammad, G. & Ali, Z. “Optimized Refactoring Mechanisms to Improve Quality Characteristics in Object-Oriented Systems”. *IEEE Access*. 2023; 11: 99143–99158. DOI: https://doi.org/10.1109/ACCESS.2023.3313186.

31. Motogna, S., Berciu, L.-M. & Moldovan, V.-A. “Artificial Intelligence Methods in Software Refactoring: A Systematic Literature Review”. *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2024. p. 309–316. DOI: https://doi.org/10.1109/SEAA64295.2024.00055.

32. Zhao, Y., Yang, Y., Zhou, Y. & Ding, Z. “DEPICTER: A Design-Principle Guided and Heuristic-Rule Constrained Software Refactoring Approach”. *IEEE Transactions on Reliability*. 2022; 71 (2): 698–715. DOI: https://doi.org/10.1109/TR.2022.3159851.

33. Almogahed, A., Mahdin, H., Omar, M., Zakaria, N. H., Mostafa, S. A., AlQahtani, S. A., Pathak, P., Shaharudin, S. M. & Hidayat, R. “A Refactoring Classification Framework for Efficient Software Maintenance”. *IEEE Access*. 2023; 11: 78904–78917. DOI: https://doi.org/10.1109/ACCESS.2023.3298678.

34. Sabir, S. & Rasool, G. “A Lightweight Approach for Detection of Software Design Smells”. *6th International Conference on Advancements in Computational Sciences (ICACS)*. 2025. p. 1–4. DOI: https://doi.org/10.1109/ICACS64902.2025.10937837.

35. Singh, N. & Singh, P. “How Do Code Refactoring Activities Impact Software Developers' Sentiments? – An Empirical Investigation Into GitHub Commits”. *24th Asia-Pacific Software Engineering Conference (APSEC)*. 2017. p. 648–653. DOI: https://doi.org/10.1109/APSEC.2017.79.

36. Meananeatra, P., Rongviriyapanish, S. & Apiwattanapong, T. “Using software metrics to select refactoring for long method bad smell”. *The 8th Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI) Association of Thailand – Conference*. 2011. p. 492–495. DOI: https://doi.org/10.1109/ECTICON.2011.5947882.

37. Usha, K., Poonguzhali, N. & Kavitha, E. “A Quantitative Approach for Evaluating the Effectiveness of Refactoring in Software Development Process”. *Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS)*. 2009. p. 1–7. DOI: https://doi.org/10.1109/ICM2CS.2009.5397935.

Conflicts of Interest: The authors declare that they have no conflict of interest regarding this study, including financial, personal, authorship or other, which could influence the research and its results presented in this article
Author Mykola A. Hodovychenko is a member of the Editorial Board of this journal. This role had no influence on the peer review process or editorial decision regarding this manuscript

Received 27.03.2025
Received after revision 12.06.2025
Accepted 18.06.2025

DOI: https://doi.org/10.15276/hait.08.2025.11
УДК 004.4'2:004.451.44

Аналіз існуючих підходів до автоматизації рефакторингу об’єктно-орієнтованих програмних систем

Годовиченко Микола Анатолійович[1)]
ORCID: https://orcid.org/0000-0001-5422-3048, hodovychneko@op.edu.ua. Scopus Author ID: 57188700773
Курінько Дмитро Дмитрович[1)]
ORCID: https://orcid.org/0000-0001-8304-3257, dmitrykurinko@gmail.com
[1)] Національний університет «Одеська Політехніка», пр. Шевченка, 1. Одеса, Україна, 65044

АНОТАЦІЯ

Автоматизований рефакторинг відіграє ключову роль у супроводі й еволюції об'єктно-орієнтованих програмних систем, де покращення внутрішньої структури коду є визначальним чинником підтримуваності, масштабованості та зменшення технічного боргу. У цій роботі представлено розширений огляд сучасних підходів до автоматизованого рефакторингу з акцентом на методологічні засади, рівень автоматизації, застосування штучного інтелекту та практичну інтеграцію в CI/CD-процеси. Розглянуто традиційні rule-based та graph-based методи, моделі глибокого навчання (CNN, GNN, LLM), історично обґрунтовані техніки (MSR), а також гібридні підходи із залученням людського контролю (human-in-the-loop). Особливу увагу приділено класифікації трансформацій згідно з авторитетною термінологією (Fowler): структурним, архітектурним (semantic) та поведінковим перетворенням з фокусом на збереження інваріантів поведінки. Окремо розглянуто формалізовані моделі рефакторингу як графові перетворення з чітко визначеними передумовами та постумовами, що дозволяє гарантувати семантичну еквівалентність між версіями програми. У роботі представлено приклад реального застосування інструмента DeepSmells, що демонструє змістовну трансформацію «до/після» та коментує обґрунтованість запропонованих змін. Досліджено виклики explainability та semantic drift, а також запропоновано способи їх усунення через SHAP-аналіз, attention-візуалізацію в трансформерах, інтеграцію з формальними верифікаторами (SMT, symbolic execution) та пояснювані AI-рекомендації. Окремий акцент зроблено на обмеженнях у динамічно типізованих мовах (Python, JavaScript), де типова статична перевірка втрачає ефективність. Узагальнення на мультимовні проєкти підтримується завдяки застосуванню моделей CodeBERT, CodeT5, PLBART, які працюють із графами, токенами та кросмовними узагальненнями. Показано практичну інтеграцію автоматизованого рефакторингу у CI/CD-середовище – через боти, refactoring-aware quality gates, періодичне застосування трансформацій у pre-commit/merge-циклах, а також перевірку інваріантів за допомогою тестів або формальних засобів. Стаття орієнтована на інженерів, дослідників і розробників інструментів, які працюють у галузі інтелектуального супроводу ПЗ та автоматизації процесів підтримки якості коду. Представлені класифікації, практичні сценарії та критерії вибору інструментів забезпечують прикладну цінність огляду для розробки власних рішень або впровадження існуючих технологій у проєкти з різними вимогами до стабільності, масштабування та рівня автоматизації.

Ключові слова: автоматизований рефакторинг; об'єктно-орієнтоване програмування; глибоке навчання; супровід програмного забезпечення; запахи коду; графові нейронні мережі; інженерія програмного забезпечення; трансформація коду; аналіз вихідного коду; семантичне моделювання коду; метрики якості програмного забезпечення

ABOUT THE AUTHORS

Mykola A. Hodovychenko - Candidate of Engineering Sciences, Associate Professor, Artificial Intelligence and Data Analysis Department. Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine
ORCID: 0000-0001-5422-3048; hodovychenko@op.edu.ua. Scopus Author ID: 57188700773
***Research field*:** machine learning and artificial intelligence, software engineering, intelligent software maintenance, video processing, motion tracking, project-based learning, pattern recognition

Годовиченко Микола Анатолійович - кандидат технічних наук, доцент кафедри Штучного інтелекту та аналізу даних, Національний університет «Одеська Політехніка», пр. Шевченка, 1. Одеса, 65044, Україна

Dmytro D. Kurinko - PhD Student, Artificial Intelligence and Data Analysis Department. Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine
ORCID: 0000-0001-8304-3257a, dmitrykurinko@gmail.com
***Research field*:** Machine learning and artificial intelligence, software engineering, pattern recognition, data-driven software architecture analysis, computer vision, knowledge representation in software systems

Курінько Дмитро Дмитрович - аспірант кафедри Штучного інтелекту та аналізу даних, Національний університет «Одеська Політехніка», пр. Шевченка, 1. Одеса, 65044, Україна