# iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring

Di Wu[1†], Fangwen Mu[2†], Lin Shi[1§], Zhaoqiang Guo[3], Kui Liu[3], Weiguang Zhuang[1], Yuqi Zhong[1], Li Zhang[1]
[1]Beihang University, Beijing, China;
[2]Institute of Software Chinese Academy of Sciences, Beijing, China;
[3]Software Engineering Application Technology Lab, Huawei, China
{shilin,EdisonWu,20373227,22373230}@buaa.edu.cn,
fangwen2020@iscas.ac.cn,gzq@smail.nju.edu.cn,brucekuiliu@gmail.com

## ABSTRACT

Detecting and refactoring code smells is challenging, laborious, and sustaining. Although large language models have demonstrated potential in identifying various types of code smells, they also have limitations such as input-output token restrictions, difficulty in accessing repository-level knowledge, and performing dynamic source code analysis. Existing learning-based methods or commercial expert toolsets have advantages in handling complex smells. They can analyze project structures and contextual information in-depth, access global code repositories, and utilize advanced code analysis techniques. However, these toolsets are often designed for specific types and patterns of code smells and can only address fixed smells, lacking flexibility and scalability. To resolve that problem, we propose iSMELL, an ensemble approach that employs various code smell detection toolsets via Mixture of Experts (MoE) architecture for comprehensive code smell detection, and enhances the LLMs with the detection results from expert toolsets for refactoring those identified code smells. First, we train a MoE model that, based on input code vectors, outputs the most suitable expert tool for identifying each type of smell. Then, we select the recommended toolsets for code smell detection and obtain their results. Finally, we equip the prompts with the detection results from the expert toolsets, thereby enhancing the refactoring capability of LLMs for code with existing smells, enabling them to provide different solutions based on the type of smell. We evaluate our approach on detecting and refactoring three classical and complex code smells, i.e., Refused Bequest, God Class, and Feature Envy. The results show that, by adopting seven expert code smell toolsets, iSMELL achieved an average F1 score of 75.17% on code smell detection, outperforming LLMs baselines by an increase of 35.05% in F1 score. We further evaluate the code refactored by the enhanced LLM. The quantitative and human evaluation results show that iSMELL could improve code quality metrics and conduct satisfactory refactoring toward the identified code smells. We believe that our proposed solution could provide new insights into better leveraging LLMs and existing approaches to resolving complex software tasks.

**ACM Reference Format:**
Di Wu[1†], Fangwen Mu[2†], Lin Shi[1§], Zhaoqiang Guo[3], Kui Liu[3], Weiguang Zhuang[1], Yuqi Zhong[1], Li Zhang[1]. 2024. iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695508

## 1 INTRODUCTION

The concept of "code smell" was first introduced by Fowler *et al.* in 1999 [1], used to denote the segments of software code that are of low quality and in urgent need of refactoring. Throughout the development and evolution of software, code smells often sneak in due to factors such as pressing delivery deadlines and developer negligence, thereby undermining the readability and maintainability of the code. However, manually identifying code smells is not easy, partly due to the subjectivity of the definition of smells and the high dependency of refactoring decisions on individual intuition and experience [2].

Recently, Large Language Models (LLMs) [3–5] have been increasingly recognized for their potential as powerful aids in various software engineering tasks [6–9]. LLMs, trained on extensive volumes of source code, have the ability to deeply comprehend code and discern a myriad of underlying issues within it. In addition, LLMs have demonstrated the ability to effectively detect and analyze various types of code smells. They also face several limitations, including constraints on input-output token counts, difficulty in directly leveraging repository-level contextual knowledge, and limitations in performing dynamic source code analysis tasks [10, 11].

Existing learning-based [12, 13] or rule-based [14–16] expert toolsets for code smell detection have advantages in handling complex smells. They can analyze project structures and contextual information in-depth, access global code repositories, and utilize advanced analysis techniques. Unfortunately, these methods have several limitations, particularly low consistency among detection toolsets [17]. This leads to divergent outcomes when distinct expert toolsets inspect identical code segments. Moreover, these toolsets are often designed for specific types and patterns of code smells and can only address fixed smells, lacking flexibility and scalability.

† Both authors contributed equally to this work.
§ Corresponding author.


*ASE '24, October 27-November 1, 2024, Sacramento, CA, USA*

ACM ISBN 979-8-4007-1248-7/24/10…$15.00
https://doi.org/10.1145/3691620.3695508

To address this challenge, we propose an ensemble approach iSMELL. It leverages a Mixture of Experts (MoE) [18] architecture to comprehensively detect code smells using expert toolsets for code smell detection, and enhances LLMs with the detection results from expert toolsets for refactoring identified code smells. First, we train a MoE model where the input vector consists of concatenated embeddings from code metrics and code representations. Based on this input, the model outputs the most suitable detection tool for each type of code smell. Then, we select the recommended expert toolsets for code smell detection and obtain their results. Finally, we construct the prompts using the detection results from the expert toolsets, thereby enhancing the LLMs' ability to refactor code with existing smells, enabling them to provide different solutions based on the type of smell. Evaluation results demonstrate that iSMELL performs effectively in detecting three popular and complex code smells (Refused Bequest, God Class, Feature Envy). The iSMELL approach achieves a significant improvement in F1 score compared to state-of-the-art large language models, with an average increase of 35.05%. Compared to the most advanced expert toolsets, the average improvement is 9.74%. We adopt a comprehensive evaluation method, including both quantitative analysis and human evaluation. The experimental findings demonstrate the efficiency and effectiveness of iSMELL in identifying and refactoring scenarios containing the aforementioned code smells, highlighting its contribution to improving code quality. iSMELL offers scalability, comprehensiveness, and flexibility by integrating diverse detection tools, with the potential for further performance enhancements through advancements in specific smell detection tools. We believe that the proposed solution can offer new insights for better utilizing LLMs and existing methods to tackle complex software tasks. Our main contributions are:

- **Technique:** Designed and implemented the first ensemble approach iSMELL that combines multi-smell detection with large-scale model refactoring, leveraging a MoE model to integrate multiple smell detection toolsets, demonstrating strong scalability.
- **Evaluation:** Results indicate that iSMELL outperforms the best-performing LLMs and expert toolsets in code smell detection. Furthermore, experiments and manual evaluations confirm the outstanding performance of iSMELL in the field of code refactoring, providing powerful tool support for software engineering practices.
- **Data:** We provide publicly accessible datasets and source code [19] to enable others to replicate our research and apply it in broader contexts.

## 2 BACKGROUND

Current code smell detection and refactoring practices mainly suffer from two challenges: the diversity and deviation of a large number of expert tools and the difficulty in making good use of LLMs.

### 2.1 The Diversity and Deviation of Expert tools

There are currently numerous code smell detection toolsets available, with Lacerda's [20] review reporting as many as 162 different toolsets, each focusing on detecting various types of smells. For instance, CCFinder [21] specializes in identifying code clones, while PMD [22] covers detection of various smells such as Long Method and God Class. Due to the specialized nature of each tool, relying solely on one tool makes it difficult to achieve comprehensive coverage of code smell types, leading to blind spots and potential omissions of certain smells. Furthermore, the lack of standardization in smell identification criteria among these toolsets is particularly prominent when dealing with more complex code smells. For example, Feature Envy, defined as a function excessively relying on external class data and methods rather than focusing on its own class members, poses challenges in standardized detection. In our research, we observed discrepancies in detection results among several mainstream toolsets even for the same code segment [23] labeled with "feature envy" in the dataset. Moreover, while a tool may accurately identify a smell in one case, it may erroneously report the presence of a smell in another segment of code [24] where none exists, indicating the instability of its results.

### 2.2 Limitations of LLMs in Smell Detection

LLMs, trained on massive code repositories containing billions of methods authored by real developers, are believed to have the potential to understand code deeply and thus effectively identify various underlying issues [25–27]. However, when facing complex smells, the performance of LLMs in these smells often diminishes. God Class refers to a class that does too much, encompassing numerous properties and methods. For God Class, the class code involved often spans thousands of lines, and the input token constraints of LLMs limit the performance of detecting this type of smell. For example, in the case of GPT-3.5, the input context token limit is 4k, which means we cannot input the entire class code to the LLMs. On the other hand, the Feature Envy smell involves methods highly interacting with multiple classes, requiring the model to comprehensively consider a wide range of method call chains and associated class information. Incorporating complete information about all related classes into the model's input range poses significant challenges. Refused Bequest occurs when a subclass is unwilling or unable to fully adhere to the behavior of its parent class, rendering the inheritance relationship awkward and superfluous. Similarly, the Refused Bequest smell also exhibits cross-class characteristics, requiring the model to possess the ability to understand and utilize repository-level contextual information, further testing the detection capabilities of LLMs. Furthermore, conventional code smell detection toolsets, by compiling and executing source code, are capable of extracting more exhaustive program structures and global project information. In contrast, LLMs in their present applications are unable to directly perform dynamic source code analysis, thereby encountering limitations in handling real-time interactive data. Therefore, while LLMs demonstrate the ability to effectively detect and analyze various code smells, they also face several limitations, including constraints on input-output token counts, difficulty in directly leveraging repository-level contextual knowledge, and limitations in performing dynamic source code analysis tasks.

Based on the above two observations, we argue that there is a need for a comprehensive multi-smell detection system that integrates the detection results of various expert toolsets and provides unified and reliable refactoring guidance through intelligent analysis to improve the accuracy and efficiency of refactoring decisions.

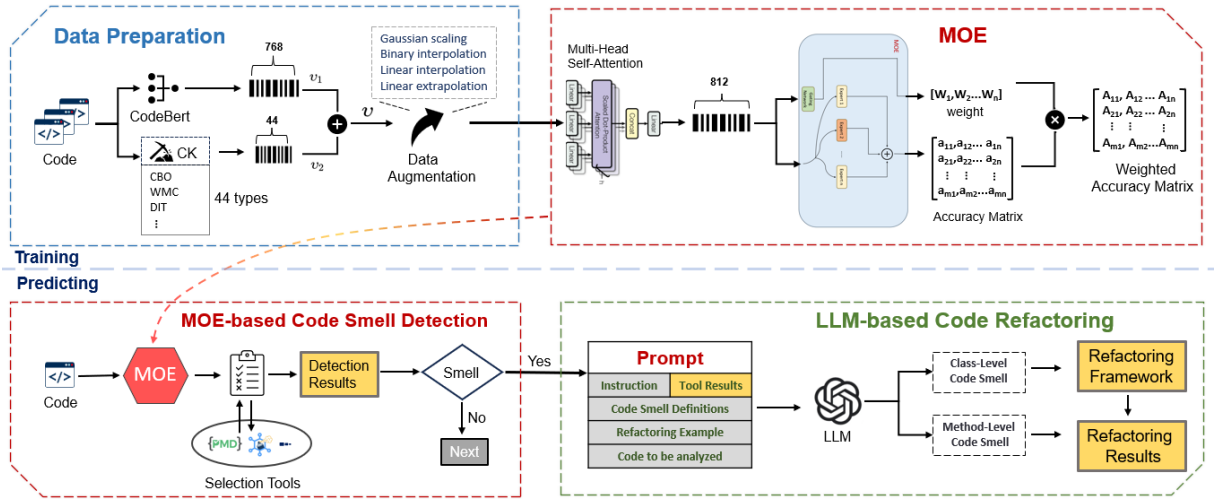


**Figure 1: The overall architecture of iSMELL**

## 3 APPROACH

In this section, we introduce our iSMELL, a comprehensive approach that integrates multi-smell detection with LLMs refactoring. Figure 1 provides an overview of iSMELL, which consists of four main stages: (1) Data Preparation, where we first obtain detection results from the expert toolsets for the dataset, then concatenate the metrics extracted from the input code with CodeBERT [3] embeddings. We employ representation-level data augmentation aimed at expanding the training dataset size to optimize iSMELL's predictive performance; (2) MoE-based Code Smell Detection, where we train the MoE model to select expert toolsets. We use a multi-head self-attention mechanism [28] to enhance the vectors input to the MoE model. The model outputs an accuracy matrix to select expert toolsets for detecting code; (3) LLM-based Code Refactoring, if the smell is detected, we adjust the prompt according to the type of smell, enabling it to provide targeted solutions. Then, we delve into the details of each stage in iSMELL.

### 3.1 Data Preparation

*3.1.1 Combined Code Embedding.* For comprehensive analysis and handling of code, it is crucial to meticulously capture the multidimensional characteristics of code. The vectors fed into our model are concatenated from metric vectors and semantic feature vectors, with the specific extraction process detailed as follows.

**BERT-based Code Embedding:** iSMELL harnesses CodeBERT to precisely encode deep semantic information of source code snippets. CodeBERT is designed based on the Transformer architecture, renowned for its powerful sequence handling capabilities and self-attention mechanisms, excelling in natural language processing (NLP) and understanding programming languages [29–31]. Following pre-training on large-scale multimodal data, CodeBERT can comprehend the complex structure, syntactic features, and underlying programming intents and logical connections within code. Upon inputting code, CodeBERT's encoder meticulously analyzes it, yielding a 768-dimensional vector $v_1$. This vector embodies a high-dimensional semantic embedding of the original code, encapsulating its multifaceted features.

**Metric-based Code Embedding:** To obtain various quantitative characteristics of the code, we use the CK Metric Extractor [32], which extracts quantitative information at class and method levels directly from uncompiled Java source code through static analysis, bypassing the compilation step. This tool encompasses 35 key metrics, such as WMC (Weighted Method Class), DIT (Depth of Inheritance Tree), NOC (Number of Children), CBO (Coupling between Objects), RFC (Response for a Class) and LCOM (Lack of Cohesion of Methods) etc. It covers the widely recognized CK metric suite [33, 34], providing a solid foundation for quality assessment and complexity analysis in the field of software engineering. For each metric, iSMELL calculates nine statistics including sums, medians, standard deviations, variances, minimums, maximums, skewness, kurtosis, and entropy. These additional statistical insights, combined with the original 35-dimensional metric vectors, constitute a higher-dimensional 44-dimensional feature vector $v_2$, comprehensively characterizing various aspects of the code. Finally, iSMELL concatenates $v_1$ with $v_2$, forming an integrated 812-dimensional vector $v$.

*3.1.2 Data Augmentation.* In our research, each data sample undergoes comprehensive code smell detection using seven specialized toolsets. These toolsets necessitate projects to be in a compilable state to ensure accurate detection. Given that our dataset originates from a diverse array of GitHub open-source projects, the task of ensuring successful configuration and compilation for each project is exceedingly time-consuming, leading to the collection of merely a few hundred valid records. The scarcity of training samples is prone to causing overfitting. Consequently, we employed data augmentation techniques to mitigate this issue. Many data augmentation techniques leverage predefined rules to transform code while preserving syntactic correctness and semantic meaning. Nonetheless, these approaches are resource-intensive, necessitating that models re-embed the data for each augmented instance. To tackle this issue, we followed previous work [35] and meticulously select four efficient representation-level data augmentation techniques to expand our training set. The augmented data is fifteen times that of the original. These techniques are Gaussian Scaling

[35], Binary Interpolation [35], Linear Interpolation [36], and Linear Extrapolation [35]. Gaussian Scaling operates by multiplying each value in a vector by a factor randomly drawn from a Gaussian distribution:

$$h_{\text{new}} = h \cdot (1 + \epsilon) \tag{1}$$

$h$ is the original feature vector (inputs). $\epsilon$ is random noise sampled from a normal distribution $N\left(0, \sigma^2\right)$ with a mean of 0 and standard deviation of $\sigma$. $h_{\text{new}}$ is the feature vector after Gaussian Scaling. Binary Interpolation is a distinctive interpolation technique where it randomly swaps some features between two selected samples:

$$h_{\text{new}}\,[i] = m[i] \cdot h_1[i] + (1 - m[i]) \cdot h_2[i] \tag{2}$$

$m[i]$ denotes a random binary mask obtained from a Bernoulli distribution. Meanwhile, $h_1[i]$ and $h_2[i]$ signify the initial data points for the features being considered. Linear Interpolation and Linear Extrapolation are very similar. Linear Interpolation assumes that the relationship between data points is linear. Linear Extrapolation predicts new data points outside the range of known data points:

$$h_{\text{new}} = \lambda \cdot h_i + (1 - \lambda) \cdot h_j \tag{3}$$

The $h_i$and $h_j$ are the original data points. The interpolation factor, $\lambda$, determines the position of the new data point relative to the two original points, effectively interpolating between them when $\lambda$ falls within the range of 0 to 1 for Linear Interpolation. Conversely, in Linear Extrapolation, $\lambda$ takes values less than 0 or greater than 1, extending beyond the known data boundaries. After transforming each sample in the dataset into an 812-dimensional vector $v$, we proceed to apply representation-level augmentation by selecting two samples with identical labels and code smell types.

## 3.2 MoE-based Code Smell Detection

To ensure the acquisition of optimal information from the most suitable detection experts, we introduce a gating network to weight each expert tool's outputs. This gating network, in collaboration with the multi-headed self-attention mechanisms embedded within each expert model, jointly captures the profound and multidimensional features of source code snippets. This synergistic process culminates in the generation of a weighted accuracy matrix.

*3.2.1 Gating Network.* To implement this mechanism, the gating network plays a crucial role. It first deeply analyzes the 812-dimensional composite vector obtained from CodeBERT and CK metric toolsets, identifying key features of code snippets through complex nonlinear transformations. Subsequently, based on these features, the gating network dynamically assigns weights to each expert tool, directly guiding the prediction of different smells in subsequent stages. It is noteworthy that since the output of each expert tool is tailored to a specific smell, the gating network ensures that for each smell, the model can extract information from the most suitable expert rather than simply averaging the opinions of all experts. The processing logic of the gating function begins by first retaining the top $k$ maximum values from the input vector $H(x)$ using the TopK function. Subsequently, the Softmax function is applied to normalize these values, resulting in the gating vector. This gating vector is utilized to weight-adjust the outputs of various expert models. The formula for the gating function is as follows:

$$G_\sigma(x) = \text{Softmax}(\text{TopK}(H(x), k)) \tag{4}$$

$G_\sigma(x)$ is the gating vector employed to weigh the outputs from expert models, generated from the input vector $H(x)$ following processing via TopK and Softmax operations. The formula for the TopK function, which retains the top $k$ maximum values:

$$\text{Top}K(v, k)_i = f(x) \tag{5}$$

$$f(x) = \begin{cases} v_i, & \text{if } v_i \text{ is in the top } k \text{ elements of } v \\ -\infty, & \text{otherwise} \end{cases} \tag{6}$$

$\text{Top}K(v, k)_i$: The output of the function, representing the $i$th element of the resulting vector retaining the top $k$ maximum values from $v$. In Equation 4, $H(x)$ constitutes the feature vector emitted by expert models, where each element $H(x)_i$ represents the output of an expert model for a specific feature.

$$\begin{aligned} H(x)_i = (x \cdot W_g)_i + \text{StandardNormal}() \\ \cdot \text{Softplus}((x \cdot W_{\text{noise}})_i) \end{aligned} \tag{7}$$

$W_g$ is the weight matrix transforming the input vector $x$ into the feature vector produced by expert models. StandardNormal() is a value randomly sampled from a standard normal distribution. Softplus(), representing the Softplus function, is introduced to incorporate non-linear characteristics into the model's processing. $W_{\text{noise}}$ constitutes the noise weight matrix, which introduces noise to the input vector $x$ with the aim of bolstering the model's robustness against variations.

*3.2.2 Multi-Head Self-Attention.* The iSMELL approach employs a Multi-Head Self-Attention mechanism to further refine and explore the deep semantics of input vectors. The core of the Multi-Head Self-Attention mechanism lies in its ability to parallelly process multiple attention heads, each focusing on a different subspace of the input vector, thereby capturing information from diverse perspectives. First, we use a set of linear transformations to map the input vector into sets of query, key, and value vectors, each set corresponding to a specific attention head. Then, by calculating the dot product between queries and keys and applying scaling and softmax functions, we generate a set of attention weights for each head, reflecting the relationships between different parts of the input vector and their importance to the current context. Next, based on these weights, we perform a weighted sum of the value vectors to generate the output for each head. The outputs of all heads are concatenated and passed through a linear network transformation, restoring them to the original dimension, but by this point, the vector has undergone deep feature interaction and information recombination, rich in cross-dimensional contextual understanding.

*3.2.3 Accuracy Matrix.* In the MoE model within iSMELL, both the gating network and each individual expert model are endowed with independent self-attention mechanisms. This comprehensive feature vector is simultaneously fed into seven distinct expert models. Each expert model independently outputs vector $v \in \mathbb{R}^{1\times3}$ for the three core smell classes. Each dimension of the vector corresponds to a specific code smell, with its numerical value directly indicating the "confidence" or accuracy level of the expert tool in detecting

that particular smell. Thus, the collection of vectors output by the seven expert models naturally forms a matrix $M \in \mathbb{R}^{7\times3}$, where each row represents an expert, and each column is associated with the detection of a specific smell type. The key step is to perform a dot product operation between the weight vector generated by the gating network and the aforementioned accuracy matrix to obtain the weighted accuracy matrix $M_{wei} \in \mathbb{R}^{7\times3}$. This operation essentially weights the accuracy of the expert model outputs, ensuring that those expert models with higher recognition ability for specific smells have a greater impact. The resulting weighted accuracy matrix contains column vectors that compare the weighted confidence of all expert toolsets under each smell type.

In the final decision stage, for each smell, we identify the expert tool with the highest weighted accuracy from the corresponding column vectors, and then use the selected expert tool to check for the corresponding code smells.

## 3.3 LLM-based Code Refactoring

In the iSMELL approach, once the Mixture of Experts (MoE) model selects expert toolsets to identify specific code smells, the system automatically triggers a detailed LLM refactoring process. This process embodies differential strategies for handling different smell types, aiming to ensure that the refactoring suggestions are both precise and efficient.

*3.3.1 Refactoring Prompt Design.* The prompt for code smell refactoring primarily consists of five components: instructions, tool results, definitions, refactoring examples, and the code to be analyzed. **Instructions:** Initially, we request the LLMs to process specific types of code smells based on the results of previous code smell detection toolsets. For each type of code smell, we also offer specific suggestions for refactoring to enhance the LLMs' ability to address that particular code smell. **Definitions:** To make LLMs understand the code smell better, we provide definitions for the code smells detected by the toolset. **Refactoring Examples:** First, we build a retrieval database *EDB* for refactoring examples. For each type of code smell, we carefully select five refactoring examples from open-source projects. Then, when preparing the prompt for a given code snippet and its specific code smell, we randomly retrieve one from the database *EDB*. **Code to be analyzed:** This component presents the code snippet that need to be analyzed. To make the LLMs learn more about the repo-level context, we ship the contextual information with the code to be analyzed. For "Refused Bequest", we input the code of the parent class into the LLMs. This step improves the model's understanding and management of

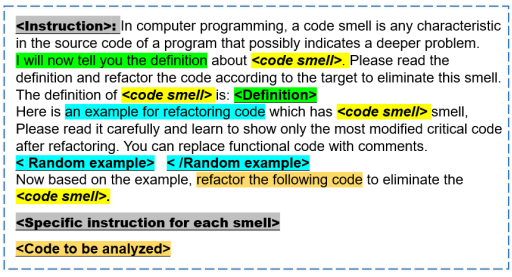


**Figure 2: Prompt for code refactoring**

inheritance structures, thereby enhancing the accuracy and effectiveness of code smell refactoring. The core of the God Class issue lies in the irrational internal structure of the class, where direct analysis of the class's code alone suffices for initiating refactoring. For Feature Envy, the primary action involves extracting parts of external classes that are frequently accessed within the method. In refactoring these smells, no additional information beyond the code under consideration needs to be provided to the LLMs. We provide the concrete prompt for each code smell on our website [19].

*3.3.2 Performing the Refactoring.* There are two main types of code refactoring for fixing code smells according to the impact scope, *i.e.*, refactoring for class-level and method-level code smell [37]. **Class-Level Refactoring** has an impact on multiple classes. Thus, both the code snippets to be analyzed and the output refactoring results are typically quite lengthy. When requiring LLMs to directly generate the complete refactored code, it typically entails multiple iterative queries to compile the entire codebase. This process can be fraught with issues such as selective forgetting and high interaction costs. Moreover, the inference process may even halt due to execution policy. Consequently, in the context of class-level code smells, we do not mandate that LLMs provide explicit modification code; instead, we employ an abstract yet efficient approach by outputting a guideline for refactoring. Specifically, which consists of four main stages. ❶ **MoE-based Smelly Code Detection**: We leverage MoE model to analyze the original code and detect smelly code. ❷ **LLM-based Structure Design**: We utilize LLM to output the abstract structure of the refactored code, i.e., class architecture, method signature, global properties, etc. ❸ **LLM-based Methods Re-Organizing**: From the given abstract structure of refactored code, we can obtain a list of refactored class/method signatures that need to fill in. Then we prompt the LLMs to match the refactored class/method signatures to the origin signatures. ❹ **Tool-based Methods Moving**: We develop a tool to do the automatic move work. In this study, God Class and Refused Bequest are suitable for class-level code smell refactoring. We release the code-mitigation tool on our website [19]. **Method-Level Refactoring** has an impact on multiple methods. Due to the limited refactoring scope, we directly request the LLM to provide the complete code of the refactored methods. In this study, Feature Envy is suitable for method-level code smell refactoring.

# 4 EXPERIMENTAL DESIGN

We address the following three research questions to evaluate the performance of iSMELL:

RQ1: How does the iSMELL perform compared to the state-of-the-art code smell detection baselines?

RQ2: How does each individual component of iSMELL contribute to the overall performance?

RQ3: What is the perceived and quantitative quality of refactored code generated by iSMELL?

## 4.1 Study Subjects

*4.1.1 Code Smells.* Lacerda et al. [20] reported on 10 of the most prevalent code smell types. We have selected three complex code smells from these, which are categorized as class-level and method-level smells based on their impact scope. The three code smells were

chosen for their representativeness and the specific challenges they present. They encompass a range of issues, from method-level to class-level concerns, such as the concentration of responsibilities, improper dependencies, and the misapplication of inheritance principles. Addressing each code smell requires a tailored approach: while complex smells may necessitate advanced detection techniques, simpler ones, like 'Long Method', can often be effectively identified using rule-based systems. For instance, DesigniteJava detects 'Long Method' by flagging methods that exceed 100 lines of code. Their definitions are as follows:

**God Class:** It refers to a class that assumes too many responsibilities. Typically, it is a large class that consolidates control and supervision of many disparate objects. **Feature Envy:** This occurs when a method excessively relies on another class. It suggests that the features of another class might be better placed elsewhere. **Refused Bequest:** This situation arises when a subclass inherits from a parent class but does not need all the behaviors provided by the parent class. Thus, the subclass rejects some behaviors inherited from the parent class.

*4.1.2 Expert Tools for Code Smell Detection.* To ensure the effectiveness of iSMELL in detecting code smells, we have selected the following expert toolsets not only because they are currently popular code analysis toolsets but also because we have considered the diversity and sources of their internal algorithms. Detailed lists of the supported smell types for these toolsets are compiled and summarized in Table 1 for comparison and reference.

**PMD** [22]: It is an extensible, multi-language static code analyzer that comes with over 400 built-in rules to detect common programming defects. **JDeodorant** [14]: It is an Eclipse plugin that integrates code smell detection with recommendations for refactoring strategies. **Organic** [38]: It is an Eclipse plugin that supports the detection of multiple code smells. It analyzes code by comparing collected metrics against internally set thresholds to identify these smells. **DesigniteJava** [39]: It is a code quality assessment tool that detects numerous design and implementation smells and computes a variety of object-oriented metrics. **JSpIRIT** [40]: It supports the detection of various code smells and facilitates the identification of code smell aggregates, as well as the prioritization of these smells for remediation. **FeTruth** [13]: It is a tool that employs deep learning techniques to detect instances of Feature Envy in code, and it also recommends refactoring strategies to address this particular code smell. **JMove** [41]: It is an Eclipse plugin that identifies opportunities for the Move Method refactoring.

## 4.2 Dataset

*4.2.1 Data Collection.* The training data originates from three widely-used datasets, which were respectively constructed by Palomba *et al.* [42], Fontana *et al.* [43], and Khomh *et al.* [44]. The composition of these datasets is illustrated in Table 2.

**Table 1: The types of code smells supported by the tools**

| Name | God Class | Feature Envy | Refused Bequest |
|---|---|---|---|
| PMD | ✓ | × | × |
| JDeodorant | ✓ | ✓ | × |
| Organic | ✓ | ✓ | ✓ |
| DesigniteJava | ✓ | × | × |
| JSpIRIT | ✓ | ✓ | ✓ |
| FeTruth | × | ✓ | × |
| JMove | × | ✓ | × |

**Table 2: Statistics of the code smell datasets**

| | Palomba *et al.* [42] | Fontana *et al.* [43] | Khomh *et al.* [44] | Total | Training | Test |
|---|---|---|---|---|---|---|
| God Class | 94 | 0 | 102 | 196 | 2352 | 39 |
| Refused Bequest | 0 | 0 | 248 | 248 | 2976 | 50 |
| Feature Envy | 25 | 185 | 0 | 210 | 2520 | 42 |

*4.2.2 Ground Truth.* We employed the seven professional code smell detection toolsets mentioned in Section 4.1.2. Subsequently, we applied these toolsets to each instance within the dataset to conduct smell detection and ascertain the ground truth. If a particular smell was identified by the expert toolsets, the corresponding instance was labeled as 1; conversely, if the respective smell was not detected, it was labeled as 0.

*4.2.3 Data Augmentation.* We evaluated the performance of our model using five-fold cross-validation. Considering the substantial similarity between linear extrapolation and linear interpolation, we combine them as one method for implementation. For each augmentation technique, we conduct five iterations. During training, we utilized all four augmentation methods and augmented the training data fifteen times. Each augmentation involved resampling the data and coefficients to expand the original dataset. The size of the augmented training data compared to the training dataset is shown in Table 2.

## 4.3 Baselines

The seven expert toolsets involved in the MoE also serve as baseline toolsets. In addition, we have selected four state-of-the-art LLMs, which are introduced as follows:

**GPT-3.5 turbo** [45]: It is a natural language processing model introduced by OpenAI, based on the GPT (Generative Pre-trained Transformer) architecture. It supports a maximum context input of 16k tokens. **GPT-4.0 turbo** [46]: It represents the latest generation of LLM technology. Compared to GPT4.0, it can accept more context inputs, supporting a maximum context input of 128k tokens. **LLAMA3-70B** [47]: Pre-trained on over 1.5 trillion data tokens, it has achieved state-of-the-art performance levels on a wide range of tasks, establishing the LLAMA series as top-tier open-source large language models applicable to various application and deployment scenarios. It supports a maximum context input of 1048K tokens. **CodeLlama-34B** [48]: Meta has fine-tuned the Llama2 version model for code programming tasks, resulting in CodeLlama-34B. It supports a maximum context input of 100k tokens.

## 4.4 Evaluation Metrics

*4.4.1 Metrics for Smell Detection.* This study investigates a multi-label binary classification problem aimed at identifying whether multiple specific code smells exist in code samples. Each code sample may be associated with multiple binary subtasks, where the presence or absence of each smell constitutes an independent binary classification problem. Therefore, the presence of each smell is considered as a positive instance of that category, while its absence is considered as a negative instance. True Positives (TPs)

**Table 3: Performances of iSMELL and expert tools baselines**

| | PMD | JDeodorant | Organic | Designite Java | JSpIRIT | FeTruth | JMove | iSMELL | Oracle |
|---|---|---|---|---|---|---|---|---|---|
| **Accuracy** | 27.38% | 45.81% | 77.68% | 31.12% | 65.60% | 19.68% | 18.57% | **81.53%** | 87.92% |
| GodClass | 82.14% | 82.14% | 86.73% | 93.37% | 72.96% | N/A | N/A | **95.14%** | 99.44% |
| FeatureEnvy | N/A | 56.19% | 70.95% | N/A | 53.33% | 59.05% | 55.71% | **72.24%** | 81.90% |
| RefusedBequest | N/A | N/A | 76.21% | N/A | 70.16% | N/A | N/A | **78.99%** | 84.68% |
| **F1** | 27.69% | 36.42% | 68.50% | 31.22% | 46.30% | 8.19% | 2.02% | **75.17%** | 83.91% |
| GodClass | 83.06% | 81.32% | 86.17% | 93.66% | 74.68% | N/A | N/A | **95.52%** | 98.58% |
| FeatureEnvy | N/A | 6.12% | **64.00%** | N/A | 14.04% | 24.56% | 6.06% | 59.98% | 75.00% |
| RefusedBequest | N/A | N/A | 45.87% | N/A | 35.09% | N/A | N/A | **60.89%** | 70.31% |
| **Recall** | 24.60% | 25.54% | 54.80% | 29.91% | 31.99% | 4.91% | 1.05% | **65.31%** | 72.28% |
| GodClass | 73.79% | 69.81% | 75.70% | 89.72% | 62.77% | N/A | N/A | **92.41%** | 97.20% |
| FeatureEnvy | N/A | 3.16% | **52.75%** | N/A | 8.42% | 14.74% | 3.16% | 47.17% | 60.00% |
| RefusedBequest | N/A | N/A | 30.12% | N/A | 24.10% | N/A | N/A | **49.01%** | 54.22% |
| **Precison** | 31.67% | 63.45% | **91.30%** | 32.65% | 83.78% | 24.56% | 25.00% | 90.71% | 100.00% |
| GodClass | 95.00% | 97.37% | **100.00%** | 97.96% | 92.19% | N/A | N/A | 99.05% | 100.00% |
| FeatureEnvy | N/A | **100.00%** | 81.36% | N/A | 42.11% | 73.68% | 75.00% | 83.67% | 100.00% |
| RefusedBequest | N/A | N/A | **96.15%** | N/A | 64.52% | N/A | N/A | 82.13% | 100.00% |

and True Negatives (TNs) are defined for each smell category, indicating that the model correctly predicts the presence or absence of the smell. False Positives (FPs) and False Negatives (FNs) are also defined accordingly, for each smell category, representing instances where the model incorrectly predicts the presence of a smell or erroneously ignores a smell that is actually present. Concerning performance evaluation metrics [49], this study employs ***F1 score***, ***Overall Accuracy***, ***Precision***, and ***Recall*** for quantifying the model's performance.

*4.4.2 Metrics for Smell Refactoring.* In manually inspecting whether and to what extent the refactored code should be accepted, we employ a scoring system ranging from 0 to 4. The detailed scoring criteria are as follows: **0 points:** There are significant differences between the suggested refactoring code and the actual refactoring code, such as missing core method logic, posing a threat to major functionality, or the code smell remaining unresolved after refactoring. Additionally, for false positives—samples where code was predicted to need refactoring but actually did not, we instructed participants to assign a score of 0. **1 point:** Significant differences are observed, including missing essential functions or fundamental changes in the code's logical structure. **2 points:** Moderate mismatches are noted, such as incomplete instructions in methods or adjustments in logical structure. **3 points:** Minor differences are present, such as variable renaming or small omissions of non-essential statements. **4 points:** All aspects of the suggested and actual refactoring codes are identical. In quantitatively assessing the performance of code refactoring, we observe the variation of the representative code-smell metrics before and after refactoring. These metrics include WMC, DIT, NOC, CBO, RFC, and LCOM (as introduced in Section 3.1.1).

## 4.5 Implementation details

For all LLMs, we set their parameter Temperatures to 0.2. Furthermore, we impose a limit of 1500 for the MAX TOKEN count in responses from all LLMs. Following previous studies [35], for linear interpolation and extrapolation, we sample the $\alpha$ value from a uniform distribution $U \sim (0.9, 1.1)$. For binary interpolation, we sample the $\alpha$ value from a Bernoulli distribution $B(p = 0.25)$. As for Gaussian scaling, we sample the $\beta$ value from a normal distribution $N(0, 0.1)$. We set the number of training rounds to 70. The training of the MoE model was conducted on a GeForce RTX 4070TI GPU. When performing code smell detection, Llama3 and CodeLlama were run on an A800 GPU with 80GB of VRAM.

**Table 4: Performances of iSMELL and LLMs baselines**

| | GPT3.5 | GPT4.0 | LLaMA3.0 | CodeLlama | **iSMELL** |
|---|---|---|---|---|---|
| **Accuracy** | 49.39% | 63.46% | 60.55% | 53.67% | **81.53%** |
| GodClass | 35.20% | 77.55% | 73.47% | 42.35% | **95.14%** |
| FeatureEnvy | 58.10% | 55.71% | 54.29% | 54.29% | **72.24%** |
| RefusedBequest | 53.23% | 58.87% | 55.65% | 62.10% | **78.99%** |
| **F1** | 30.61% | 55.66% | 52.57% | 19.20% | **75.17%** |
| GodClass | 16.99% | 80.53% | 71.74% | 1.74% | **95.52%** |
| FeatureEnvy | 33.33% | 4.12% | 7.69% | 11.11% | **59.98%** |
| RefusedBequest | 39.58% | 52.78% | 57.03% | 38.16% | **60.89%** |
| **Recall** | 25.61% | 52.63% | 50.18% | 12.63% | **65.31%** |
| GodClass | 12.15% | 85.05% | 61.68% | 0.93% | **92.41%** |
| FeatureEnvy | 23.16% | 2.11% | 4.21% | 6.32% | **47.17%** |
| RefusedBequest | 45.78% | 68.67% | **87.95%** | 34.94% | 49.01% |
| **Precison** | 38.02% | 59.06% | 55.21% | 40.00% | **90.71%** |
| GodClass | 28.26% | 76.47% | 85.71% | 12.50% | **99.05%** |
| FeatureEnvy | 59.46% | **100.00%** | 44.44% | 46.15% | 83.67% |
| RefusedBequest | 34.86% | 57.89% | 42.20% | 42.03% | **82.13%** |

# 5 RESULTS

## 5.1 Advantages on Code Smell Detection

*5.1.1 Prompt Design for LLM-baselines.* To fairly assess the performance disparity between our method and LLMs in smell detection, we meticulously designed the prompts. We removed superfluous comments from the code to enable the LLMs to accommodate and process more contextual information. Furthermore, when code involves inheritance relationships, we fed the parent class code into the LLMs concurrently, thereby enhancing their comprehension and handling of the inheritance structure. The prompt for smell detection mainly consists of three parts: *instruction*, *definition*, and *code to be analyzed*. ❶ Instruction: The instruction section prompts the LLMs to systematically examine the code, aiming to identify three typical code smells, including God Class, Feature Envy, and Refused Bequest. ❷ Definition: The definition part elaborates on the specific meanings of these three smells to ensure the model has an accurate understanding of them. ❸ Code to be analyzed: In the section for the code to be analyzed, the target code segment for

smell detection is directly embedded. Due to the lengthy nature of the prompt for smell detection, we will disclose the specific details along with the code and dataset [19].

*5.1.2 Results.* Assessment results are shown in Table 3 and Table 4, where the "Oracle" column data represents the theoretical upper limit of detection performance under the optimal tool selection. It is worth noting that due to the independent operation of LLMs and their lack of reliance on expert toolsets, their detection efficiency theoretically can surpass this upper limit. Comparison between iSMELL and expert toolsets is as follows.

**iSMELL vs. Individual Expert Tool**. For the detection of God Class, iSMELL achieved an F1 score of 95.52%, surpassing the performance of all expert toolsets. Specifically, iSMELL outperformed the best expert tool, DesigniteJava (93.66%), by 1.86%. iSMELL's accuracy was 95.14%, a 1.77% improvement over DesigniteJava (93.37%). In detecting Feature Envy, iSMELL attained an F1 score of 59.98%, surpassing most expert toolsets but falling short of Organic (64.00%). iSMELL's accuracy was 72.24%, a 1.29% improvement over the best expert tool, Organic (70.95%). For Refused Bequest, iSMELL achieved an F1 score of 60.89%, surpassing the performance of all expert toolsets. Specifically, iSMELL outperformed the best expert tool, Organic (45.87%), by 15.02%. iSMELL's accuracy was 78.99%, a 2.78% improvement over Organic (76.21%).

**iSMELL vs. LLMs**. For detecting God Class, iSMELL achieved an F1 score of 95.52%, surpassing all LLMs. Specifically, iSMELL surpassed the best-performing LLM, GPT4.0 (F1 score of 80.53%), by 18.61%. The accuracy of iSMELL was 95.14%, marking a 22.68% increase over GPT4.0's accuracy of 77.55%. For identifying FeatureEnvy, iSMELL registered an F1 score of 59.98%, surpassing all other LLMs. Notably, iSMELL improved upon the top LLM, GPT3.5 (which had an F1 score of 33.33%), by an impressive 79.96%. The accuracy of iSMELL reached 72.24%, representing a significant 24.34% enhancement over GPT3.5's accuracy of 58.10%. In the case of RefusedBequest, iSMELL's F1 score stood at 60.89%. iSMELL outperformed the best LLM in this category, Llama (F1 score of 57.03%), by 6.77%. The accuracy of iSMELL was 78.99%, indicating a substantial 27.20% improvement over CodeLlama's accuracy of 62.10%.

**Answering RQ1:** The iSMELL achieves a significant improvement in F1 score and accuracy compared to state-of-the-art large language models, with an average increase of 35.05% and 28.47%, respectively. Compared to the most advanced expert toolsets, the average improvement is 9.74% and 4.96%, respectively.

## 5.2 Ablation Study on Code Smell Detection

*5.2.1 Variants.* To evaluate core components contributions, we obtained two variants: (1) **iSMELL w/o CM**, removing the CK metric vectors integrated into the data preprocessing stage. (2) **iSMELL w/o GN**, removing the gating network responsible for weight allocation in the MoE model, assigning equal weights to each expert toolset. We trained these variants using the same experimental setup as iSMELL and evaluated performance on the same test sets. The detailed analysis of experiments is outlined below.

**Table 5: Ablation study on iSMELL**

| | iSMELL | iSMELL w/o CM | iSMELL w/o GN |
|---|---|---|---|
| **Accuracy** | **81.53%** | 79.26% | 77.72% |
| GodClass | **95.14%** | 88.66% | 87.66% |
| FeatureEnvy | **72.24%** | 72.03% | 68.01% |
| RefusedBequest | **78.99%** | 77.70% | 78.13% |
| **F1** | **75.17%** | 71.14% | 68.02% |
| GodClass | **95.52%** | 88.42% | 87.38% |
| FeatureEnvy | **59.98%** | 59.11% | 50.83% |
| RefusedBequest | **60.89%** | 56.34% | 57.25% |
| **Recall** | **65.31%** | 59.15% | 55.33% |
| GodClass | **92.41%** | 81.72% | 79.37% |
| FeatureEnvy | **47.17%** | 45.46% | 37.59% |
| RefusedBequest | **49.01%** | 44.06% | 45.01% |
| **Precison** | **90.71%** | 89.97% | 90.18% |
| GodClass | **99.05%** | 96.78% | 97.84% |
| FeatureEnvy | 83.67% | **86.82%** | 82.95% |
| RefusedBequest | 82.13% | 82.57% | **83.49%** |

*5.2.2 Results.* Table 5 shows the performance of iSMELL and its two variants. It can be seen that removing these two components results in a significant performance decrease. Specifically, when comparing iSMELL with iSMELL w/o CM, removing the CK metric resulted in a decrease in accuracy by 2.27% and a decrease in F1 score by 4.03%. When comparing iSMELL with iSMELL w/o GN, after the gate control network became ineffective, the accuracy and F1 scores decreased by 3.81% and 7.15%, respectively. Removing metrics-induced performance degradation explicitly confirms the indispensability of these static code metrics in identifying code smells. They provide crucial perspectives on code structure and potential problematic areas for the model. Experimental results highlight the specificity and complementarity of different expert toolsets in smell detection, with the dynamic weight configuration mechanism at the gating network proving to be a key factor in enhancing model performance.

**Answering RQ2:** Both the code metrics and the gating network components have positive contributions to the performance of iSMELL. Gated networks contribute more significantly. After removing the gated networks, the model's accuracy and F1 scores decreased by 3.81% and 7.15%.

## 5.3 Performance on Code Smell Refactoring

**Table 6: Quantitative metrics results**

| | Refused Bequest | | God Class | | Feature Envy | |
|---|---|---|---|---|---|---|
| | Before | After | Before | After | Before | After |
| **CBO** | 3.17 | 1.83 (↓ 1.34) | 41.38 | 22.00 (↓ 19.38) | 32.94 | 31.88 (↓ 1.06) |
| **WMC** | 7.56 | 5.83 (↓ 1.73) | 382.25 | 79.06 (↓ 303.19) | 239.24 | 215.88 (↓ 23.36) |
| **DIT** | 1.89 | 1.33 (↓ 0.56) | 3.5 | 2.25 (↓ 1.25) | 1.47 | 1.47 (→) |
| **NOC** | 0.28 | 0.17 (↓ 0.11) | 0.19 | 0.19 (→) | 0.29 | 0.29 (→) |
| **RFC** | 9.78 | 7.11 (↓ 2.67) | 145.31 | 48.38 (↓ 96.93) | 110.53 | 100.71 (↓ 9.82) |
| **LCOM** | 9.56 | 4.72 (↓ 4.84) | 1897.19 | 159.06 (↓ 1738.13) | 798.24 | 798.12 (↓ 0.12) |

*5.3.1 Quantitative Analysis.* According to Table 6, we can clearly observe the average changes in code quality attributes before and after refactoring. Before refactoring, God Classes often have a large number of methods, bear too much responsibility, and are highly

coupled with other classes, leading to poor cohesion. After refactoring, these methods are distributed among multiple classes, significantly reducing the number and complexity of methods in individual classes. The responsibilities of each class become clearer, reducing unnecessary coupling. As a result, metrics such as WMC, CBO, RFC, and LCOM experience significant decreases after God Class refactoring. Since God Class refactoring often does not involve adjustments to inheritance relationships, changes in DIT and NOC are minimal. Since Refused Bequest refactoring involves removing unnecessary inheritance levels, DIT will decrease here. Additionally, refactoring typically involves subclasses no longer inheriting methods from the superclass that they don't need or aren't suitable for, indirectly reducing the WMC, CBO, RFC, and LCOM of subclasses. However, the impact is usually smaller compared to God Class refactoring because it typically affects only a few related classes. This refactoring has a limited direct impact on the number of subclasses, so NOC changes are minimal. Feature Envy refactoring mainly addresses excessive dependencies of methods on other classes by moving methods or creating local extension classes to improve. This directly reduces the WMC of the original class while possibly slightly decreasing CBO and RFC because it reduces direct dependencies on external classes. Feature Envy refactoring has no effect on DIT and NOC as its main focus is on method location rather than class structure. LCOM may slightly improve as method relocation enhances the cohesion of the class where it's moved.

*5.3.2 Human Evaluation.* **Procedure**. In our study, focusing on the code smell data predicted by the MoE model, we conducted in-depth refactoring attempts on 25 randomly selected instances for each category of code smells. Out of a total of 75 samples, there are some false-positive samples, i.e., samples predicted to need refactoring but actually not. The participants score these samples as 0. We enlisted the participation of three individuals, comprising two master's degree students and one experienced developer. All participants possess a minimum of three years of experience in Java development, with one having over five years of experience. It is noteworthy that these participants are not co-authors of this paper. Agreements were signed with each participant, explicitly mandating objective annotation or evaluation. Each evaluator was tasked with reviewing 50 instances, ensuring that each piece of code underwent independent assessment by at least two reviewers. In cases of discrepancy, we established a discussion mechanism, obliging the concerned reviewers to communicate thoroughly until consensus was reached, thereby enhancing the consistency and reliability of the evaluation outcomes. Furthermore, feedback from authoritative code smell detection toolsets served as a pivotal reference for assessing the quality of the code before and after refactoring. If the refactored code from the large language model is accepted, we also evaluate the effectiveness of large language model refactoring using the metric of Content consistency.

**Results**. Figure 3 illustrates the results of human evaluation, depicting the accuracy of iSMELL's refactoring using violin plots. Overall, the code generated by iSMELL is acceptable in most cases and effectively preserves the core functionality of the code. Across the three smells (Feature Envy, God Class, Refused Bequest), the average refactoring accuracies of our method are 2.60, 2.08, and 2.36, with acceptance rates of 68%, 64%, and 72%, respectively. After conducting the God Class refactoring, we did not observe any

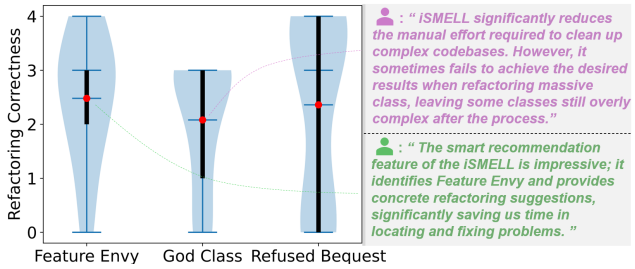


**Figure 3: Human evaluation scores and comments**

samples meeting the 4-point criterion. This phenomenon can be attributed to the excessive redundancy within the God Class code itself. Due to the token count limitations faced by LLMs during output generation, this frequently leads to incompleteness in content, thereby affecting the evaluation results. Compared to other smells, the proportion of zero scores in the refactoring results for Refused Bequest is higher. This is because six false positive cases in the total sample mostly belong to the Refused Bequest type, leading to an increase in low scores in the evaluation. During the evaluation of the refactoring tool, we collected feedback from several participants, as shown in Figure 3. These comments illustrate the dual nature of iSMELL: while it is effective in many cases, particularly with "Feature Envy," there is room for improvement when it comes to handling larger, more complex classes.

In summary, integrating detection and refactoring capabilities within the tool streamlines the workflow, offering developers a convenient and efficient way to maintain high-quality code standards.


**Figure 4: A bad case for refactoring *God Class*. Despite refactoring, the process wasn't thorough enough and the *God Class* still persists, yelling an interactive need for multiple iterative refactorings.**

*5.3.3 Case Study.* To further investigate why iSMELL scored lower in human evaluation, we conducted a qualitative analysis on a randomly selected bad case under score 1. As shown in Figure 4, the God Class 'Project' contained 1,992 lines of code (LOC). iSMELL put forth an initial suggestion for the class structure, which the participants found to be suitable. However, the outcome after the LLM's mapping was less satisfactory; even after being divided into seven

classes, the original class still retained 900 LOC. The participants pointed out that although the split and mapping were somewhat accurate, the overall structure remained excessively large and complex. Consequently, they rejected the refactoring result, signaling the necessity for further refinement to attain an acceptable level of code quality and class granularity.

Our findings suggest that the difficulty in refactoring longer classes may be due to the initial concentration of responsibilities, making it challenging to entirely eliminate code smells in a single refactoring effort. The complexity and interconnections within the components of longer classes can lead to residual smells after refactoring. To enhance the effectiveness of refactoring, our future work will focus on implementing multiple iterative refactorings. By doing so, we aim to address any residual smells that remain after the initial refactoring, ensuring a more thorough code cleanup.

**Answering RQ3:** In both human evaluation and quantitative assessment, the code generated by iSMELL is generally acceptable and effectively preserves the core functionality of the code, with a noticeable improvement in the quality attributes of the refactored code.

# 6 DISCUSSION

## 6.1 Advantages and Limitations

**Advantages of iSMELL.** ❶ **Scalability**: In this work, we focus on three representative code smells that pose significant challenges, as described in Section 4.1.1. Although we evaluated a limited number of code smells, our method, iSMELL, is scalable and capable of addressing a wide range of code smells beyond those selected. The number of smells iSMELL can address is determined by the expert tools integrated into iSMELL. By simply augmenting the rows of the MoE matrix, we can incorporate additional detection tools, thereby expanding our detection capabilities. Similarly, by increasing the columns in the MoE matrix, we can target more code smells, thus broadening our scope without compromising efficiency or performance. ❷ **Comprehensiveness**: By integrating various detection algorithms and expert tools, iSMELL establishes a robust platform for code smell detection. This integration overcomes the limitations of individual tools, enabling the identification of a broader range of code smells. ❸ **Flexibility**: Utilizing a MoE framework, iSMELL demonstrates adaptability and scalability. This framework facilitates the seamless integration of emerging detection tools and technologies, allowing it to stay current with advancements in software engineering. The modular design of the MoE structure ensures that updates and replacements can be easily managed, maintaining the system's relevance and upgradability over time. ❹ **Optimized Performance**: iSMELL leverages the strengths of diverse tools, selecting the most appropriate algorithm for specific scenarios. This approach enhances both accuracy and efficiency in addressing complex or atypical code smells. By selecting the optimal solutions from its extensive toolkit, iSMELL minimizes false positives and maximizes the detection of genuine issues.

**The potential and current state of refactoring tools.** The potential (upper bound) of our approach's performance in code smell detection, referred to as the Oracle, is directly influenced by the integrated detection tools. Significant performance disparities among these tools impact the effectiveness of our method, particularly in detecting complex smells such as feature envy. Additionally, the availability of tools for detecting certain smells constrains iSMELL's overall effectiveness. Therefore, developing more efficient detection tools for specific types of smells is both necessary and urgent. Such advancements are crucial for enhancing the overall performance of methods that utilize LLMs to integrate detection tools, like iSMELL.

**The limitations and potentials of LLMs in software refactoring.** While LLMs have demonstrated strong capabilities in code generation, there remains a significant gap in their ability to effectively perform code refactoring in practice. LLMs excel in handling refactoring tasks that do not require extensive contextual or repository-level understanding but struggle with complex code transformations. Another limitation is the restriction on input/output token length, which often necessitates multiple interactions to pass comprehensive contextual information. This can result in issues such as selective forgetting and increased interaction costs. LLMs, being pattern recognition and generation systems for static text, lack the ability to directly perceive the actual runtime state of code. Dynamic code analysis, which involves observing a program's behavior at runtime—tracking variable states, function calls, memory usage, concurrent behaviors, and more—presents inherent challenges for LLMs. As a result, LLMs face significant limitations when it comes to performing dynamic code analysis.

## 6.2 Potential Application Scenarios

The iSMELL that we propose leverages a MoE architecture to integrate various expert detection toolsets for comprehensive code inspection and enhances LLMs to refactor code. We argue that this framework could be further extended to other complex software engineering tasks. For example, **vulnerability detection** requires human experts to manually define the features of vulnerabilities, which is a laborious and tedious task. As a result, various automated vulnerability detection toolsets have emerged, some from the commercial sector, such as Trivy [50], OpenVAS [51], Clair [52], and others from academia [53–56]. Similar to the challenges in code smell detection, these expert tools are diverse and have deviations in certain types of vulnerabilities, and LLMs have limitations in terms of static or dynamic code analysis. By integrating expert toolsets from vulnerability detection and the LLMs, we not only cover a broader range of vulnerability types but also potentially explain or patch the detected vulnerability via LLMs. There has been a wide range of software engineering tasks that are worth exploring, such as code review, automated program repair, *etc.*

On the time efficiency aspect, our approach is a trade-off solution on applying expert toolsets and LLMs. Training an MoE model for 70 epochs approximately takes 5 minutes, and inferring a single data entry requires less than 0.02 seconds. Refactoring one complex smell code instance takes around 30 seconds. Overall, our proposed method constitutes a lightweight and time-efficient solution for code refactoring, blending LLMs with specialized tooling. When migrating this approach to other application scenarios, the time efficiency is affected by factors such as the efficiency of domain-specific expert toolsets and the complexity of data preprocessing within the applied LLMs.

We believe that our proposed solution can provide new insights for better leveraging LLMs and existing methods to tackle complex software tasks more effectively.

## 6.3 Threats to validity

*6.3.1 Internal Validity.* To obtain accurate assessment metrics, we adopted widely recognized open-source metric extraction toolsets in the research field, which are also commonly used by other scholars. However, considering the large number of projects covered in our analysis dataset, some metric indicators may have encountered miscalculations due to technical obstacles during the parsing. Specifically, during our error troubleshooting process, instances of inaccurate metric calculations were identified. Nevertheless, given that the expert toolsets we selected have undergone rigorous testing by the community and are widely trusted, we believe that the proportion of metric values with calculation deviations is minimal and will not significantly impact the overall research conclusions.

*6.3.2 External Validity.* We selected three representative datasets that encompass numerous software development projects closely related to modern industry. These datasets not only exhibit diversity in scale but also span across various business domains, aiming to mitigate concerns regarding result generalizability. However, all projects included in the analysis are of open-source nature and are uniformly written in the Java language, which inevitably limits the generalizability of our conclusions to industry projects using other programming languages or non-open-source environments.

# 7 RELATED WORK

**Code Smell Detection.** Most existing code smell detectors are heuristic-based [14, 15, 17, 40, 57]. However, recent research has shifted towards integrating deep learning techniques [12, 13], aiming to enhance detection accuracy and generalization through automatic feature learning. ***1) Heuristics-based Approaches.*** Tsantalis *et al.* [14, 15] introduced JDeodorant, which uses "distance" to quantify method-class interaction frequencies, helping identify "Feature Envy" and suggesting "Move Method" refactoring. For "God Classes", JDeodorant employs cluster analysis to detect complexity and recommend decomposition by extracting methods and attributes into new classes. Moha *et al.* [58] developed DECOR, a heuristic tool that detects code smells using predefined rules. For God Classes, DECOR analyzes metrics like LCOM5, method-/attribute counts, and one-to-many associations, comparing them against its detection rules. ***2) Deep Learning-based Approaches.*** Liu *et al.* [12] were the first to apply deep learning technology to detect and address Feature Envy. Their model architecture cleverly combines CNNs and LSTMs. They also innovatively developed a technique for automatically generating training data. Additionally, Liu *et al.* [13] mined historical change data from open-source projects, extracted representative positive and negative samples, and combined heuristic rules with learning-based filters to identify smells. Li and Zhang [59] parse code into abstract syntax trees with control and data flow edges, then employed bidirectional LSTM networks with graph convolutional networks and attention mechanisms to predict various smells. Yu [60] collects code metrics and calling relationships to convert into graphs, concurrently introducing a graph augmenter to obtain enhanced graphs, then applies graph neural networks to detect Feature Envy. Building upon these studies, we integrate heuristic-based with learning-based smell detection techniques in our solution, aspiring to establish a highly scalable approach. This approach not only encompasses a broader spectrum of smell types but also bolsters detection capability through technological complementarity.

**Code Smell Refactoring.** In addressing the challenges of optimizing software codebases, a myriad of refactoring strategies have been extensively explored and implemented. Each study typically focuses on tailored methodologies to address specific issues, aiming to enhance code structure and quality across multiple levels, including classes, methods, and variables. The feTruth developed by Liu *et al.* [13] utilizes a combination of heuristic and learning-based filters to exclude non-Feature Envy cases, then employs a neural network classifier to predict potential refactoring suggestions. In practice, feTruth also demonstrates high accuracy in recommending target classes. Aniche *et al.* [61] frame the refactoring recommendation problem as a binary classification task. Through analysis of over two million real refactoring instances, they validate the effectiveness of various machine learning algorithms in predicting diverse refactoring operations. Ma *et al.* [62] leverage the pre-trained model CodeT5 [63] to predict the target classes for method movement by analyzing the relationship between methods and potential target classes. Ouni *et al.* [64] proposed a multi-objective optimization algorithmic approach to identify sequences of refactorings, leveraging the development history collected from software projects to reinforce the refactoring outcomes. While the bulk of current research centers on proposing refactoring strategy suggestions, primarily concentrating on operations at the method and class levels, our approach harnesses the potent learning capabilities of LLMs to directly output refactored code snippets, achieving a more nuanced form of code-level refactoring.

# 8 CONCLUSION AND FUTURE WORK

This paper introduces iSMELL, a method for comprehensive code smell detection through a Mixture of Experts (MoE) architecture, that utilizes various code smell detection toolsets, and enhances Large Language Models (LLMs) to refactor identified code smells. Evaluation results demonstrate that iSMELL outperforms other baselines in code smell detection. We employ both quantitative and human evaluations of LLM-refactored code, indicating that iSMELL satisfactorily refactors identified code smells and improves code quality attributes. In the future, we plan to establish a repository for refactored code. This repository aims to leverage search algorithms to identify the refactoring examples that are most similar to the smelly code, upon which prompts will be constructed based on these examples. We also plan to systematically compare the effectiveness of our refactoring approach with existing mainstream methods.

# ACKNOWLEDGMENTS

We sincerely appreciate the anonymous reviewers for their constructive and insightful suggestions for improving this manuscript. This work was supported by the grant from the National Natural Science Foundation of China (Grant Nos.62272445 and 62332001) and Huawei.

# REFERENCES


[1] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[2] José Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow. 2022. Code smells detection and visualization: a systematic literature review. *Archives of Computational Methods in Engineering* 29, 1 (2022), 47–94.

[3] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[4] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[5] 2023. An overview of Bard: an early experiment with generative AI. https://ai.google/static/documents/google-about-bard.pdf.

[6] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An empirical study on the usage of transformer models for code completion. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4818–4837.

[7] Aleksandra Eliseeva, Yaroslav Sokolov, Egor Bogomolov, Yaroslav Golubev, Danny Dig, and Timofey Bryksin. 2023. From commit message generation to history-aware commit message completion. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 723–735.

[8] Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. 2024. Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example. *arXiv preprint arXiv:2402.07138* (2024).

[9] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[10] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. 2023. Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects. *Authorea Preprints* (2023).

[11] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169* (2023).

[12] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2019. Deep learning based code smell detection. *IEEE transactions on Software Engineering* 47, 9 (2019), 1811–1837.

[13] Bo Liu, Hui Liu, Guangjie Li, Nan Niu, Zimao Xu, Yifan Wang, Yunni Xia, Yuxia Zhang, and Yanjie Jiang. 2023. Deep Learning Based Feature Envy Detection Boosted by Real-World Examples. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 908–920.

[14] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. 2007. Jdeodorant: Identification and removal of feature envy bad smells. In *2007 ieee international conference on software maintenance*. IEEE, 519–520.

[15] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering*. 1037–1039.

[16] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2009. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* 36, 1 (2009), 20–36.

[17] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* 108 (2019), 115–138. https://doi.org/10.1016/j.infsof.2018.12.009

[18] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. 1991. Adaptive mixtures of local experts. *Neural computation* 3, 1 (1991), 79–87.

[19] 2024. https://github.com/iSMELL2024/iSMELL.

[20] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167 (2020), 110610.

[21] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering* 28, 7 (2002), 654–670.

[22] 2024. PMD - source code analyzer. https://github.com/pmd/pmd.

[23] 2024. A code segment having Feature Envy. https://github.com/apache/hbase/blob/34487ecc6f90f486325b625a6909de888008f4b2/src/main/java/org/apache/hadoop/hbase/master/AssignmentManager.java#L527-L641.

[24] 2024. A code segment without Feature Envy. https://github.com/ArtOfIllusion/ArtOfIllusion/blob/2d66694e95fec8d8d0c38c01505c24ad1df821b1/ArtOfIllusion/src/artofillusion/object/SplineMesh.java#L486-L587.

[25] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1646–1656.

[26] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1282–1294.

[27] Wenbo Hu, Yifan Xu, Yi Li, Weiyue Li, Zeyuan Chen, and Zhuowen Tu. 2024. Bliva: A simple multimodal llm for better handling of text-rich visual questions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 2256–2264.

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[29] Tim Sonnekalb, Bernd Gruner, Clemens-Alexander Brust, and Patrick Mäder. 2022. Generalizability of code clone detection on codebert. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–3.

[30] Yuchen Cai, Aashish Yadavally, Abhishek Mishra, Genesis Montejo, and Tien Nguyen. 2024. Programming Assistant for Exception Handling with CodeBERT. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[31] Xin Zhou, DongGyun Han, and David Lo. 2021. Assessing generalizability of codebert. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 425–436.

[32] 2023. Java Code Metrics Collection Tool. https://github.com/mauricioaniche/ck.

[33] Shyam R Chidamber and Chris F Kemerer. 1991. Towards a metrics suite for object oriented design. In *Conference proceedings on Object-oriented programming systems, languages, and applications*. 197–211.

[34] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[35] Haochen Li, Chunyan Miao, Cyril Leung, Yanxian Huang, Yuan Huang, Hongyu Zhang, and Yanlin Wang. 2022. Exploring representation-level augmentation for code search. *arXiv preprint arXiv:2210.12285* (2022).

[36] Thierry Blu, Philippe Thévenaz, and Michael Unser. 2004. Linear interpolation revitalized. *IEEE Transactions on Image Processing* 13, 5 (2004), 710–719.

[37] Bridget Nyirongo, Yanjie Jiang, He Jiang, and Hui Liu. 2024. A Survey of Deep Learning Based Software Refactoring. *arXiv preprint arXiv:2404.19226* (2024).

[38] 2024. code-smells-detector. https://github.com/opus-research/organic.

[39] 2024. code quality assessment tool. https://www.designite-tools.com/products-dj.

[40] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente. 2013. Recommending move method refactorings using dependency sets. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 232–241.

[41] Santiago Vidal, Hernan Vazquez, J Andres Diaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. 2015. JSpIRIT: a flexible tool for the analysis of code smells. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, 1–6.

[42] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In *Proceedings of the 40th International Conference on Software Engineering*. 482–482.

[43] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21 (2016), 1143–1191.

[44] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering* 17 (2012), 243–275.

[45] OpenAI. 2023. GPT-3.5-Turbo. https://platform.openai.com/docs/models/gpt-3-5, note = Accessed: 2024.6.

[46] OpenAI. 2023. GPT-4.0-Turbo. https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4, note = Accessed: 2024.6.

[47] 2024. Introducing meta llama 3: The most capa-ble openly available llm to date. https://ai.meta.com/blog/meta-llama-3/..

[48] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[49] Marina Sokolova and Guy Lapalme. 2009. A systematic analysis of performance measures for classification tasks. *Information Processing Management* 45, 4 (2009), 427–437. https://doi.org/10.1016/j.ipm.2009.03.002

[50] 2024. Vulnerability Detection Tool. https://github.com/aquasecurity/trivy.

[51] 2024. Vulnerability Detection Tool. https://github.com/greenbone/openvas-scanner.

[52] 2024. Vulnerability Detection Tool. https://github.com/quay/clair.

[53] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27.

[54] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 935–947.

[55] Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 691–702.

[56] Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. 2024. Vision transformer inspired automated vulnerability repair. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–29.

[57] Bahareh Bafandeh Mayvan, Abbas Rasoolzadegan, and Abbas Javan Jafari. 2020. Bad smell detection using quality metrics and refactoring opportunities. *Journal of Software: Evolution and Process* 32, 8 (2020), e2255.

[58] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering* 36, 1 (2010), 20–36. https://doi.org/10.1109/TSE.2009.50

[59] Yichen Li and Xiaofang Zhang. 2022. Multi-Label Code Smell Detection with Hybrid Model based on Deep Learning.. In *SEKE*. 42–47.

[60] Dongjin Yu, Yihang Xu, Lehui Weng, Jie Chen, Xin Chen, and Quanxin Yang. 2022. Detecting and Refactoring Feature Envy Based on Graph Neural Network. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 458–469.

[61] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius HS Durelli. 2020. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1432–1450.

[62] Wenhao Ma, Yaoxiang Yu, Xiaoming Ruan, and Bo Cai. 2023. Pre-trained Model Based Feature Envy Detection. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 430–440.

[63] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[64] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Mohamed Salah Hamdi. 2015. Improving multi-objective code-smells correction using development history. *Journal of Systems and Software* 105 (2015), 18–39.