Practitioners' expectations on code smell detection

Zexian Zhang[1,2,3], Shuang Yin[4], Wenliu Wei[5], Xiaoxue Ma[6], Jacky Wai Keung[6], Fuyang Li[1,3], and Wenhua Hu[1,3*]
[1]School of Computer Science and Artificial Intelligence, Wuhan University of Technology, Wuhan, China,
{zexianzhang, fyli, whu10}@whut.edu.cn
[2] Sanya Science and Education Innovation Park of Wuhan University of Technology, Sanya, China
[3] Hubei Key Laboratory of Transportation Internet of Things, Wuhan University of Technology, Wuhan, China
[4] School of Computer, Wuhan Qingchuan University, Wuhan, China, shuangyin_cs@163.com
[5] Wuhan Maritime Communication Research Institute, Wuhan, China, wenliu.wei@outlook.com
[6] Department of Computer Science, City University of Hong Kong, Hong Kong, China,
xiaoxuema3-c@my.cityu.edu.hk, jacky.keung@cityu.edu.hk

2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC) | 979-8-3503-7696-8/24/$31.00 ©2024 IEEE | DOI: 10.1109/COMPSAC61105.2024.00175

***Abstract*—Code smell detection can automatically identify code smells in software source code to help developers to improve code maintainability, readability, and overall code quality. Currently, a wide variety of code smell detection techniques/tools are proposed for practical use. However, it is unclear what practitioners expect for code smell detection tools and whether the existing research meets their needs. To fill the gap, we conduct an empirical study. We first interview 10 software development professionals and subsequently survey 310 software practitioners about their practices and expectations of code smell detection tools. In addition, we conduct an extensive literature review of code smell detection papers published in major publications from 2014 to 2024, and compare current research findings with practitioners' expectations. From this comparison, we highlight the direction in which researchers need to work to develop code smell detection techniques that are important to practitioners.**

***Index Terms*—Code Smell Detection, Empirical Study, Practitioners' Expectations**

I. Introduction

As software systems become larger and more complex, software quality attracts the attention of researchers in the field of software engineering [1]–[5]. One important aspect that affects software quality is code smell. Code smell refers to certain symptoms or indications in the source code that suggest there may be underlying problems or potential design flaws [6]–[8]. Code smell detection aims to automatically identify code smells in software source code to ensure code quality, improve software maintainability, and promote good programming practices. Despite the large amount of research in code smell detection, few studies have investigated practitioners' expectations of code smell detection tools. It remains unclear whether practitioners appreciate current code smell detection tools, what factors influence their decision to adopt them, and what is the minimum threshold for their adoption. Practitioner perspectives are important to help guide researchers in creating solutions that satisfy developers.

In this paper, we begin with semi-structured interviews with 10 professionals with an average of 5.1 years of software development experience. Through the interviews, we qualitatively investigate the state of practice of using code smell detection tools, the issues they face, and their expectations of code smell detection tools. We then conduct an exploratory survey of 310 software practitioners from 37 countries across five continents to quantitatively validate the expectations practitioners identified in our interviews. Finally, we conduct a literature review of research papers published in major venues from 2014 to 2024 and compare the techniques proposed in the papers with practitioners' expectations. In particular, we address the following four Research Questions (RQs):

RQ1: What is the state of code smell detection tools, and what are the issues? This research question investigates the current practices of code smell detection tools and the issues faced when using them. Among the participants, 41.3% report that they have used code smell detection tools, with ESlint being the most popular tool. Uncertainty about tool effectiveness and unfamiliarity with code smell detection tools are the main reasons why some participants do not use these tools. In addition, 31% of those who have used existing code smell detection tools express dissatisfaction, highlighting "hard to deal with complex or specific scenarios" and "limitations of different programming languages and projects" as the most prominent issues with these tools.

RQ2: Are code smell detection tools important for practitioners? This research question investigates how practitioners understand the importance of code smell detection tools. 87% of participants agree that these tools are worthwhile and essential to their software development. In addition, 77% of the participants agree that these tools can significantly improve code readability and adherence to coding norms.

RQ3: What are practitioners' expectations on code smell detection tools? This research question investigates practitioners' specific expectations and adoption factors for code smell detection tools. We investigate expectations from several aspects, including preferred detection granularity level, access to service, important factors of the tool, and then we quantitatively investigate general aspects, such as recall, precision, minimum number of code lines to detect, maximum setup time, and detection time. Most participants expect class level/method level detection (83.9%) and offline services (68.4%). They consider the capability of the tools to have high

*Corresponding author: Wenhua Hu.

2836-3795/24/$31.00 ©2024 IEEE
DOI 10.1109/COMPSAC61105.2024.00175

detection accuracy and to improve code quality as important factors for participants to adopt the tools. In addition, for the effectiveness, most participants expect the tools to have a recall and precision at least 70%. For the efficiency, most participants expect the tools to be able to detect at least 7,000 lines of code, with the maximum setup cost of 10 minutes to 1 hour and the detection time of 0.1 - 10 seconds.

RQ4: How close are the current state-of-the-art code smell detection studies to satisfying practitioner needs and demands before adoption? This research question investigates code smell detection research and explores the gap between research and practitioner expectations. We have identified 15 papers on code smell detection techniques. Participants show significant concern regarding Long Parameter Lists and Long Methods, yet few papers address these code smells. Additionally, no papers focus on the functional level, yet participants are concerned about this level. The prevalent use of offline techniques aligns with practitioners' preferences, emphasizing a practical orientation in tool development. In terms of adoption factors, despite the significant focus on detection accuracy and processing speed in current studies, practitioners focus on the capability of tools to improve code quality, the ability of tools to have custom rules and configurations, and the ease of use of tools, which are relatively underexplored in existing research. Furthermore, the findings underscore a critical mismatch between practitioners' expectations and the limited attention devoted to installation, configuration, and learning times in current studies.

Our paper aims to help researchers better consider the needs of practitioners in order to develop code smell detection tools with the potential for widespread adoption and high satisfaction. In summary, the contributions are as follows:

(1) We interview 10 professionals and survey 310 practitioners from 37 countries to understand practitioners' expectations. This includes their perceptions of current code smell detection tools, the importance of code smell detection tools, and the factors that influence their adoption.

(2) We conduct an extensive literature review of code smell detection papers published in major publications in the software engineering field from 2014 to 2024. We then compare the current state-of-the-art studies with practitioner preferences and outline potential implications.

II. Research Methodology

As depicted in Figure 1, our study is divided into three parts: interview, practitioner questionnaire, and literature review. We start with interviewing professionals about their practices in code smell detection, the issues they encounter, and their expectations for code smell detection tools, then conduct a questionnaire survey to validate and extend practitioners' expectations based on the interview. Finally, we conduct a literature review to analyse whether and to what extent current technologies meet practitioners' needs.

A. Interview

1) Protocol: The first author of our paper conducts a series of face-to-face, in-depth, semi-structured interviews to

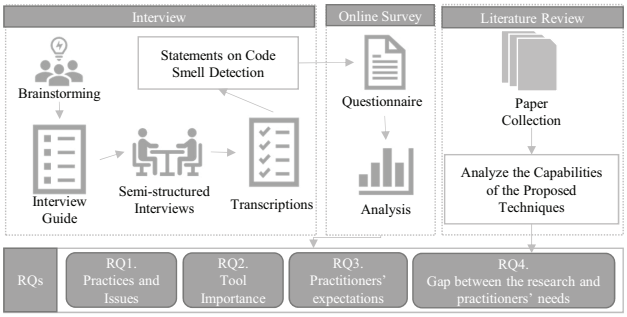


Fig. 1: The overview of the research methodology.

thoroughly explore practitioners' practices, issues and expectations of code smell detection tools. The interview guide is developed through a brainstorming process to facilitate this exploration. Each interview consists of three parts. In the first part, the demographic questions are asked to gather information about the interviewees' backgrounds, such as their job roles, work experience, and commonly used programming languages. In the second part, code smells and the use of their detection tools are introduced to ensure that interviewees understand how these tools work and their potential benefits. The interviewees are then asked if they have any relevant code smell detection tools practices and any relevant issues they have encountered. In the third section, open-ended questions are asked to understand interviewees' expectations regarding code smell detection tools.

2) Interviewees: Professionals in various roles such as developers, testers, and project managers are invited to participate in the interviews through our industry networks. A total of 10 software professionals from 9 global IT companies including Microsoft, Huawei, Tecent, Alibaba, and ByteDance, participate in the interviews, each of which lasts 40-60 minutes. Participants have an average of 5.1 years of professional experience in software development (range: 2 to 12 years, median: 5 years, standard deviation: 3.4 years). At the end of the interviews, we acknowledge the participants and provide them with a brief overview of our upcoming plans.

3) Data Analysis: The first author of our paper transcribes interviews and summarizes participants' views into comment cards. The second author validates and provides suggestions for improvement. After revisions, both authors independently analyze and compile comment cards into potential questionnaire descriptions. The Cohen's Kappa value, indicating general agreement, is 0.7. Discrepancies are resolved through discussions. Both authors jointly review the final set of statements to minimize bias. Eventually, based on the interview results, 9 issues with using code smell detection tools, 8 reasons for non-adoption, 5 conclusions on tool importance, 2 expectations on tool, and 7 adoption factors are identified.

B. Questionnaire Survey

1) Design: The survey consists of different types of questions including single/multiple-choice, Likert scale (Options: Strongly Disagree to Strongly Agree or Not Important to Very Important), and short answer questions. We include the

category "Don't Know" to filter respondents who do not understand or prefer not to answer our questions. The survey consists of five sections:

Demographics: The survey first asks for demographic information about the participants, including country/area of residence, primary job role, experience in years, and programming experience.

Practice and issue of code smell detection tool: In this section we introduce the concepts and examples of code smells and examples of code smell detection tools in order for participants to understand code smells and detection tools, and then survey whether participants have ever used the code smell detection tool. For those who have used we continue to ask which tools are used and the problems encountered. For those who have not used the tool, we ask why.

Tool importance: In this section we ask participants how they perceive the importance of the detection tool: (i) Essential: I will always use this tool to help me detect the presence of smells in my code; (ii) Worthwhile: I will use this tool to help me detect code smells; (iii) Unimportant: I will not use this tool; (iv) Unwise: This tool will negatively impact me or my team; (v) I do not know. We then ask participants to rate statements about the importance of the tool (i.e., efficiency and time saving, improve code readability, improve code performance, adherence to coding norms, and prevent potential bugs).

Practitioners' Expectations: This section surveys practitioners' expectations of these tools, including the preferred level of detection granularity (i.e., class level/method level, file level, functional level, and others) and access to service (i.e., public network, internal network, and offline (e.g., integration into an IDE)).

Tool Adoption: This section asks about the factors that influence their possibility of adopting the tools. We first ask participants to rate the factors that are important for tool adoption (i.e., generalization, fast processing speed, detection accuracy, custom rules and configurations, provide explanations and recommendations, improve code quality, and ease of use). We then quantitatively survey the general aspects, such as recall, precision, minimum number of code lines to detect, maximum setup time, and runtime for a single detection.

At the end of the survey, we ask participants for any other thoughts or concerns they have about code smell detection tools. Respondents may or may not make final comments. Our questionnaire is chosen to provide an English version of the survey on Google Forms and a Chinese version on a popular Chinese survey website[1], since the former one is an international lingua franca, and the latter one is commonly used. Many of our survey respondents would like to be fluent in one of these two languages.

2) Participant Recruitment: In order to obtain an adequate number of professionals working in the industry and open-source practitioners, we follow two steps to invite participants: First, we reach out to professionals within our social network who work in IT companies and ask for their assistance in disseminating our survey. Specifically, we send invitations to practitioners in some well-known IT companies, such as Huawei, encouraging them to share our survey with their colleagues. Second, we collect contributors' public email addresses from GitHub repositories. We focus on collecting developers who concern on code smell, bad smell, or code refactoring projects. We sent an email to 9,000 potential developers with a link to our survey. We receive 243 auto-responses indicating that the recipient is not available.

Eventually, we receive a total of 315 survey responses. In addition, we exclude 2 responses with completion times of less than three minutes and 3 responses from individuals who are not professional software engineers or open source developers, and whose job roles are not related to software development, software testing, algorithm design, architect, or project management. The data reported herein is based on the remaining 310 valid responses. These respondents live in 37 countries on five continents. The two countries with the largest number of respondents are China and the United States. Most participants have more than ten years of professional experience in software development.

3) Data Analysis: We analyse the results according to question type. For multiple-choice and single-choice questions, we report the percentage of each option selected. For Likert scale questions, we plot bar graphs to illustrate the distribution of Likert scores. For short-answer questions, we analyse the results qualitatively. In addition, we exclude "Don't Know" scores because they represent a small fraction of all scores.

C. Literature Review

The papers on code smell detection techniques are usually published in the field of software engineering. Therefore, we review the full research papers published in TOSEM, TSE, FSE, ICSE, and ASE from 2014 to 2024. This is because these conferences and journals are the main publishing venues for the software engineering research community. By reading the titles and abstracts of all papers, we determine whether each paper proposes a new code smell detection technique that can help practitioners find code smells. Finally, we include 15 full research papers, with 3 from TOSEM, 8 from TSE, 1 from FSE, 1 from ICSE, and 2 from ASE. Their code smell detection techniques include the heuristic-based [9]–[18], the machine learning-based [10], [19], and the deep learning-based [20]–[22]. For these included papers, we read their contents and analyse their proposed methods in terms of detection granularity, adoption factors, access to services, and efficiency. We compare the capabilities of the technologies presented in the papers with the adoption thresholds of the practitioners we survey to identify gaps between the current state-of-the-art researches and the needs of practitioners.

III. Result

A. RQ1: Practices and Issues of Code Smell Detection Tool

We investigate practices and issues related to the code smell detection tools. For practitioners who have used the tools, we

[1]https://www.wjx.cn

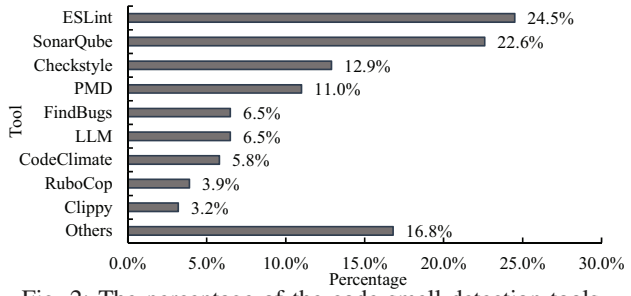


Fig. 2: The percentage of the code smell detection tools.

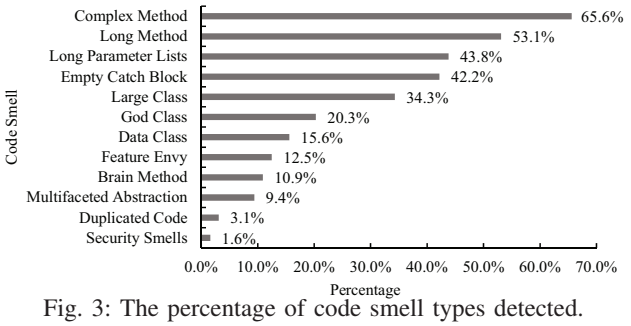


Fig. 3: The percentage of code smell types detected.

investigate which code smell detection tools they have used and the issues they have encountered when using them. For those who have not used these tools, we explore their reasons for not using them.

1) Practices of Code Smell Detection Tools: Among all the responses, 41.3% participants have used code smell detection tools. As shown in Figure 2, the ESlint tool is the most commonly used tool, employed by 24.5% of the participants. SonarQube, Checkstyle, and PMD are the second, third, and fourth most commonly used tools, employed by 22.6%, 12.9%, and 11% of the participants, respectively. Other tools used by a few participants, such as Pyflakes, IntelliJ, JetBrains, and Ruff, account for 16.8%.

Among participants who have used the code smell detection tool, 12 main types of code smells are detected. As shown in Figure 3, Complex Method (i.e., Methods with high cyclomatic complexity [23]) is the most commonly detected, identified by 65.6% of participants. Long Method (i.e., Methods with excessive code logic [24]), Long Parameter Lists (i.e., Methods with excessively long parameter lists [25]), Empty Catch Block (i.e., Empty exception catch blocks [26]), and Large Class (i.e., Classes with many methods and data members [27]) are the second, third, fourth, and fifth most commonly detected types of code smells, identified by 53.1%, 43.8%, 42.2%, and 34.3% of participants, respectively.

Among all responses, 58.7% participants do not use the code smell detection tool. As shown in Figure 5, the main reasons for this include uncertainty about tool effectiveness 57% Agree or Strongly Agree), code smell detection tool unfamiliarity (52%), concerns about installation and learning (34%), and technical knowledge gap in tool usage (30%). As the participants share:

✍ *I haven't used these tools before, but I could argue that they would slow down development in certain scenarios to enforce unrealistic software quality.*

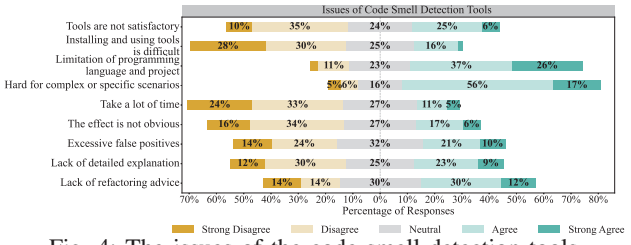


Fig. 4: The issues of the code smell detection tools.

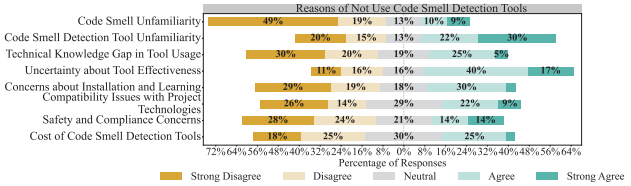


Fig. 5: The reasons for participants not using tools.

✍ *I haven't used code smell detection tools, nor have I really known about the idea of code smell. So far, my best code smell detection tool has been my own nose!*

✍ *Effectiveness of the tools I used was also quite low, let alone the complexity of configuration. You basically spend more time configuring than benefiting from the results.*

2) Issues of Code Smell Detection Tools: Among all the participants who have used the code smell detection tool, we explore the issues they have encountered. As shown in Figure 4, nearly 31% of the participants express dissatisfaction with the utilized tool. The most prominent issue identified by them is "Hard to handle complex or special scenarios", with 73% of participants agreeing or strongly agreeing with this statement. As the participants share:

✍ *Mostly it is not context aware for our codes and the issues detected are usually trivial.*

✍ *Code smells are nebulous issues that hint at higher level design issues that can't generally be fixed locally.*

The second most significant issue identified with existing tools is "Limitations of programming language and project", with 63% agree or strongly agree. As the participants share:

✍ *The languages I use don't have complicated code analysis tools, the ones I specified are basically advanced linters. They're fast but don't go too deep.*

✍ *Tools like CodeClimate and SonarQube are opinionated about PHP and don't really provide good customization options to work in with specific frameworks (e.g., WordPress).*

✍ *Code smell tools can only help reducing bugs if the chosen software design patterns and structure are already suitable for the project. When functions are already filled with "feature creep" or other "code smells", it is likely there are other major flaws in the software structure which can't be helped by these "code smell" tools.*

The third most common issue identified is "Lack of refactoring advice", with 42% of participants agreeing or strongly agreeing. As a participant shares:

✍ *It would be ideal if the tool can suggest what is wrong and how to fix it as well.*

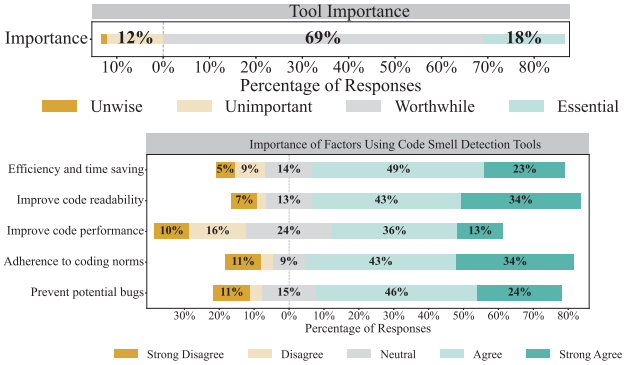


Fig. 6: The tool importance and the importance factors.

Finding 1: 41.3% of the participants have used the code smell detection tool. Their most commonly used tool is ESlint and the most frequently detected code smell is Complex Method. 58.7% of the participants have not used the code smell detection tool, and the top-2 reasons are uncertainty about tool effectiveness and code smell detection tool unfamiliarity.

Finding 2: The top-2 issues encountered by participants using code smell detection tools are hard to handle complex or special scenarios and limitations of programming language and project.

B. RQ2: Tool Importance

Figure 6a outlines the participants' rating of the tool's importance, categorized as "Essential", "Worthwhile", "Unimportant", and "Unwise". Remarkably, 87% of participants consider the code smell detection tool either "Essential" or "Worthwhile". More specifically, approximately 18% of participants rate it as essential and will always use this tool to detect the presence of smells in code.

We further explore the factors that participants consider important in terms of tools. As shown in Figure 6b, the most important factor identified by participants is "Improve code readability", with 77% of participants agreeing or strongly agreeing. As a participant shares:

✎ *Readability is critical to long-term maintenace.*

Another factor that participants identified as important is "Adherence to coding norms" (77%). As participants share:

✎ *Probably the main use for such a tool would be to help junior developers on the team get feedback on their coding style more quickly rather than having to wait for review from senior teammates.*

✎ *I always focus on readability and code styles. Everything in the list is indeed important, but those two items make it less difficult for new developers to onboard a team.*

While most participants acknowledge the importance of the code smell detection tool and recognize its benefits, a minority holds contrasting views. We analyze the reasons behind this dissent based on their responses:

1) On the one hand, the tools cannot replace human review. Some participants believe that human review of code smells is more accurate and trustworthy than tool detection. As participants share:

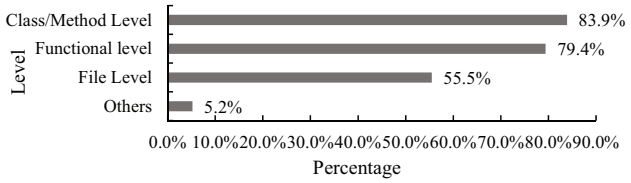


Fig. 7: The expectations on detection granularity level.

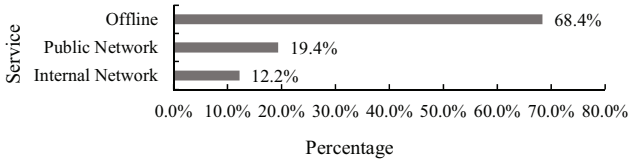


Fig. 8: The expectations on access to service.

✎ *In any case, unless there's explicit errors, design and architectural changes should be reviewed by a human, and style changes probably autoformatted (and avoid discussions about those).*

✎ *In terms of actual code smell, there's no substitute for human review. Any attempts to reduce human review requirements are more likely to cause issues eventually, as there will necessarily be false positives and negatives.*

2) On the other, the tools only focus on the code level and do not support evaluating business logic. As participants share:

✎ *Automated code smell detection tools work very well for keeping conventions and agreements, but detecting performance optimizations would be a challenging exercise - to some degree that may be possible for simple scenarios, but optimizations need to be measured to make sure that they are effective. Optimizations are also often made on a much higher level, like on the level of whole business process, which obviously cannot be analyzed by code smell detection tool.*

✎ *I don't believe such a tool could be efficient catching impactful bugs, as the majority of them are usually in the business logic of the application rather than related to the quirks of programming language.*

Finding 3: 87% of participants agree that the code smell detection tool is important. Most participants (77%) think the tools can improve code readability and adherence to coding norms.

C. RQ3: Practitioners' Expectations

We investigate practitioners' expectations of code detection tools. We explore detection granularity levels (class/method level, file level, and functional level), access to service (public network, internal network, and offline), and adoption factors (generalization, fast processing speed, detection accuracy, custom rules and configurations, provide explanations and recommendations, improve code quality, and ease of use). In addition, we quantitatively investigate practitioners' expectations of other aspects that affect their use of the tool, including recall, precision, the minimum number of lines detected, the

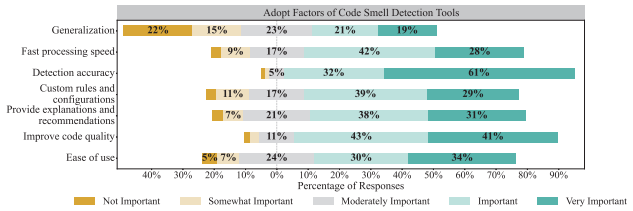


Fig. 9: The adoption factors of code smell detection tool.

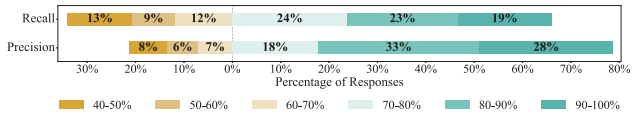


Fig. 10: The expectations on detection effectiveness.

maximum installation time consumption, and the maximum detection time.

1) Expectations on Detection Granularity Level: Code smells are mainly categorised into class and method levels, and most code smell detection techniques also focus primarily on these levels, with a few techniques focusing on the file level. As shown in Figure 7, 83.9%, 79.4%, and 55.5% of participants prefer the class/method level, functional level, and file level, respectively. Other levels of granularity expected by participants, such as project level, package level, application level, and architectural level only account for 5.2%. Notably, the percentages do not add up to 100% because respondents could indicate multiple preferred granularity levels.

2) Expectations on Access to Service: As shown in Figure 8, participants expect to use code smell detection tools in offline environments. In contrast, internal and public networks are notably unpopular. This preference is largely driven by privacy concerns related to the detected code, as participants response: *Privacy is the main concern. Most external tools tend to share code to a server where it runs it's analysis. This poses safety issues esp. When working with copyrighted code as companies will have a policy against using tools that can view their intellectual properties.* and *Tools must be offline to protect intellectual property.*

3) Adoption Factors: Figure 9, 10, and 11 illustrate the factors that influence the possibility of practitioners adopting the tools, including factors, effectiveness, and efficiency.

Factors: As shown in Figure 9, the top-2 most popular factors are: detection accuracy (i.e., whether the expected results of the inspection are effective in helping to detect smells in the code) and improving code quality (i.e., whether the code refactored based on the results of the detection is more readable, easier to understand, and easier to maintain). More than 84% of the participants consider these two criteria to be important or very important. As participants share:

✍ *Inaccuracy leads to needing to suppress a lot.*

✍ *An important thing here is how easy it is to get rid of false positives. I'd rather the tool doesn't detect something than produce noise that has to be ignored every time.*

✍ *If they complain about things which are not really a problem; and then don't really create any positive results, then they will be not be used by developers.*

In addition, roughly 70% of the participants express concern about fast processing speed (i.e., whether the detection tool can quickly detect the given items, saving the cost of manual inspection) and provide explanations and recommendations (i.e., the tool provides contextual explanations of the results of the inspection and suggests possible refactoring).

Furthermore, ease of use (i.e., the ease of installing, configuring, and learning to use the tool) and custom rules and configurations (i.e., the ability to customize detection methods and heuristic rules to meet the needs of different projects) are also important for more than 64% of the participants.

Effectiveness: Figure 10 illustrates the participants' expectations regarding the recall (i.e., the percentage of the actual smell in the file/code is correctly detected) and precision (i.e., the percentage of true smelly codes among detected smelly results) of the code smell detection tool. 66% of participants expect the recall to be at least 70% or higher before considering its use. Furthermore, 19% of participants have even higher requirements, expecting a recall of 90% or more. Similarly, 79% of participants expect the precision to be at least 70% or higher before considering its use, with 28% expecting a precision of 90% or more. Participants' views prioritize reducing false positives or inaccurate reporting rates (i.e., detecting non-smelly codes as smelly):

✍ *Low recall (even below $<$ 20%) is OK, as long as there are few false positives.*

✍ *Few false positives and few false negatives are important. If is flooded with a lot of bad detection, the tool is annoying. If the tool overlooks too much stuff, it is not that useful.*

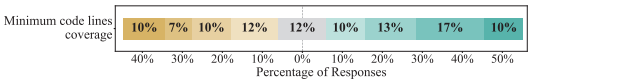
Efficiency: We investigate practitioners' expectations of the efficiency of tools in terms of the minimum lines of code[2] to be detected at a run (as shown in Figure 11a), the maximum installation and configuration time (as shown in Figure 11b), and the maximum runtime to detect a file (as shown in Figure 11c), respectively. We observe that detecting at least 7,000 lines of code or more at a time receives the most attention (40%). In addition, the installation, configuration, and learning time between 10 minutes and 1 hours receive the most attention (63%). Furthermore, the maximum detection time of a file in the range of 0.1 to 10 seconds receives the most attention (77%). As participants share:

✍ *I would like to see some examples in a few minutes and if it impresses me, I am willing to spend hours or even days to perfectly tune it.*

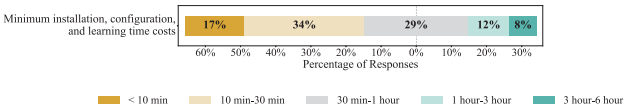
✍ *Installation: $<$ 10 min; basic configuration: $<$ 30 min; basic usage: 1 h; advanced usage: no limit.*

✍ *Single file check: 1-3 s; Many files/full project: $<$ 1 s per file.*

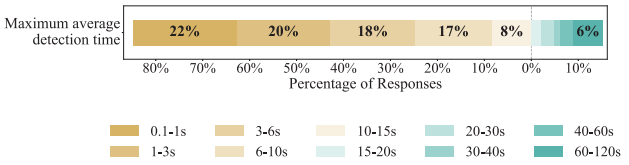
[2]The values of options refer to Kochhar et al. [28], which explored the expectation of the minimum detected lines of fault localization techniques.



(a) The expectations on detecting the minimum code lines.



(b) The expectations on the maximum installation, configuration, and learning time costs.



(c) The expectations on the maximum average detection time.

Fig. 11: The expectations on detection efficiency.

Finding 4: Class level/method level detection (83.9%) and offline services (68.4%) are the most expected.
Finding 5: High detection accuracy and improving code quality are important factors for participants to adopt the tool.
Finding 6: Most participants expect the tools to have at least 70% recall and precision.
Finding 7: Most participants expect the tools to detect at least 7,000 lines of code, have the maximum installation cost of 10 minutes to 1 hour, and detect in 0.1 to 10 seconds.

D. RQ4: Current State-of-the-Art Research

We summarise the total of 15 research papers identified from major publications in software engineering. Table I and II show the capability of the state-of-the-art code smell detection techniques and respondents' capability expectations. The Likert score refers the average of the different importance scores: not important (1), somewhat important (2), moderately important (3), important (4), and very important (5).

1) Granularity Level: (a) 12 papers detect code smells at class/method level. They generally detect code smells based on software metrics extracted from the source code and thresholds determined manually or by classifiers. The main code smells they detect include Feature Envy, God Class, Complex Method, and Brain Method. However the most practitioners are concerned with different types of code smell, such as Long Parameter Lists and Long Method. (b) 3 papers focus on file level detection. They usually take as input a code file or a metric extracted from the file level. For example, Li et al. [18] detect duplicate log code smells by matching log messages and method names. Saavedra et al. [17] detect the presence of security smells by detecting the presence of rules for weak crypts in the file (e.g., isWeakCrypt() "md5", "sha1", "arcfour"). (c) No papers focus on functional level detection, but 79% of respondents expect tools to achieve this level.

TABLE I: The respondents' tool usage preferences and the capabilities of current research. Sat. Rate represents the satisfaction rate of respondents choices.

	Description	Sat. Rate	Papers
Granularity	Class/Method Level	84%	[9]–[11], [13]–[16], [19]–[22], [29]
	File Level	56%	[11], [17], [18]
	Functional Level	79%	-
Access	Offline	68%	[10], [12]–[21]
	Online	32%	[17], [22]
	Description	**Likert Score**	**Papers**
Adoption Factors	Generalization	2.77	-
	Fast processing speed	3.80	[9], [14], [15], [17], [20], [21]
	Detection accuracy	4.48	[12], [13], [17]–[22], [29]
	Custom rules and configurations	3.77	[9], [13], [14], [20]
	Explanations and recommendations	3.83	[10], [12]–[15], [17], [20], [22]
	Improving code quality	4.17	[11], [17], [20]
	Ease of use	3.78	-

2) Access to Service: Most papers present methods that can deploy services in the offline environment, which is in line with the expectations of most practitioners. In contrast, 2 papers propose methods that can get services online [17], [22], since they both mention deploying the model to the server without any model training in local. This involves uploading local files to the server, so it is disliked by most practitioners.

3) Adoption Factors: Among all collected papers, 9 papers focus on the detection accuracy, which is considered important by most practitioners (Likert score = 4.48). Only 3 papers focus on improving code quality capabilities of techniques, but it is noticed by most practitioners (Likert score = 4.17). 8 papers mention that their methods can provide explanations and refactoring recommendations. 6 papers focus on the fast processing speed of techniques. 4 papers mention their techniques can custom heuristic rules and configurations. These three factors are also agreed to some extent by the respondents (Likert score = 3.83, 3.80, and 3.77). However, there are no papers that focus on the ease of use of the tools, while it is focused on by the participants (Likert score = 3.78).

4) Effectiveness: Among the collected papers, 11 papers use the recall as evaluation metric. 6 papers satisfy at least 100% of the participants and 5 papers satisfy at least 81% of the participants. Overall, the code smell detection techniques usually satisfy the participants' expectations in terms of recall. Similarly, in terms of precision, 3 papers satisfy at least 100% of participants, 4 papers satisfy at least 72% of participants, 3 papers satisfy at least 39% of participants, and 1 paper satisfies at least 21% of participants. Overall, the code smell detection techniques usually satisfy the participants' expectations in terms of precision. However, the capability of these techniques to achieve high recall and precision may be related to the use of less complex and non-industrial datasets. For instance, Palomba et al. [29] trained the model to detect code smells

on different versions of the projects with redundant instances and obtained high precision and recall results. Their another study [19] used the dataset that mostly contained only one type of code smell, which does not correspond to real-world scenarios (Nucci et al. [30] point out that a single software instance usually contains multiple code smells). As stated by the participants: *The free and open source code smell tools have detected simple design issues, but do not tend to provide new insights into more complex design issues. While they may help novice developers, especially on newer codebases, they have not been shown to provide new insights to development communities working with mature codebase.*

5) Efficiency: Among the collected papers, no papers focus on the number of lines detected and the installation, configuration, and learning time of the tools. 3 papers explicitly consider detection time. Wang et al. [15] achieves the best detection time of 54 seconds. Liu et al. [21] takes 0.2 minutes to detect using the trained classifier. The average execution time of Saavedra et al.'s [17] method runs on the three datasets averaged 86-1668 seconds and the best detection time can up to 14 seconds. The efficiency of these methods does not satisfy 100% practitioners' expectations (i.e., 0.1-10s).

Finding 8: Participants show significant concern regarding Long Parameter Lists and Long Methods, yet few papers address these code smells. And no papers focus on the functional level, yet participants are concerned about this level.
Finding 9: Most papers implement offline techniques, which is expected by the participants.
Finding 10: Very few papers focus on the capability of tools to improve code quality, the ability of tools to have custom rules and configurations, and the ease of use of tools, but they are considered important by the participants.
Finding 11: Code smell detection techniques typically satisfy participants' expectations in terms of recall and precision. But participants expect the techniques to deal with complex or specific situations and to assist in evaluating business logic.
Finding 12: No papers focus on the number of lines detected and the installation, configuration and learning time of the methods, while very few papers propose methods with detection times that satisfy the expectations of at most 23% of the participants.

IV. Discussion

A. Implications

1) Code smell detection tool: Most participants expect code smell detection tools to be smarter. 73% participants expect the tools to handle complex or specialised scenarios, like some real-life large-scale projects. Some participants also expect the tools to automatically adjust the detection rules to standardise the coding style of the users. As stated by the participants: *It would be cool if machine learning made it possible, after analyzing other projects, to understand which substances are not acceptable and what cannot be configured using standard tools.* In addition, some participants expect the tools can understand context. As participants share: *Mostly it is not context aware for our codes and the issues detected are usually trivial. I would love to see these kinds of tools improve in a way, that allows them to fully understand context and be domain aware as well.* Furthermore, some participants expect the tools to provide explanations and recommendations. As participants share: *It is important to explain suggested changes to also educate the programmer that wrote the 'smelly code'.*

TABLE II: The respondents' capability expectations and the capabilities of current research. Cap. Range represents the capability range.

	Description	Cap. Range	Sat. Rate	Papers
Effectiveness	Recall	90-100%	100%	[13], [18], [20]–[22], [29]
		80-90%	81%	[9], [11], [16], [17], [19]
		70-80%	58%	-
		<70%	34%	-
	Precision	90-100%	100%	[18], [21], [22]
		80-90%	72%	[9], [11], [19], [29]
		70-80%	39%	[13], [16], [17]
		<70%	21%	[20]
Efficiency	Minimum code lines coverage	100K-1M	100%	-
		10K-100K	90%	-
		1K-10K	73%	-
		<1K	27%	-
	Installation, configuration, and learning time costs	<10min	100%	-
		10min-30min	83%	-
		30min-1h	49%	-
		>1 h	20%	-
	Maximum average detection time	0.1-10s	100%	-
		10-60s	23%	[15], [17], [21]
		>60s	6%	-

2) Adoption factors: Most of existing studies focus on detection accuracy, fast processing speed, and provide explanations and refactoring recommendations. However, in addition to these factors being agreed upon by the participants, they are equally concerned about the capability of the detection tools to improve code quality and the ease of use of the tools. As participants shared: *If they complain about things which are not really a problem; and then don't really create any positive results, then they will be not be used by developers. Does the tool make my code better or make my thinking 'institutional'. Effectiveness of the tools I used was also quite low, let alone the complexity of configuration. You basically spend more time configuring than benefiting from the results.*

3) Balancing effectiveness and efficiency: Most of existing studies focus on detection accuracy, evaluating the recall and precision of techniques. The recall and precision of these techniques generally meet participants' expectations (at least 70%). However, their detection times rarely satisfy expectations (0.1-10 seconds). Yet efficiency is important to practitioners. As participants shared: *Speed and resource efficiency is also important, especially for offline use; but it avoids developers*

context-switching away while the tool is running. Efficiency: having a linter as part of your continous integration pipeline streamlines code-review by providing a first line of defense against bad code, reducing my need to explicitly review trivialisms. Therefore, future studies could prioritize the detection time and improve the efficiency of the tools.

B. Threats to Validity

One potential threat in our survey is that some participants may not fully understand the questions, especially if they have never used a code smell detection tool before. To mitigate this, we include clear images illustrating the tool's use to aid participants' comprehension. To ensure that our survey can be easily understood by respondents from China, we translate it into Chinese. In this process, we take great care to minimise bias in the bilingual presentation of surveys and ensure that there is no ambiguity between English and Chinese terms. Additionally, we cannot guarantee that all participants' responses accurately reflect their beliefs, a common limitation recognized by practitioners in many previous studies [31]. Another threat is that our participants may not be representative of typical programmers. Our solution is to survey a wide range of practitioners working in many IT companies. We believe that we have minimised the impact of this threat on our findings.

V. Related work

Many code smell detection methods have been proposed, which can be divided into two categories, i.e., the heuristic based and machine/deep learning based. The heuristic based method [32] employ custom rules to calculate feature values and set thresholds for identifying code smells. For instance, the defined rules for Data Class code smell are (WOC>33%) and (NOPA+NOAM>3 & WMC<31) or (NOPA+NOAM>5 & WMC<47) [32], where WOC denotes the weight of the class, NOPA denotes the number of public attributes, NOAM means the number of non-static public methods, and WMC denotes the number of weighted methods. With the rise of the machine/deep learning algorithms [33]–[36], a large number of researchers propose the machine/deep learning techniques for code smell detection, which leverage labeled data for training classifiers and eliminate the need for rigid rules. The commonly used classification models include support vector machine [30], [37], k-nearest neighbour [30], [37], decision tree [30], [37]. The commonly used deep learning models include long short-term memory networks [38], recurrent neural networks [39], and convolutional neural networks [20]–[22], [39].

Besides research on code smell detection techniques, there has been some work focusing on researching code smell practices. Yamashita et al. [40], Palomba et al. [41], and Taibi et al. [42] explored developers' awareness of, interest in, and their perceived criticality for code smells. They found that developers vary in their awareness, importance assigned, and engagement in code smell elimination and their judgments on design problems vary based on experience and system knowledge. Nardome et al. [43] discovered that video game developers recognize the influence of code smells on game quality and maintainability. Design and game logic-related code smells are considered crucial, while those linked to multiplayer and animation are less emphasized. They also offered suggestions and techniques to tackle code smells, aiding developers in enhancing the quality and performance of their video games. Tufano et al. [44] found that code smells are introduced in diverse ways and at various times, persisting for different durations. Differences in perception and handling may arise from developers not considering them harmful, concerns about refactoring introducing bugs or costs, or dissatisfaction with existing identification and refactoring tools. Bezerra et al. [45] investigated student learning in code smell refactoring, discovering challenges such as understanding source code, encountering new smells post-refactoring, choosing techniques, and lacking reference materials. However, no previous researches investigate primarily practitioners' expectations of code smell detection tools. In this paper, we conduct a large-scale user survey that investigates multiple aspects of practitioners' expectations of code smell detection tools. Additionally, we perform a literature review to highlight disparities between current research and practitioners' expectations, offering insights for future research directions.

VI. Conclusion

In this paper, we interview 10 professionals and survey 310 practitioners to explore their perspectives on code smell detection tools, focusing on their practices, challenges, and expectations. Our findings underscore practitioners' strong emphasis on high detection accuracy and the capability to improve code quality. Additionally, practitioners expect tools to satisfy various requirements related to detection granularity, adoption factors, access to service, effectiveness, and efficiency. Through a literature review, we compare current research capabilities with practitioners' expectations, and identify areas for improvement to satisfy practitioners' needs.

Acknowledgment

This work was in part supported by the Natural Science Foundation of China (62202350), the Start-up Grant from Wuhan University of Technology (104-40120693), and the General Research Fund of the Research Grants Council of Hong Kong and the research funds of the City University of Hong Kong (6000796, 9229109, 9229098, 9220103, 9229029).

References

[1] X. Yu, L. Liu, L. Zhu, J. W. Keung, Z. Wang, and F. Li, "A multi-objective effort-aware defect prediction approach based on nsga-ii," *Applied Soft Computing*, vol. 149, p. 110941, 2023.

[2] L. Gong, H. Zhang, J. Zhang, M. Wei, and Z. Huang, "A comprehensive investigation of the impact of class overlap on software defect prediction," *IEEE transactions on software engineering*, vol. 49, no. 4, pp. 2440–2458, 2022.

[3] X. Yu, J. Liu, Z. Yang, X. Jia, Q. Ling, and S. Ye, "Learning from imbalanced data for predicting the number of software defects," in *2017 IEEE 28th international symposium on software reliability engineering*. IEEE, 2017, pp. 78–89.

[4] X. Yu, J. Liu, J. W. Keung, Q. Li, K. E. Bennin, Z. Xu, J. Wang, and X. Cui, "Improving ranking-oriented defect prediction using a cost-sensitive ranking svm," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 139–153, 2019.

[5] L. Gong, G. K. Rajbahadur, A. E. Hassan, and S. Jiang, "Revisiting the impact of dependency network metrics on software defect prediction," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 5030–5049, 2021.

[6] W. Hu, L. Liu, P. Yang, K. Zou, J. Li, G. Lin, and J. Xiang, "Revisiting" code smell severity classification using machine learning techniques"," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference*. IEEE, 2023, pp. 840–849.

[7] F. Li, K. Zou, J. W. Keung, X. Yu, S. Feng, and Y. Xiao, "On the relative value of imbalanced learning for code smell detection," *Software: Practice and Experience*, 2023.

[8] L. Liu, G. Lin, L. Zhu, Z. Yang, P. Song, X. Wang, and W. Hu, "Revisiting code smell severity prioritization using learning to rank techniques," *Expert Systems with Applications*, p. 123483, 2024.

[9] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 1, pp. 1–44, 2014.

[10] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 4, pp. 1–39, 2014.

[11] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[12] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, 2013.

[13] H. Liu, Q. Liu, Z. Niu, and Y. Liu, "Dynamic and automatic feedback-based threshold adaptation for code smell detection," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 544–558, 2015.

[14] S. Vidal, I. Berra, S. Zulliani, C. Marcos, and J. A. D. Pace, "Assessing the refactoring of brain methods," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 1, pp. 1–43, 2018.

[15] Y. Wang, H. Yu, Z. Zhu, W. Zhang, and Y. Zhao, "Automatic software refactoring via weighted clustering in method-level networks," *IEEE Transactions on Software Engineering*, vol. 44, no. 3, pp. 202–236, 2017.

[16] F. Palomba, D. A. Tamburri, F. A. Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, "Beyond technical aspects: How do community smells influence the intensity of code smells?" *IEEE transactions on software engineering*, vol. 47, no. 1, pp. 108–129, 2018.

[17] N. Saavedra and J. F. Ferreira, "Glitch: Automated polyglot security smell detection in infrastructure as code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[18] Z. Li, T.-H. Chen, J. Yang, and W. Shang, "Dlfinder: characterizing and detecting duplicate logging code smells," in *2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE, 2019, pp. 152–163.

[19] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," *IEEE Transactions on Software Engineering*, vol. 45, no. 2, pp. 194–218, 2017.

[20] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 385–396.

[21] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, "Deep learning based code smell detection," *IEEE transactions on Software Engineering*, vol. 47, no. 9, pp. 1811–1837, 2019.

[22] B. Liu, H. Liu, G. Li, N. Niu, Z. Xu, Y. Wang, Y. Xia, Y. Zhang, and Y. Jiang, "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*, 2023, pp. 908–920.

[23] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," *Journal of Systems and Software*, vol. 176, p. 110936, 2021.

[24] M. Hadj-Kacem and N. Bouassida, "A hybrid approach to detect code smells using deep learning." in *International Conference on Evaluation of Novel Approaches to Software Engineering*, 2018, pp. 137–146.

[25] Y. Li and X. Zhang, "Multi-label code smell detection with hybrid model based on deep learning." in *International Conference on Software Engineering and Knowledge Engineering*, 2022, pp. 42–47.

[26] W. Xu and X. Zhang, "Multi-granularity code smell detection using deep learning method based on abstract syntax tree," in *Proc. 33rd Int. Conf. Software Engineering and Knowledge Engineering*, 2021, pp. 503–509.

[27] D. K. Kim, "Finding bad code smells with neural network models," *International Journal of Electrical and Computer Engineering*, vol. 7, no. 6, p. 3613, 2017.

[28] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 165–176.

[29] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2014.

[30] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *2018 ieee 25th international conference on software analysis, evolution and reengineering*. IEEE, 2018, pp. 612–621.

[31] C. Wang, J. Hu, C. Gao, Y. Jin, T. Xie, H. Huang, Z. Lei, and Y. Deng, "Practitioners' expectations on code completion," *arXiv preprint arXiv:2301.03846*, 2023.

[32] S. Charalampidou, A. Ampatzoglou, and P. Avgeriou, "Size and cohesion metrics as indicators of the long method bad smell: An empirical study," in *The 11th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM Press, 2015.

[33] C. He, J. Wu, and Q. Zhang, "Proximity-aware research leadership recommendation in research collaboration via deep neural networks," *Journal of the Association for Information Science and Technology*, vol. 73, no. 1, pp. 70–89, 2022.

[34] C. He, F. Liu, K. Dong, J. Wu, and Q. Zhang, "Research on the formation mechanism of research leadership relations: An exponential random graph model analysis approach," *Journal of Informetrics*, vol. 17, no. 2, p. 101401, 2023.

[35] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang, "A multi-modal transformer-based code summarization approach for smart contracts," in *2021 IEEE/ACM 29th International Conference on Program Comprehension*. IEEE, 2021, pp. 1–12.

[36] Z. Yang, J. W. Keung, X. Yu, Y. Xiao, Z. Jin, and J. Zhang, "On the significance of category prediction for code-comment synchronization," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–41, 2023.

[37] S. Jain and A. Saha, "Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection," *Science of Computer Programming*, vol. 212, p. 102713, 2021.

[38] X. Guo, C. Shi, and H. Jiang, "Deep semantic-based feature envy identification," in *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, 2019, pp. 1–6.

[39] A. Hamdy and M. Tazy, "Deep hybrid features for code smells detection," *Journal of Theoretical and Applied Information Technology*, vol. 98, no. 14, pp. 2684–2696, 2020.

[40] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *2013 20th working conference on reverse engineering*. IEEE, 2013, pp. 242–251.

[41] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 101–110.

[42] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Information and Software Technology*, vol. 92, pp. 223–235, 2017.

[43] V. Nardone, B. Muse, M. Abidi, F. Khomh, and M. Di Penta, "Video game bad smells: What they are and how developers perceive them," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–35, 2023.

[44] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[45] C. Bezerra, H. Damasceno, and J. Teixeira, "Perceptions and difficulties of software engineering students in code smells refactoring," in *Anais do X Workshop de Visualização, Evolução e Manutenção de Software*. SBC, 2022, pp. 41–45.