Detecting Code Smells using ChatGPT: Initial Insights

Luciana Lourdes Silva
luciana.lourdes.silva@ifmg.edu.br
Federal Institute of Minas Gerais
Brazil

Jânio Silva
janio.silva@ifmg.edu.br
Federal Institute of Minas Gerais
Brazil

João Eduardo Montandon
jemaf@ufmg.br
Federal University of Minas Gerais
Brazil

Marcus Andrade
marcus-v@outlook.com
Federal Institute of Minas Gerais
Brazil

Marco Tulio Valente
mtov@dcc.ufmg.br
Federal University of Minas Gerais
Brazil

Abstract

This paper describes first insights on the effectiveness of ChatGPT to detect bad smells in Java projects. We use a large dataset comprising four code smells (Blob, Data Class, Feature Envy, and Long Method) classified into three severity levels. We run two different prompts to assess ChatGPT's proficiency: i) a generic prompt to verify whether the model can detect smells and ii) a prompt specifying the smells classified in the dataset. We apply evaluation metrics in terms of precision, recall, and F-measure to quantify the ChatGPT abilities' in identifying the aforementioned smells. Our preliminary results the odds of ChatGPT providing a correct outcome with a specific are 2.54 times higher compared to a generic prompt. Moreover, ChatGPT is more effective at detecting smells with critical severity (F-measure reaching 0.52) than smells with minor severity (F-measure equals to 0.43). To conclude, we discuss the implications of our results and highlight future work in leveraging large language models for detecting code smells.

CCS Concepts

• **Computing methodologies** → *Natural language processing*.

Keywords

Large Language Models; ChatGPT; Code Smells; Bad Smells; Refactoring Opportunities; Software Quality

ACM Reference Format:
Luciana Lourdes Silva, Jânio Silva, João Eduardo Montandon, Marcus Andrade, and Marco Tulio Valente. 2024. Detecting Code Smells using ChatGPT: Initial Insights. In *Proceedings of (ESEIW 2024 ESEM Emerging Results, Vision and Reflection Papers Tracks).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Large Language Models (LLMs) are being used to automate a variety of Software Engineering activities [1, 3, 5, 8–10, 12, 15, 16]. The primary one is code generation—specifically, code completion—using tools like GitHub Copilot [2], which has become an integral part of developers' workflow. For instance, a recent GitHub study based on telemetry data from nearly 1 million users shows that developers tend to accept 30% of the code suggestions provided by the Copilot tool [17]. The study extrapolates that this acceptance rate is equivalent to adding 15 million developers to the global workforce of software development professionals.

However, language models like GPT are also being used to support other Software Engineering activities, besides generating and completing code automatically. For example, they are being used for identifying and repairing bugs [8, 9], generate unit tests [10], increase the coverage of existing unit tests [1], generate user stories [15], review pull requests [12, 13, 16], etc.

However, to our knowledge, we still **lack works that explore the use of LLMs in a key software improvement activity: the identification of code (or bad) smells**. These structures, as proposed by Beck and Fowler in the 1990s, are indicators of low-quality code and, therefore, may potentially represent an opportunity to perform refactorings [11]. Thus, the identification of code smells could be a new problem in which the application of Large Language Models can contribute to the work of Software Engineers. The main advantage would be the adoption of the same tool for both code generation and for detecting design problems.

Therefore, in this paper, we explore the use of ChatGPT to identify smells in the code of Java projects. To this purpose, we leverage an existing dataset of code smells, called MLCQ [7], which contains 14,739 instances of four well-known code smells: **Blob** (when one class becomes excessively large and complex, often assuming multiple responsibilities that should ideally be handled by separate components), **Data Class** (a class that primarily stores data without implementing any significant functionality or behavior), **Feature Envy** (when a method in one class excessively accesses the data or methods of another class rather than relying on its own class data and methods), and **Long Method** (a method or function that is excessively lengthy and complex).

In the study, we used ChatGPT version 3.5-Turbo and two prompts: first, a prompt in which we asked the model in open and generic terms to identify smells in a given code block (class or method); after that, we also evaluated a prompt in which we explicitly listed the names of the four smells that we intended ChatGPT to identify. Particularly, our first goal was to answer four research questions:

RQ.1: Can ChatGPT identify code smells using a generic prompt?

ESEIW 2024 ESEM Emerging Results, Vision and Reflection Papers Tracks, Oct 20–25, 2024, Barcelona, Spain

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

RQ.2: Can ChatGPT identify code smells with a detailed prompt?

RQ.3: How does the effectiveness of ChatGPT differ between generic and detailed prompts?

RQ.4: How does the severity level of smells influence ChatGPT's performance?

Our current findings show that **a specific prompt is more efficient in detecting smells than a generic one**. For example, for Data Class, the F-measure scores were 0.59 (specific prompt) and 0.11 (generic prompt). Another important and interesting finding was that the **models perform better in identifying critical smells compared to less severe ones**. For instance, in the case of critical Data Classes, the F-measure achieved with ChatGPT is 0.67, against 0.53 for the instances of this smell labeled as having minor severity.

This initial work provided us with various insights on how to improve our current results. For example, we plan to enrich the prompts with contextual information, such as the dependencies of the code under analysis. In a second moment, we also plan to train and fine tune ChatGPT on code-specific datasets containing code smells instances.

The remainder of this paper is organized as follows. In Section 2, we present the design of the study, including the datasets and prompts used in the research. Section 3 presents and motivates the proposed research questions and Section 4 presents the results of each one. Then, in Section 5 we document threats to validity and in Section 6 we comment on related work. Finally, Section 7 concludes with lessons learned and insights for future work.

Replication Package: Our dataset includes all the information used to conduct this study. It is publicly available at GitHub: https://github.com/soft-eng-datasets/code-smells-chatGPT.

2 Study Design

2.1 Dataset

In this work, we select MLCQ [7], an industry code smell dataset. The dataset consists of four code smells (Blob, Data Class, Feature Envy, and Long Method) classified into four-level severity scales (None, Minor, Major, and Critical). The original dataset has 14,739 instances. However, 11,448 instances have a severity of None (77.7%). Thus, for this research, we decided to discard these instances and focus on the remaining 3,291 instances classified as Minor, Major, or Critical. Table 1 presents the statistics of our curated dataset. We can see that it includes 974 Blobs (BL), 1,057 Data Classes (DC), 454 instances of Feature Envy (FE), and 806 Long Methods (LM).

Table 1: Dataset statistics (BL: Blob; DC: Data Class; FE: Feature Envy; LM: Long Method)

Severity	BL	DC	FE	LM	Total
Minor	535	510	288	454	1,787
Major	312	401	142	274	1,129
Critical	127	146	24	78	375
Total	974	1,057	454	806	3,291

2.2 ChatGPT Prompts

For each smell, we submitted the prompts presented in Figure 1 and Figure 2 to ChatGPT. The difference between both prompts lies in the details of information provided to ChatGPT. As we can see in Figure 1, the first prompt is more generic, as we do not inform which smells we intend to detect. By contrast, and as we can see in Figure 2, the second prompt lists the smells that we plan to detect using ChatGPT, i.e., it explicitly mentions the four smells in our dataset. Both prompts end with the code to be evaluated by ChatGPT. We opted to follow this dual prompt strategy to assess how the lack of context affects the results produced by ChatGPT.

Prompt #1

I need to check if the Java code below contains code smells (aka bad smells). Could you please identify which smells occur in the following code? However, do not describe the smells, just list them.

Please start your answer with "YES I found bad smells" when you find any bad smell. Otherwise, start your answer with "NO, I did not find any bad smell".

When you start to list the detected bad smells, always put in your answer "the bad smells are:" amongst the text your answer and always separate it in this format: 1. Long method, 2. Feature envy

[Java source code with the smell]

Figure 1: Prompt #1 (generic prompt)

Prompt #2

The list below presents common code smells (aka bad smells). I need to check if the Java code provided at the end of the input contains at least one of them.
* Blob
* Data Class
* Feature Envy
* Long Method
Could you please identify which smells occur in the following code? However, do not describe the smells, just list them.

Please start your answer with "YES I found bad smells" when you find any bad smell. Otherwise, start your answer with "NO, I did not find any bad smell".

When you start to list the detected bad smells, always put in your answer "the bad smells are:" amongst the text your answer and always separate it in this format: 1. Long method, 2. Feature envy ...

[Java source code with the smell]

Figure 2: Prompt #2 (mentions the smells of interest)

However, we failed to retrieve the code of 472 smells (14%), because their GitHub repositories are not available anymore. We have to access the repositories to append the source code where the smell occurs in both prompts. Thus, we discarded such smells. After that, we used a Python script to submit the remaining prompts to ChatGPT version 3.5-Turbo. However, 52 prompts (0.16%) exceeded the maximal number of tokens expected by GPT-3.5. At the time of our experiment, this limit was 16,385 tokens. Then, we also removed such prompts.

2.3 Processing ChatGPT Answers

The ChatGPT output was parsed to retrieve the identified smells. As can be checked in Figures 1 and 2, we explicitly defined in both prompts the output format. Consequently, when we find a YES in a ChatGPT response, we attempt to automatically parse the response to get the smells after the string "*The bad smells are:*". Next, we shown an example of answer generated by ChatGPT that we could successfully parse using the mentioned strategy.

```
YES I found bad smells The bad smells are:
1. Long method 2. Feature envy
```

However, in 27 cases ChatGPT did not produce answers following the specified output format. Thus, such answers were manually reviewed by the paper authors to figure out the smells that were in fact identified by the model. To illustrate, we show an example of an incorrectly formatted answer. As we can see, the JSON output includes several fields. In this particular case, we only considered valid the identification of a Long Method, since it is the one preceded by the sentence: "the bad smells are:".

```
{
"NO, I did not find any bad smell" : "Blob" ,
"Data Class" : [],
"Feature Envy" : "YES I found bad smells",
"the bad smells are" : [ "Long Method" ]
}
```

2.4 Evaluation Metrics

To assess ChatGPT performance when used to detect code smells, we apply the following metrics that are widely used in Information Retrieval and Machine Learning: (1) precision, provides insights into how many of the smells detected by ChatGPT are true positives; (2) recall, measures how many of the actual smells in the dataset were identified by ChatGPT; and (3) F-measure, which is the harmonic mean of precision and recall, i.e., it balances the results of precision and recall in a single metric.

3 Research Questions

The goal of this emerging paper is to provide first results on the effectiveness of ChatGPT in detecting code smells. Hence, we elaborated three research questions to be answered, described below.

RQ.1: Can ChatGPT identify code smells using a generic prompt? In this first RQ, we submit each smell occurrence to ChatGPT using the prompt described in Figure 1. Our goal is to analyze the performance of the model at detecting smells in scenarios lacking proper context. In other words, we intend to check whether ChatGPT is able to detect smells when they are not explicitly mentioned in the prompt.

RQ.2: Can ChatGPT identify code smells with a detailed prompt? To answer this second RQ, we used the prompt described in Figure 2. As mentioned previously, this prompt lists the code smells we are interested in, i.e., Blob, Data Class, Feature Envy, and Long Method. Our goal is to evaluate the performance of ChatGPT with prompts containing more information about the task at hand, i.e., the code smells we plan to detect.

RQ.3: How does the effectiveness of ChatGPT differ between generic and detailed prompts? This research question investigates the ChatGPT comparative effectiveness in detecting code smells in Java code. We statistically compare the performance of ChatGPT between Prompt #1 (a generic prompt) and Prompt #2 (a detailed prompt) to understand how the level of detail in the prompt influences ChatGPT's ability to identify code smells accurately.

Particularly, we applied McNemar's test to evaluate the effectiveness of different prompts in improving ChatGPT's performance. McNemar's test is a non-parametric statistical test used to determine whether there is a significant difference between paired nominal data [14]. Specifically, the test evaluates whether the discordances between two cases match. This assessment is technically referred to as the homogeneity of the contingency table. The test is commonly applied in medicine to compare the effect of a treatment against a control. The *null hypothesis (H0)* means that the two cases exhibit discordance to an equal extent. The rejection of the *null hypothesis* indicates evidence suggesting that the cases disagree differently.

In our context, the paired data represents the correctness of ChatGPT's responses to Prompt #1 and Prompt #2. With the analysis of these paired proportions, we assess whether a more detailed prompt (i.e., Prompt #2) leads to a statistically significant improvement in ChatGPT's capability compared to Prompt #1. For this, we compute:

- *Absolute Risk Difference* - calculates the difference in proportions between two groups. In McNemar's test, this difference is applied over the ratio of the success event between the two paired data. For example, consider two groups trying to lose weight; if 70% of one group (who exercise often) loses weight compared to 40% of another group (who do not), the Absolute Risk Difference is 30%. This means that adopting the first group's treatment increases the likelihood of losing weight by 30%. In our context, this value indicates how much more one prompt can return a correct result over another.
- *Odds Ratio* - represents the ratio of the odds of an event occurring in one group to the odds of it occurring in another group. An odds ratio less than 1 indicates that the odds of an event occurring in the first group are lower compared to the second group, while a value greater than 1 suggests the opposite. In our context, this value indicates by which extent the odds of ChatGPT providing a correct outcome with one prompt is higher when compared to another.

RQ.4: How does the severity level of smells influence ChatGPT's performance? While RQ.1 and RQ.2 investigate the ChatGPT's performance according to the provided prompt, this RQ analyzes how the smells' severity level impact on ChatGPT's performance. For each prompt, we grouped the detected smells based on their severity level described in Table 6 to compute precision, recall and F-measure.

4 Results

RQ.1: Can ChatGPT identify code smells using a generic prompt?

Table 2 shows the performance of ChatGPT at detecting code smells with Prompt #1, i.e., a generic prompt. We achieved the highest precision for Data Class (0.52), while Feature Envy presented the lowest score (0.14). The highest recall was obtained for Long Method

(0.92), while the Data Class presented the lowest one (0.06). In terms of F-measure, the results vary between 0.11 (Data Class) to 0.41 (Long Method). Finally, we were not able to calculate the performance metrics for Blob since ChatGPT did not identify any Blob occurrence correctly, i.e., the model misclassified all samples labeled as Blob.

Table 2: Results for Prompt #1.

Code Smell	Precision	Recall	F-measure
Blob	-	-	-
Data Class	0.52	0.06	0.11
Feature Envy	0.14	0.75	0.23
Long Method	0.27	0.92	0.41

Summary: The performance of ChatGPT was not good for any of the smells. The best results were obtained in identifying Long Method (F-measure = 0.41), while the worst were for Blobs, where ChatGPT did not return any result, neither positive nor negative.

RQ.2: Can ChatGPT identify code smells with a detailed prompt?

Table 3 presents the performance of ChatGPT at detecting the code smells with Prompt #2, i.e., a detailed prompt. We also indicate in the table, within parentheses, the difference compared with the results from Prompt #1. We would like to highlight two points regarding the results presented in Table 3. First, we were able to obtain results for Blob, although with poor performance (F-measure = 0.19). Second, for the other smells, the results presented a significant improvement for Data Class (an increase of 0.48 in F-measure), but the same was not observed for Feature Envy (decrease of 0.02 in F-measure) and Long Method (an increase of just 0.05).

Table 3: Results for Prompt #2.

Code Smell	Precision	Recall	F-measure
Blob	0.31	0.14	0.19
Data Class	0.53 (+0.01)	0.67 (+0.61)	0.59 (+0.48)
Feature Envy	0.28 (+0.14)	0.16 (-0.59)	0.21 (-0.02)
Long Method	0.31 (+0.04)	0.90 (-0.02)	0.46 (+0.05)

Summary: When using a detailed prompt, the main improvement occurred for Data Class, which now has an F-measure of 0.59. However, for all other smells, the F-measure remained below 0.50.

RQ.3: How does the effectiveness of ChatGPT differ between generic and detailed prompts?

Table 4 shows the contingency table to calculate McNemar's test for Prompt #1 and Prompt #2 in terms of outcome correctness. Each cell in the table represents the number of responses by ChatGPT, categorized by correctness for both prompts. The rows indicate the correctness to Prompt #1, while the columns represent its correctness to Prompt #2. For instance, the cell at the intersection of "Correct" row and "Correct" column shows the number of instances where ChatGPT provided *correct* responses to both Prompts. Conversely, the cell at the intersection of row and column "Incorrect" shows the number of instances where ChatGPT provided *incorrect* responses to both prompts.

Table 4: Contingency table comparing the correctness of ChatGPT's answers for Prompt #1 and Prompt #2.

		Prompt #2		Total
		Correct	Incorrect	
Prompt #1	Correct	647	294	941
	Incorrect	748	1,078	1,826
	Total	1,395	1,372	2,767

ChatGPT failed to detect smells correctly in 1,078 instances for both prompts, while it correctly identified 647 instances matching with the dataset. ChatGPT accurately classified 748 instances using Prompt #2 but failed to do so with Prompt #1. Conversely, the model correctly identified 294 instances with Prompt #1 but misclassified them with Prompt #2. That is, the model accurately classified 941 samples using Prompt #1 against 1,395 samples with Prompt #2.

We used the NCSS Statistical Tool[1] to run McNemar's test on the contigency table. The test showed a significant statistical difference between the outcomes of both prompts, with Prompt #2 outperforming Prompt #1. With an Odds Ratio of 2.54, ChatGPT is 2.54 times more likely to provide a correct outcome with Prompt #2. The Absolute Risk Difference was 0.1641, indicating that Prompt #2 has a 16% higher chance of yielding correct outcomes overall.

Summary: McNemar's test result strongly confirms a statistically difference between ChatGPT's Prompt #2 and Prompt #1 for detecting bad smells. Specifically, the odds of ChatGPT providing a correct outcome with Prompt #2 are 2.54 times higher compared to Prompt #1.

RQ.4: How does the severity level of smells influence ChatGPT's performance?

Table 5 presents precision, recall, and F-measure values for Prompt #1 and Prompt #2 across three severity levels: Minor, Major, and Critical. Despite the metrics not revealing impressive outcomes, ChatGPT's performance with Prompt #2 outperforms Prompt #1 across all severity levels for all metrics. Interestingly, the highest precision, recall, and F-measure were measured on the Critical severity level. The second best results were achieved for Major severity instances. For example, at the Critical severity level, Prompt #2 demonstrates higher precision (0.48) and recall (0.58) compared to Prompt #1 (precision = 0.25, recall = 0.29). These metrics suggest that Prompt #2 (F1-measure = 0.52) is more effective than Prompt #1 (F1-measure = 0.27) at identifying critical smells in the codebase. Similarly, at the Major severity level, Prompt #2 presents higher precision (0.45), recall (0.53), and F-measure (0.49) than Prompt #1 (precision = 0.30, recall = 0.36, and F-measure = 0.33). This finding suggests that detecting Critical and Major smells in the code may be easier than identify Minor ones.

Summary: ChatGPT is more effective at detecting smells with critical severity (F-measure reaching 0.52) than smells with minor severity (F-measure equals to 0.43 when using Prompt #2).

[1] https://www.ncss.com/software/ncss/

Table 5: Results by Prompt and Severity Levels.

Metric	Prompt	Minor	Major	Critical
Precision	#1	0.30	0.30	0.25
	#2	0.41	0.45	0.48
Recall	#1	0.36	0.36	0.29
	#2	0.47	0.53	0.58
F-measure	#1	0.33	0.33	0.27
	#2	0.43	0.49	0.52

Table 6 reports the F-measure results considering the Severity and type of smell. Thus, it allows us to analyze whether ChatGPT excels at detecting specific Code Smells at particular Severity Levels.

Table 6: F-measure by Severity Level for each Smell.

Smell	Severity	Prompt #1	Prompt #2
Blob	Critical	-	0.21
	Major	-	0.18
	Minor	-	0.20
Data Class	Critical	0.10	0.67
	Major	0.16	0.64
	Minor	0.08	0.53
Feature Envy	Critical	0.15	0.29
	Major	0.22	0.16
	Minor	0.25	0.22
Long Method	Critical	0.39	0.47
	Major	0.42	0.47
	Minor	0.41	0.45

Blob. As previously reported in the analysis of RQ.1 (Table 2), we could not compute the metrics for Blob using Prompt #1. When using Prompt #2, the model performance to detect Blob is not significant at any severity level. The highest F-measure (0.21) occurs when used to detect critical smells.

Data Class. In this case, there is a substantial improvement when it was submitted along with Prompt #2. The model achieved an F-measure of 0.67 for Critical and 0.64 for Major smells; more than 50 points higher compared with Prompt #1. Both scenarios are also the highest ones overall, i.e., when considering other smells.

Feature Envy. The performance at detecting this smell was not good at any severity. The highest score was achieved when detecting Critical smells with Prompt #2 (0.29).

Long Method. Prompt #1 obtained its best results when detecting this smell in all Severity Levels. It scored 0.39 for Critical, 0.42 for Major, and 0.41 for Minor smells. These results are 24, 20, and 16 points higher when compared with Feature Envy, the second-best scenario at the same prompt. By contrast, the performance was mostly equivalent when analyzing against Prompt #2, i.e., the difference was below 10 points for all severity levels.

Summary: When analyzing each smell separately, ChatGPT achieved its best performance when detecting Critical and Major Data Class occurrences. Interestingly, Long Method reported equivalent results between Prompts #1 and #2.

5 Threats to Validity

Selected Language. The performance of ChatGPT may vary according to the programming language used in the prompt. This study focused on a very popular programming Language (Java), yet the results can not be generalized to smells in other languages.

Selected Prompts. As a language model, the outcomes of ChatGPT are directly influenced by the quality of the prompt provided to it. In this initial work, we investigated two prompt strategies varying according to their level of detail. As we will discuss in the next section, in future work we intend to evaluate other prompts based on the particularities of each code smell.

Selected Smells. We evaluated ChatGPT against four types of traditional code smells: Blob, Data Class, Feature Envy, and Long Method. Despite the restricted number of categories, all smells were extracted from a curated dataset list [7].

6 Related Work

Large Language Models are Deep Learning-based models that were designed to deal with Natural Language Processing tasks [5, 9]. In the context of Automated Software Engineering, many researchers are investigating how these models help software engineers during their development tasks, such as code completion [2], test case generation [1, 10], code translation [9], program repair [8], etc. With respect to their prompt methods, we observe that most studies use either Pre-Trained models or Prompt-Engineering techniques [6]. While the former involves training a general-purpose models with data specific for the task at hand, the latter consists of refining the prompt to make the model producing high-quality answers.

Pre-Trained Models. Tufano et al. [12] investigated the performance of Pre-Trained models based on T5 in automating the Code Review process. The authors tested their models in a large-scale dataset containing 168K code reviews and observed the Language Model outperforms Deep Learning-based models previously implemented by the same authors [13]. Liu et al. [5] developed a pre-trained language model based on multi task learning optimized for code completion tasks. The authors trained this model with hybrid objective functions which incorporated both code understanding and code generation tasks. This model was tested against two real-world datasets containing programs implemented in Java and TypeScript. Overall, the proposed model outperformed state-of-the-art tools. Zhang et al. [16] analyzed the performance of well-known language models in summarizing titles from pull requests performed on GitHub projects. Overall, the authors obtained better results when relying on a pre-trained version of the BART model.

Prompt Engineering. Siddiq et al. [10] investigated the effectiveness of three language models in generating unit tests. The authors developed seven prompt heuristics that increased the compilation rate, and reduced the smells of the tests generated by the models. Similarly, Alshahwan et al. [1] implemented a Language Model-based tool to improve already existing human-written tests. The tool was sucessfully deployed at two Meta's products, where developers accepted 73% of the recommendations provided by the tool. Imai [4] conducted a preliminary study evaluating the performance of GitHub copilot as a pair-programming tool. The author setup three

controlled experiments comparing the source code produced with and without Copilot. The obtained results suggested that Copilot generates more lines of code, but with lower quality, overall. White et al. [15] leveraged prompt designed technique for general software engineering tasks, including requirements elicitation, deployment, and testing. Shin et al. [9] investigated the effectiveness of three distinct prompt engineering techniques to solve code generation, code summarization, and code translation tasks. The results have shown that conversational prompts produced better results when compared with the other ones.

This work also rely on Prompt Engineering techniques to assist developers in software quality issues. Specifically, we study the effectiveness of GPT-based models at detecting code smells.

7 Discussion and Future Work

7.1 Potential Factors for Varied Performance

We highlight the factors that may lead to the model's variadic performance when detecting the code smells, and what can be done to mitigate them.

Complexity of Blobs and Feature Envies. As indicated by the F-measure presented in Table 6, the performance of ChatGPT in detecting Blob and Feature Envy smells was notably poor across all severity levels and prompts. This could be due to the inherent complexity and context-dependent characteristics of these smells. We intend to explore methods to incorporate structural data about the code to address the lack of contextual information.

Good Performance with Data Class smell. ChatGPT presented good performance in detecting Data Classes when compared to Blobs, as shown in Table 6. This smell often manifests in classes by exposing their fields (violating the information hiding principle) with excessive data handling. As ChatGPT is known for its high performance in natural language processing tasks, the model's ability to detect textual associations among the class members may have played an important role in detecting Data Class. Thus, the model's capability of detecting associations within textual data may have played an important role in identifying this smell.

Identification of Critical and Major Smells. ChatGPT reported a relevant improvement when detecting Data Classes and Long Methods smells categorized as Critical or Major. This outcome can be attributed to the following factors:

- *Clear Indicators:* Critical and Major smells may exhibit specific characteristics that distinguish them from other severity levels. For instance, the Critical Long Method occurrence in `JsonGenerator`[2] is substantially larger (106 LOC) than the Minor Long Method in `DisruptorComponent`[3] (24 LOC).
- *Task-Specific Prompts:* The structure of the prompt used may play an essential role in the model's ability to identify Critical and Major smells. We see this evidence in Table 6. While Prompt #2 explicitly enumerates the target smells, Prompt #1 remains generic. This difference in prompt design contributed to a higher performance of Prompt #2 across all Critical and Major smells, except for Feature Envy (Major). We plan to investigate prompts tailored for different severity levels. This approach may help ChatGPT at focusing on even more relevant particularities of each code smell, enhancing its detection performance.

Implications and Further Investigations. The varying performance of ChatGPT at detecting different code smells shed light on some dificulties regarding this task. The model presented better results with Data Classes and Long Methods at Critical and Major levels. However, detecting complex and context-dependent smells—like Blob and Feature Envy—remains challenging. We plan to investigate further the underlying factors that affect ChatGPT's performance across different code smells.

7.2 Integration of Contextual Information

As future work, we aim to integrate contextual information to improve the model's effectiveness. Particularly, we plan to adopt different techniques, including fine-tuning the model, crafting task-specific prompts, or combining both of them.

Prompt Design. We plan to incorporate relevant contextual information in the prompt to help the model identifying the smells. This mechanism can contribute to:

- *Flexibility:* We can extend the prompt description with structural information about the source code under analysis to cover the lack of contextual comprehension needed to detect some code smells. Besides, we can split the prompt into task-specific steps explaining ChatGPT about how to detect each smell properly. This strategy is usually called Chaing of Thoughts [6].
- *Interpretability:* We can incrementally refine the prompt description for particular smells based on its performance score. This strategy would help us emphasize which aspects of the prompt should be improved, thereby increasing its interpretability from the model's perspective.

Fine-Tuning the Model. This approach involves training ChatGPT on code-specific datasets consisting of samples code smells. Once the model is fine-tuned, it may be able to identify nuanced patterns and variations associated with the smells, thereby improving its performance. This mechanism can contribute to:

- *Adaptability:* Fine-tuning enables ChatGPT to learn specific features and patterns directly from labeled data, allowing the model to refine its representations for code smell detection.
- *Generalization:* Fine-tuning the model on a diverse codebase may enhance ChatGPT's performance at detecting smells among different programming languages, code styles, and application domains, leading to more robust detection performance.

[2]https://github.com/apache/zookeeper/blob/07c3aaf3d723fb3144c0aedc0c2b655325df70e9/zookeeper-contrib/zookeeper-contrib-loggraph/src/main/java/org/apache/zookeeper/graph/JsonGenerator.java/#L75-L213

[3]https://github.com/apache/camel/blob/8a85a70643c4d6eec2d3abddeea44ecb06c2f486/components/camel-disruptor/src/main/java/org/apache/camel/component/disruptor/DisruptorComponent.java/#L64-L108

References

[1] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement Using Large Language Models at Meta. In *32nd ACM Symposium on the Foundations of Software Engineering (FSE)*.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv:2107.03374 [cs]* (July 2021).

[3] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. [n. d.]. Code to Comment "Translation": Data, Metrics, Baselining & Evaluation. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2020-09). 746–757.

[4] Saki Imai. 2022. Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study. In *International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 319–321.

[5] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-Task Learning Based Pre-Trained Language Model for Code Completion. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 473–485.

[6] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-Train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. arXiv:2107.13586 [cs]

[7] Lech Madeyski and Tomasz Lewowski. 2020. MLCQ: Industry-Relevant Code Smell Data Set. In *24th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. Association for Computing Machinery, 342–347.

[8] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2023. Using Transfer Learning for Code-Related Tasks. *IEEE Transactions on Software Engineering (TOSEM)* 49, 4 (2023), 1580–1598.

[9] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. 2023. Prompt Engineering or Fine Tuning: An Empirical Assessment of Large Language Models in Automated Software Engineering Tasks. https://doi.org/10.48550/arXiv.2310.10508

[10] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2024. Using Large Language Models to Generate JUnit Tests: An Empirical Study. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*.

[11] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. 2021. A Systematic Literature Review on Bad Smells–5 W's: Which, When, What, Who, Where. *IEEE Transactions on Software Engineering (TSE)* 47, 1 (2021), 17–66.

[12] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using Pre-Trained Models to Boost Code Review Automation. In *44th International Conference on Software Engineering (ICSE)*. 2291–2302.

[13] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards Automating Code Review Activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 163–174.

[14] Peter H. Westfall, James F. Troendle, and Gene Pennello. 2010. Multiple McNemar tests. *Biometrics* 66, 4 (2010), 1185–1191. https://doi.org/10.1111/j.1541-0420.2010.01408.x

[15] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design.

[16] Ting Zhang, Ivana Clairine Irsan, Ferdian Thung, DongGyun Han, David Lo, and Lingxiao Jiang. 2022. Automatic Pull Request Title Generation. In *International Conference on Software Maintenance and Evolution (ICSME)*. 71–81.

[17] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2024. Measuring GitHub Copilot's Impact on Productivity. *Commun. ACM* 67, 3 (2024), 54–63.