On the use of LLMs for design pattern detection in software models*

Yassine Abdeljalil[1,*,†], Ansgar Radermacher[2,†], Marcos Didonet Del Fabro[3,†] and Chokri Mraidha[4,†]

Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Abstract

Design patterns are important for improving code reusability, maintainability, and overall software quality by providing standardized solutions. Most part of design pattern detection methods, such as static analysis and graph-based techniques, face challenges due to variability in pattern implementations, scalability issues, and ambiguity in design rationale. In addition, extracting and processing full code bases is complex and not always feasible due to resource limitations. Integrating design pattern detection with inference through Large Language Models (LLMs) and MDE techniques may overcome some of these limits by producing abstractions of large code bases. In this paper, we propose to align with Model-Driven Engineering (MDE) principles, helping in the automation of extraction of code bases in the form of UML models and injecting them into an LLM-based design-pattern recognition flow. We use the P-MART repository for design pattern detection to evaluate the solution's effectiveness. We compare different LLMs using models with and without comments. According to our findings, LLMs are able to identify a variety of Gang of Four (GoF) design patterns using UML models as input, but with limits, particularly when combining multiple patterns.

Keywords

LLMs, Software Models, MDE, GoF design patterns, P-MART

1. Introduction

Design patterns are reusable solutions to a common problem encountered during software design. They improve code reusability, maintainability, and overall software quality by providing standardized solutions to software engineering problems [1]. Adopting design patterns is a best practice in software development that helps minimize complexity and fosters a shared vocabulary for design discussions [2].

Traditional methods for detecting design patterns, like rule-based, graph-based, and static analysis, face several notable challenges. The variation in how the patterns are implemented is one of the main issues. Design patterns may have varied implementations across various codebases, and it becomes challenging for these techniques to identify all occurrences, leading to missed detections or false positives [3].

When analyzing big and complicated codebases, these methodologies frequently suffer with scalability, leading to time-consuming and resource-demanding processes [4]. Moreover, the complexity of design patterns is increased by ambiguity in the representation of design rationale goals, which makes identification even more difficult [5]. These factors collectively contribute to the low success rate of classical detection methods, as reported in state-of-the-art studies [4].

Thanks to their advanced text analysis capabilities, Large Language Models can be used in many software engineering (SE) tasks, including code generation, documentation, bug detection, and more complex tasks [6]. However, analyzing entire codebases remains a complex challenge due to the

Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10-13, 2025

*Corresponding author.

†These authors contributed equally.

yassine.abdeljalil@cea.fr (Y. Abdeljalil); ansgar.radermacher@cea.fr (A. Radermacher); marcos.didonetdelfabro@cea.fr (M. D. D. Fabro); chokri.mraidha@cea.fr (C. Mraidha)

0009-0003-0148-3008 (Y. Abdeljalil); 0000-0001-6257-4657 (A. Radermacher); 0000-0002-8573-6281 (M. D. D. Fabro); 0000-0003-2993-5734 (C. Mraidha)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2025-12-10

CEUR-WS.org/Vol-4122/paper17.pdf

extensive effort and resources required. For this reason, integrating design pattern detection within Model-Driven Engineering (MDE) may offer advantages by generating abstractions that simplify and enhance the analysis of large codebases [7].

This study investigates the potential of LLMs to detect design patterns from UML representations by addressing two primary research questions:

- **RQ1:** To what extent can Large Language Models accurately detect GoF design patterns from UML models?
- **RQ2:** How do different input conditions such as the presence of comments and the combination of multiple design patterns affect the performance of LLMs in design pattern detection?

To answer these questions, we provide a comparative analysis of the extent to which LLMs coupled with UML models can detect GoF design patterns [8]. We apply prompt engineering technique to focus on specific design pattern scenarios, including the use of single or combination patterns. In addition, we provide input models with and without comments to verify how much extra information helps with pattern detection. We process an extensive dataset of UML models to assess the model's performance, calculating metrics such as accuracy, recall, and F1-Score using different LLMs. These metrics provide an understanding of the LLM's capability in design pattern recognition using UML models, as well as where the extracting and processing workflow may be improved.

The structure of this paper is as follows. Section 2 describes the methodology employed to prepare data and assess the current capabilities of LLMs in performing single and multiple design pattern detection within modeling tasks. The results of these experiments are presented and discussed in Section 3. Section 4 reviews some related works and situates our approach within the existing research landscape. Finally, Section 5 concludes the paper with some closing remarks.

2. Design Pattern Extraction Flow

This section outlines our methodology for evaluating the capability of Large Language Models in recognizing and classifying design patterns. Figure 1 summarizes the different steps that structure our approach. Below, we give more details about each step that forms our approach.

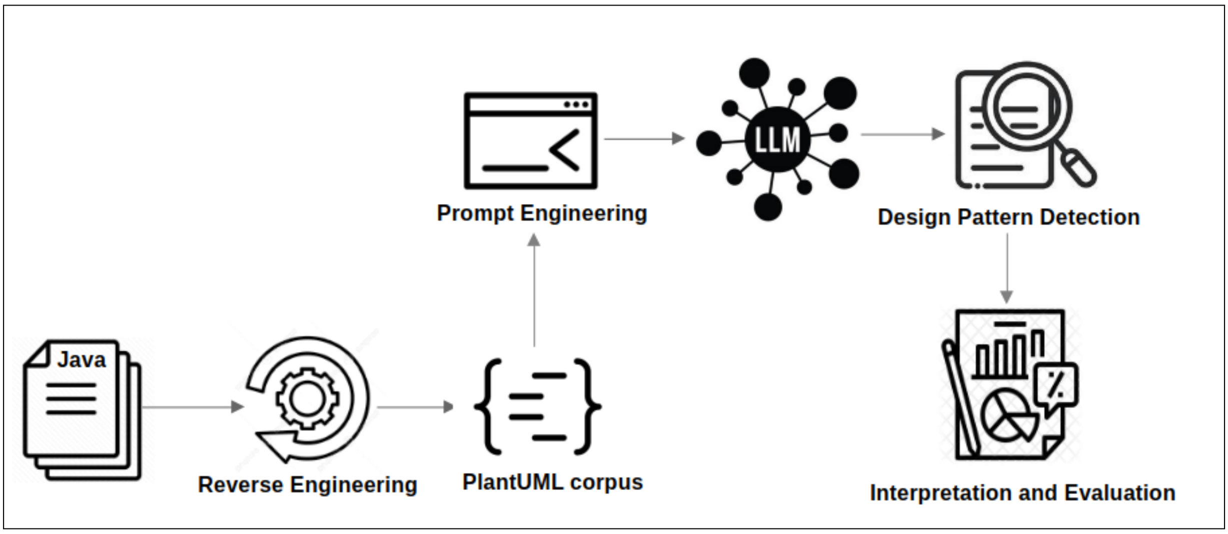

Figure 1: Our approach for design pattern detection in UML models.

- *Reverse engineering existing source codes*
  Given the limited availability and scarcity of datasets related to design pattern implementation in UML model format, our initial step involves reverse engineering existing source code that

implements a variety of design patterns. For this reason, we choose P-MART [9], a repository that comprises several unmodified open-source Java projects. P-MART includes annotations that identify micro-architectures related to these projects' Gang of Four (GoF) design patterns. Transforming source code into UML models enables the generation of a varied dataset that encapsulates the structural signatures and relationships intrinsic to each design pattern. This process analyzes code to extract class hierarchies, method interactions, and object collaborations that characterize distinct design patterns.

- *Building a PlantUML corpus*
  We chose PlantUML because it provides a clear and simple way to describe UML models using textual presentation [10]. This text format captures the key parts of a model, like class definitions, method signatures, access modifiers, and relationships such as inheritance and association. This ensures that the semantic content of UML models is preserved for analysis by the LLM, thus allowing the extraction of meaningful information. However, certain complex details of UML models might get lost in translation, which could affect the extraction of meaningful information by large language models.
- *Prompt creation*
  With the generated PlantUML code, we use an automated process to handle the corpus and insert the corresponding files into the prompt. Rather than embedding the code manually, the files are retrieved from a specified folder and integrated into the prompt in a structured manner. The prompt is designed to guide the LLM in analyzing the input for design pattern detection. Figure 2 shows an example of the prompt used for both single and multiple design pattern detection. By combining clear instructions with well-organized input files, we help the LLM produce responses that follow the expected format and contain the appropriate level of detail.

> **User:** You are a software architect. Analyze the following PlantUML code and identify all relevant GoF (Gang of Four) software design patterns it represents.
> For each identified pattern, please respond with its name in uppercase, followed by a concise explanation justifying your answer, and provide a confidence score.

Figure 2: GoF Design Pattern Identification Prompt for PlantUML Code

- *Design pattern processing*
  Once the Java code is converted into PlantUML and paired with the crafted prompt, the next step is to run it through a large language model (LLM). At this stage, the LLM takes the PlantUML and the prompt as input and then processes that information to show how well it understands the structure and relationships within the code. This processing step leverages the LLM's capabilities to provide insights or identify patterns in the PlantUML code.
- *LLM's output*
  The LLM identifies the detected design pattern along with detailed explanations. It uses specific elements from the PlantUML code to justify its identification and illustrates how the code carries out the pattern's distinctive features. For example, it might highlight that the Singleton class has a private constructor and a static ‘getInstance()‘ method, key traits of the ‘Singleton‘ pattern. Similarly, the Observer pattern might note the presence of an ‘attach()‘ method, a ‘notify()‘ method, and an ‘Observer‘ interface implemented by concrete observer classes.
- *Result interpretation and Evaluation*
  In the final stage, it is necessary to interpret and verify the LLM's output by comparing the detected patterns and explanations against the intended designs in the original PlantUML dataset. This crucial verification process is a benchmark for ensuring our approach's accuracy, reliability, and effectiveness. By assessing whether the LLM correctly identified the patterns and provided

valid explanations, we can evaluate its capability to understand and analyze software design patterns from PlantUML code. We use metrics like precision, recall, and F1 score to quantitatively measure the LLM's performance. Additionally, employing a confusion matrix helps us visualize the model's prediction efficacy by detailing true positives, false positives, true negatives, and false negatives, which aids in identifying specific strengths and weaknesses of the LLM. These evaluation tools and methods ensure a thorough understanding of the model's capabilities and guide future optimizations and refinements.

3. Design pattern detection evaluation

We structure the evaluation of the effectiveness of integrating LLMs and MBSE in design pattern identification into two phases. In the **first phase**, we assess the LLM's ability to detect single and multiple design patterns within UML models. This assessment is the first empirical qualitative analysis to determine how well the model understands different design structures. This first phase enables refining the extraction and prompting process on a single file/pattern basis. We iteratively tested different prompt formulations, adjusting the level of detail, phrasing, and contextual information provided. This step ensures that the model's responses are more consistent when analyzing individual UML files and detecting patterns at varying levels of complexity.

In the **second phase**, we scale our evaluation by automating the processing of a PlantUML corpus, enabling the LLM to recognize and classify design patterns across multiple PlantUML files. This allows us to compute key performance metrics such as precision, recall, and F1-score, quantitatively assessing the model's accuracy and reliability at a larger scale.

Due to the limited availability of datasets containing software models with commonly used design patterns, we opted for reverse engineering from source code. To achieve this, we leveraged P-MART [9], a database of Java implementations of GoF design patterns. Using Papyrus SW Designer [11], we first transformed the Java code into UML class diagrams. These diagrams were then converted into PlantUML format, enabling the LLM to process and analyze them automatically. To analyze and compare the reasoning processes of different large language models, we use three distinct models: DeepSeek-R1:70b, Llama3.3:70b, and Qwen2.5-coder:32b, and for all our tests, we use the prompt that we presented in Figure 2. To ensure consistency and fairness in our comparisons, we used each model's default settings without modifying parameters such as temperature or top-k. This approach allows us to gain deeper insights into their decision-making patterns, strengths, and limitations.

3.1. Single Design Pattern Detection

In this first scenario, we evaluate the ability of LLMs to identify standard design patterns, specifically Singleton and Observer, both of which play a fundamental role in software design. We begin our analysis of the Singleton pattern by representing it as a UML class diagram, which is then translated into PlantUML code. Figure 3 illustrates the PlantUML representation alongside the Llama3.3 model's response. The LLM successfully identifies the Singleton pattern and justifies its answer based on the presence of a private constructor, a static instance captain within the Captain class, and the static method getCaptain(). Notably, we incorporate a confidence score in the prompt to gauge the model's certainty in its classification. This score provides insight into how strongly the model believes its prediction, allowing for a more structured evaluation of its reliability. We also verify the correctness of the response against the ground-truth annotations provided by the P-MART dataset. This dual approach allows for both a subjective measure of the model's confidence and an objective validation of its output, contributing to a more robust evaluation of its performance.

In our subsequent analysis, we focus on the Observer pattern, utilizing its dedicated PlantUML representation as depicted in Figure 4. This PlantUML code effectively showcases the interaction between the subject and observer classes, highlighting the notification mechanism central to the Observer pattern. The Llama3.3 model accurately identifies this structure as the Observer pattern,

```
@startuml
package Book {
    class Captain {
        - Captain()
        + captain(): Captain
        + getCaptain(): Captain
    }

    class MainExample {
        + main(String[] args)
    }
}
@enduml
```

Assistant: SINGLETON . The Captain class has a private constructor and provides a method captain() to create instances of itself, as well as a method getCaptain() to retrieve an instance, which suggests that it is implementing the Singleton pattern to control access to its instances. Confidence score: 90

Figure 3: PlantUML and Resulting output for Singleton pattern detection

justifying by noting that a publisher maintains a list of listeners, offers a method for their registration, and notifies them upon the occurrence of specific events.

```
@startuml
class ButtonClickPublisher {
    - listeners: List
    + addButtonClickListener(listener: ButtonClickListener)
    + notifyButtonClickListeners()
    + main()
}

class ButtonClickListener {
    + actionPerformed(event: java.awt.event.ActionEvent)
}

class GUIApp {
    + main(args: String)
}
@enduml
```

Assistant: OBSERVER. This code represents the Observer pattern because it includes a subject (ButtonClickPublisher) that maintains a list of observers (ButtonClickListener) and notifies them when a specific event occurs (button click). The addButtonClickListener method allows observers to register with the subject, and the notifyButtonClickListeners method is used to notify all registered observers. Confidence score: 90

Figure 4: PlantUML and Resulting output for Observer pattern detection

3.2. Multiple Design Pattern Detection

In this experiment, we seek to evaluate the LLM's ability to recognize and interpret multiple design patterns working together within a single system. In real-world applications, it's common to combine patterns rather than using them in isolation for each class. To explore this, we create a PlantUML diagram that integrates both the Adapter and Composite patterns.

The Adapter pattern is represented by the ShapeTranslator class, which acts as an adapter between the Graphic and OldShape interfaces. The ShapeTranslator class implements the Graphic interface and contains an instance of OldShape. This allows the OldShape class to be used where a Graphic is expected, effectively adapting one interface to another.

The Composite pattern is depicted through the Graphic interface, which is implemented by both the Circle and GraphicGroup classes. The GraphicGroup class is a composite that can hold multiple Graphic

Table 1
PlantUML dataset for single design pattern detection

Project	#DP	#PlantUML	Classes/Interfaces		#Tokens (UML)		#Tokens (Java)
			No Cmt	With Cmt	No Cmt	With Cmt	With Cmt
QuickUML2001	7	12	43	104	1394	3613	35721
Lexi v0.1.1 alpha	5	12	20	50	1993	6505	28264
JRefactory v2.6.24	26	44	130	173	5603	19986	276216
JUnit v3.7	8	14	52	68	1804	2862	18300
JHotDraw v5.1	23	46	186	205	13276	26367	44109

objects (such as individual Circle instances or other GraphicGroup instances), and it is responsible for drawing them as a whole. This structure enables clients to treat individual Circle objects and entire graphics groups uniformly, simplifying the management of complex graphical structures. Figure 5 illustrates the PlantUML we use and the result generated by Llama3.3 in response to the prompt shown in Figure 2.

```
@startuml
interface Graphic {
  + draw(): void
}
class Circle implements Graphic {
  + draw(): void
}
class GraphicGroup implements Graphic {
  - graphics: List<Graphic>
  + draw(): void
}
interface OldShape {
  + render(): void
}
class ShapeTranslator implements Graphic {
  - oldShape: OldShape
  + draw(): void
}
class OldCircle implements OldShape {
  + render(): void
}
ShapeTranslator --> OldShape
GraphicGroup *-- Graphic
@enduml
```

Assistant: Based on the provided PlantUML code, the following GoF software design patterns are represented:
COMPOSITE: This pattern is represented by the GraphicGroup class, which contains a list of Graphic objects and implements the Graphic interface. The GraphicGroup class acts as a container for other graphics, allowing it to be treated as a single unit. Confidence score: 90%.
ADAPTER: This pattern is represented by the ShapeTranslator class, which implements the Graphic interface and contains an OldShape object. The ShapeTranslator class adapts the OldShape interface to match the Graphic interface, allowing OldShape objects to be used in a context that expects Graphic objects. Confidence score: 95%

Figure 5: PlantUML and Resulting output for multiple pattern detection

3.3. Comparative Evaluation

In this phase, we process PlantUML corpus generated through the reverse engineering of Java code sourced from the P-MART database using Papyrus Designer tool [11]. The Papyrus Designer tool allows us to reverse engineer the Java code into UML models with the option of including or excluding comments. First, we analyze individual design patterns by extracting and examining their corresponding UML representations. Next, we construct and process multiple design pattern combinations based on the same PlantUML corpus. Table 1 presents the number of design patterns and the total number of PlantUML files used in our study. Since the PlantUML model does not contain method bodies, the number of tokens is an order of magnitude smaller compared to Java code. This has the advantage that it fits more likely into the LLM's context window, and inference consumes fewer resources. However, it may imply that patterns for which characteristic elements can be found in method bodies are less likely detected compared to analyzing source code.

Our evaluation serves a dual purpose: first, to compare the ability of three LLMs, Llama3.3, DeepSeek-

R1, and Qwen2.5-Coder, to identify design patterns using PlantUML as input, which contains less information than the complete source code, since it also contains method implementations. Second, to assess the impact of comments on detection performance. To achieve this, we analyze two versions of the PlantUML corpus: one with comments and one without. We construct a confusion matrix for each model and compute key performance metrics, including accuracy, recall, and F1-score. This comparative analysis allows us to evaluate the strengths and limitations of different LLMs in design pattern recognition while also determining whether the inclusion of comments improves detection accuracy and reliability.

We perform multiple queries to enhance the reliability of our evaluation. Specifically, for each PlantUML file, both with and without comments, we call each LLM three times to identify GoF design patterns. From these three attempts, we select the prediction with the highest confidence score returned by the model. The LLM itself calculates the confidence score as a result of the request. While there may be differences between calculations, it is the first rapid method to rank the provided results. This approach helps mitigate variability in LLM responses and ensures that our assessment reflects the model's most confident prediction. By applying this method across all evaluated models, Llama3.3, DeepSeek-R1, and Qwen2.5-Coder, we aim to obtain a more robust comparison of their ability to recognize design patterns and determine whether comments contribute to improved detection performance.

The analysis of model performance metrics highlights the varying impact of comments on the accuracy and effectiveness of different LLMs in design pattern recognition tasks. Llama3.3 exhibits the most significant improvement when comments are provided, with accuracy increasing from 0.40 to 0.50 and the F1-score rising from 0.39 to 0.48, as shown in Figure 6, indicating that this model effectively leverages contextual information. Qwen2.5-Coder notably improves when comments are provided, with accuracy increasing from 0.39 to 0.44 and the F1-score rising from 0.37 to 0.42, as shown in Figure 8. DeepSeek-R1 shows a more significant gain, with accuracy improving from 0.32 to 0.44 and the F1-score increasing from 0.35 to 0.46, particularly benefiting from enhanced recall and precision (Figure 7). Despite these variations, all three models consistently identify common design patterns such as Singleton, State, Strategy, Observer, and Decorator. However, they struggle with less frequent or more structurally complex patterns, highlighting their limitations in fully understanding and generalizing all design patterns. These findings suggest that while comments can enhance model performance, their impact depends on the model's inherent ability to process contextual cues and recognize structural relationships.

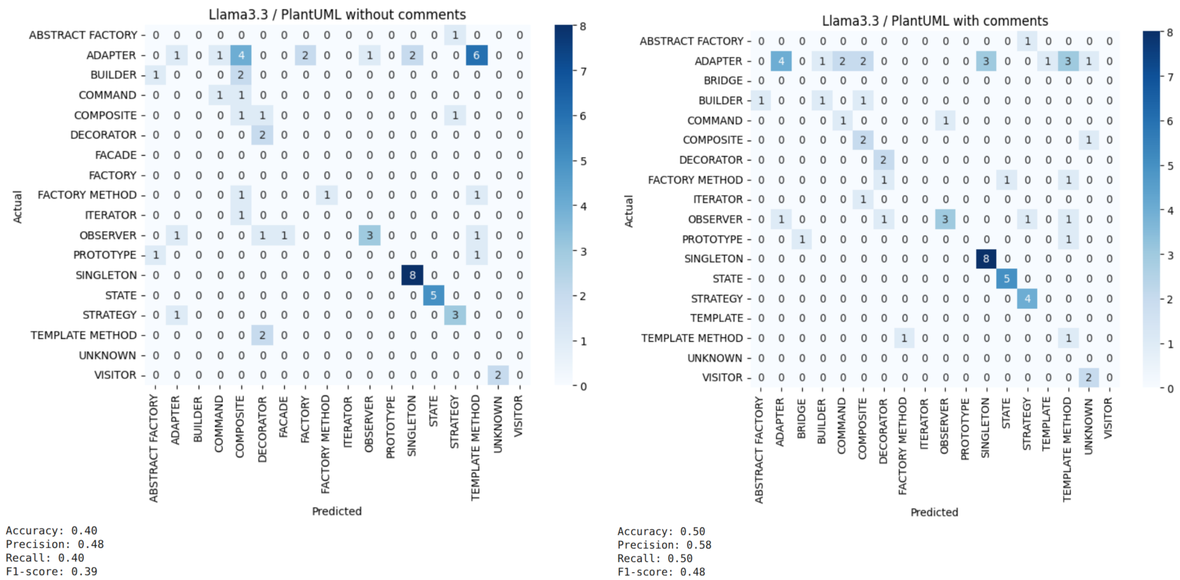

Figure 6: Confusion Matrices using Llama3.3: With and Without Comments

We conduct a second evaluation to detect multiple design patterns within PlantUML files, building

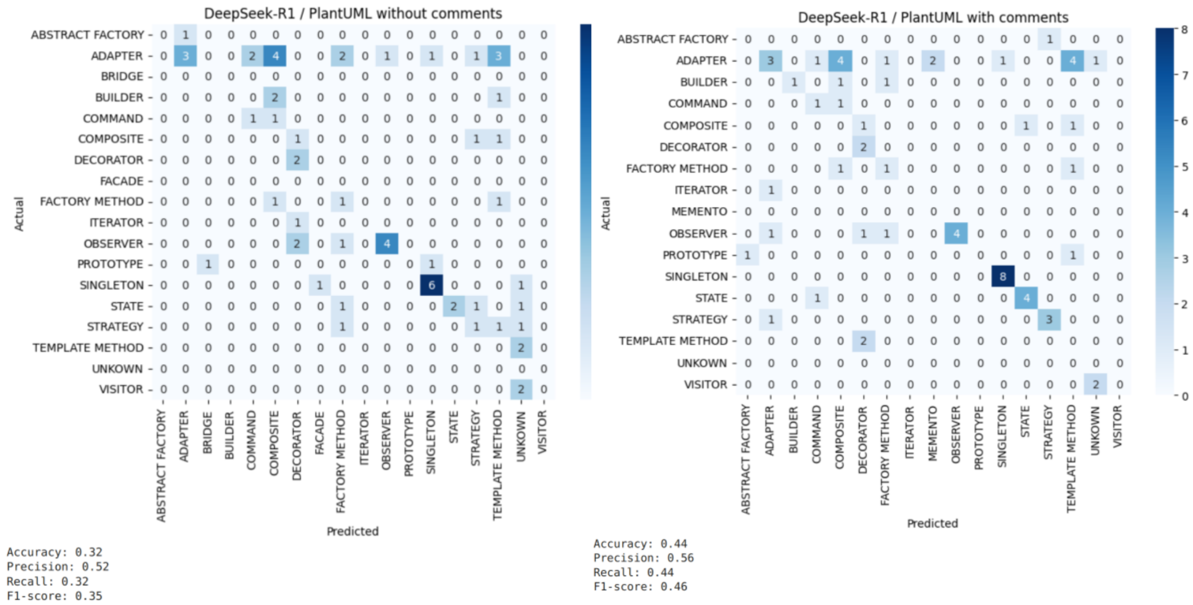

Figure 7: Confusion Matrices using DeepSeek-R1: With and Without Comments

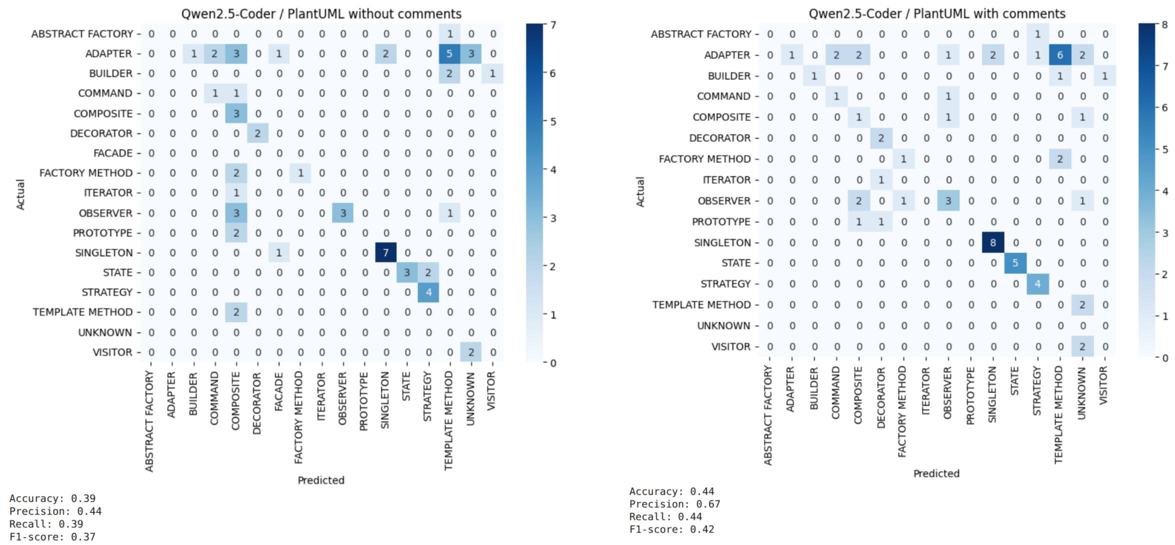

Figure 8: Confusion Matrices using Qwen2.5-Coder: With and Without Comments

upon our earlier analysis of single design pattern detection. For this evaluation, we utilize the same LLMs: Llama3.3, Qwen 2.5-Coder, and DeepSeek-R1, which were employed in the initial analysis. Additionally, we use the same PlantUML corpus (Table 1) to construct combinations of design patterns from the three main categories: Creational, Structural, and Behavioral patterns. Representative examples include {Observer, Singleton} and {State, Strategy}. For each combination, we generate between 20 and 30 PlantUML files, each reflecting the coexistence and interaction of the selected patterns. The LLMs are then tasked with identifying all present patterns in each file. To ensure consistency in the evaluation, we reuse the same prompt shown in Figure 2.

Table 2 shows the results of multiple design pattern detection using Llama3.3, Qwen2.5-Coder, and DeepSeek-R1, with and without comments in PlantUML files. Overall, Llama3.3 achieves the highest accuracy with scores ranging from 0.32 to 0.53, followed by DeepSeek-R1, while Qwen2.5-Coder performs the worst. Including comments improves detection for all models, with Llama3.3 benefiting

Table 2
Results for multiple design pattern detection

Multiple DP	Llama3.3		Qwen2.5-Coder		DeepSeek-R1	
	No Cmt	With Cmt	No Cmt	With Cmt	No Cmt	With Cmt
Observer & Singleton	0.45	0.53	0.35	0.41	0.38	0.48
Observer & Adapter	0.36	0.49	0.27	0.3	0.35	0.47
State & Strategy	0.43	0.51	0.3	0.35	0.36	0.43
Decorator & State	0.39	0.45	0.34	0.39	0.37	0.42
Command & Composite	0.35	0.42	0.29	0.37	0.33	0.39
Iterator & Abstract Factory	0.32	0.39	0.23	0.28	0.35	0.37

the most. Some pattern combinations, like Observer & Singleton, are consistently detected, while others, such as Iterator & Abstract Factory, are less frequently identified. DeepSeek-R1 consistently outperforms Qwen2.5-Coder, particularly without comments, making it a more reliable choice for design pattern detection.

A scoring system is applied to assess the LLM's predictions: if the LLM correctly identifies all expected patterns, it receives a score of 1. If it detects only some of the expected patterns, the score is 0.5, and if it detects none of the patterns, the score is 0. The scores for each file are summed, and the average score across all files is calculated to evaluate the overall accuracy of the LLM in detecting multiple design patterns.

This evaluation highlights the impact of comments on model performance and demonstrates that while all models show some improvement, Llama3.3 consistently outperforms the others. Additionally, DeepSeek-R1's ability to match or exceed Qwen 2.5-Coder in most cases suggests that it is a more reliable choice for design pattern detection, particularly when comments are absent. This analysis provides deeper insight into the LLMs' capabilities in differentiating and recognizing multiple design pattern combinations compared to the earlier single-pattern detection evaluation.

4. Related Work

Design pattern detection has been an active research area for decades. Early work focused on rule-based and static analysis techniques, where manually defined rules and abstract syntax tree (AST) or graph-based representations were used to identify instances of the Gang of Four design patterns. For example, methods based on similarity scoring of program graphs and rule extraction [12] achieved high recall but required substantial manual effort to define pattern-specific rules. Other approaches have leveraged software metrics combined with traditional machine learning algorithms such as k-nearest neighbors (KNN), decision trees, and support vector machines (SVM) to detect design patterns based on quantitative characteristics extracted from UML models or source code [13] [14].

More recent advances have explored deep learning techniques. In [15], researchers proposed methods that build semantic representations of source code, often via Word2Vec or convolutional neural networks (CNN), to extract features for pattern detection automatically. These approaches reduced the need for manual feature engineering but typically focused on detecting single patterns in isolated code units.

Parallel to these efforts, the emergence of LLMs has opened a new avenue for design pattern detection. Recent studies such as "Do Code LLMs Understand Design Patterns?" [16] and the work by Schindler et al. on "LLM-Based Design Pattern Detection" [15] demonstrate that pretrained LLMs (e.g., CodeBERT, CodeGPT, RoBERTa) can capture both syntactic and semantic information from source code without requiring extensive task-specific pretraining. These LLM-based methods typically extract embeddings from code and employ simple classifiers (e.g., k-nearest neighbors) to identify design pattern roles. They show performance comparable to state-of-the-art methods while reducing the manual effort for feature extraction and rule specification.

Our work builds on these recent LLM-based approaches but addresses two key challenges. First, unlike most existing methods that target a single design pattern per code unit, our approach is designed to detect both single and multiple pattern instances concurrently. This reflects more realistic scenarios in which classes often participate in multiple patterns. Second, we incorporate PlantUML models to represent class structures at a higher level of abstraction without requiring method-level implementations, which enables broader generalization across different codebases.

5. Conclusions

We analyzed the utilization of LLMs, coupled with PlantUML as input, to detect Gang of Four (GoF) design patterns. For RQ1, our results show that while LLMs are capable of identifying both single and combined design patterns from UML models even without method-level details, the overall accuracy is not very high. This suggests that relying solely on LLMs may not be sufficient for precise detection and opens up opportunities to explore additional techniques or hybrid approaches to improve performance.

For RQ2, we analyzed how adding comments and combining multiple design patterns in a single UML model affect detection. Adding brief comments led to a slight improvement in performance, and LLMs were able to detect multiple coexisting patterns to some extent. However, their effectiveness decreased as the complexity of the model increased.

A significant part of the effort in establishing this workflow involved file extraction, curation, and integration to adapt the input to the expected LLM format (context + prompt). This highlights the importance of integrated analysis workflows to expedite the adaptation for different scenarios.

Future work could include defining an integrated framework for model-based analysis or conducting an extended comparison with code-based approaches.

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT, Grammarly in order to: Grammar and spelling check, Paraphrase and reword. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

Acknowledgements This work was partially supported by CodeCommons BPI grant n. DOS0239480/00.

References

[1] V. K. Thatikonda, H. R. V. Mudunuri, Leveraging design patterns to architect robust and adaptable software systems, International Journal of Computer Applications (2023). URL: https://api.semanticscholar.org/CorpusID:265085212.

[2] N. Qamar, A. A. Malik, Impact of design patterns on software complexity and size, April 2020 (2020). URL: https://api.semanticscholar.org/CorpusID:219115046.

[3] C. Schindler, A. Rausch, Llm-based design pattern detection, arXiv preprint arXiv:2502.18458 (2025).

[4] H. Yarahmadi, S. M. H. Hasheminejad, Design pattern detection approaches: a systematic review of the literature, Artificial Intelligence Review 53 (2020) 5789–5846.

[5] L. Aladib, S. P. Lee, Pattern detection and design rationale traceability: an integrated approach to software design quality, IET Software 13 (2019) 249–259.

[6] Q. Zhang, C. Fang, Y. Xie, Y. Zhang, Y. Yang, W. Sun, S. Yu, Z. Chen, A survey on large language models for software engineering, ArXiv abs/2312.15223 (2023). URL: https://api.semanticscholar.org/CorpusID:266551742.

[7] J. Di Rocco, D. Di Ruscio, C. Di Sipio, P. T. Nguyen, R. Rubei, On the use of large language models in model-driven engineering, Software and Systems Modeling (2025) 1–26.

[8] J. Hunt, Gang of four design patterns, Scala Design Patterns: Patterns for Practical Reuse and Design (2013) 135–136.

[9] Y.-G. Guéhéneuc, P-mart: Pattern-like micro architecture repository, Proceedings of the 1st EuroPLoP Focus Group on pattern repositories (2007) 1–3.

[10] A. Roques, PlantUML: Open-Source Tool for UML Diagrams, 2009. URL: https://plantuml.com/.

[11] Papyrus team, Papyrus Software Designer, https://gitlab.eclipse.org/eclipse/papyrus/org.eclipse.papyrus-designer/-/wikis, 2025. [Online; accessed April-2025].

[12] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, Design pattern detection using similarity scoring, IEEE transactions on software engineering 32 (2006) 896–909.

[13] S. Uchiyama, H. Washizaki, Y. Fukazawa, A. Kubo, Design pattern detection using software metrics and machine learning, in: First international workshop on model-driven software migration (MDSM 2011), 2011, p. 38.

[14] N. Nazar, A. Aleti, Y. Zheng, Feature-based software design pattern detection, Journal of Systems and Software 185 (2022) 111179. URL: https://www.sciencedirect.com/science/article/pii/S0164121221002624. doi:https://doi.org/10.1016/j.jss.2021.111179.

[15] S. K. Pandey, S. Chand, J. Horkoff, M. Staron, M. Ochodek, D. Durisic, Design pattern recognition: a study of large language models, Empirical Software Engineering 30 (2025) 69.

[16] Z. Pan, X. Song, Y. Wang, R. Cao, B. Li, Y. Li, H. Liu, Do code llms understand design patterns?, arXiv preprint arXiv:2501.04835 (2025).