LLM-Based Design Pattern Detection

Christian Schindler and Andreas Rausch
Institute for Software and Systems Engineering
Clausthal University of Technology
Clausthal-Zellerfeld, Germany
e-mail: {christian.schindler | andreas.rausch}@tu-clausthal.de

***Abstract*—Detecting design pattern instances in unfamiliar codebases remains a challenging yet essential task for improving software quality and maintainability. Traditional static analysis tools often struggle with the complexity, variability, and lack of explicit annotations that characterize real-world pattern implementations. In this paper, we present a novel approach leveraging Large Language Models (LLMs) to automatically identify design pattern instances across diverse codebases. Our method focuses on recognizing the roles classes play within the pattern instances. By providing clearer insights into software structure and intent, this research aims to support developers, improve comprehension, and streamline tasks such as refactoring, maintenance, and adherence to best practices.**

Keywords-Design Pattern detection; Large Language Model.

I. Introduction

Identifying design pattern instances in code is a valuable goal as it enables a deeper understanding of the structural and behavioral principles underlying software systems. By uncovering these patterns, developers and other stakeholders can gain insights into code quality, maintainability, and adherence to best practices, even in unfamiliar code bases [1]. Automating this process can significantly reduce the time and effort required for code comprehension, facilitate knowledge transfer among teams, and improve software evolution and refactoring efforts. Furthermore, it can aid in identifying reusable components, fostering consistency, and enhancing the overall robustness of software design.

It is a challenging task due to several factors. Design patterns are often implemented with significant variations tailored to specific use cases, making consistent recognition difficult [2]. Developers frequently deviate from canonical implementations or introduce domain-specific adaptations, complicating detection efforts [3]. Additionally, design patterns are typically embedded implicitly within the code's structure and behavior, rather than being explicitly annotated or identifiable by a set of keywords, requiring a deep contextual understanding of the code, its dependencies, and its intent. Furthermore, applying pattern detection techniques to large, complex, and poorly documented code bases poses scalability challenges, as the sheer volume of code and intertwined dependencies can overwhelm traditional approaches. These factors collectively highlight the complexity of automating design pattern identification in diverse and unfamiliar code bases.

We have defined the following Research Questions (RQs). RQ1: How can LLMs be leveraged to automatically detect and annotate design pattern instances in software codebases? RQ2: How good can LLMs detect and annotate design pattern instances in software codebases? RQ3: What are challenges/limitations faced by LLMs in identifying design pattern instances in code bases, and how can these be addressed?

This paper is organized as follows. In Section 2, we motivate the work and define the main research task. Section 3 surveys related work, while Section 4 details the experimental setup and describes the dataset. Section 5 presents the experimental results. Finally, Section 6 concludes the paper and outlines directions for future research.

II. Motivating example

The idea of a design pattern in software engineering is to provide a reusable, general solution to a commonly occurring problem within a given context in software design. Design patterns encapsulate best practices and proven strategies for solving these problems, offering a structured approach to building robust, maintainable, and flexible software systems. The design pattern is described on a conceptual level, defining roles and their specific responsibilities within the pattern to achieve its intended design purpose.

A design pattern instance refers to the concrete implementation of a design pattern within a specific piece of software. In this context-specific realization, the roles defined by the design pattern are embodied by individual components, such as classes, objects, or packages, which collaboratively fulfill the pattern's intended structure and behavior.

The task we want to work on is to localize such design pattern instances in code. We want to detect the design pattern applied with the respective pairs of components and their roles.

III. Related Work

A. Design Pattern Detection

Detecting design pattern instances in existing code is a desirable task in later stages of a software system's lifecycle, particularly for maintenance purposes and to facilitate the onboarding of new developers. Various approaches have been proposed to address this challenge.

Matching-based approaches are common, such as those that rely on similarity scores [4] or graph structures [5] [6]. They have several limitations. Firstly, they often require a high degree of structural similarity between the detected instances and the target pattern. For example, they may mandate an identical number of roles or a specific implementation style. Secondly, the computational cost of the detection process can be significant, depending on the chosen matching strategy.

 ISBN: 978-1-68558-263-0

Furthermore, multi-stage approaches have been explored. These often involve a learning phase [7] [8] or a pattern definition phase [9] [10] prior to the actual detection of design pattern instances. Learning-based approaches in the first stage are limited by the requirement of a substantial amount of annotated training data. This data must encompass the necessary diversity of design pattern instances for the patterns of interest.

(Semi-)formal definition of design pattern structures also faces limitations. These include the expressiveness of the chosen language, the mapping of abstract concepts to language-specific constructs, and the inherent difficulty of defining patterns at this level of abstraction. The quality of this definition significantly impacts the subsequent detection phase and the accuracy of the results.

Several approaches focus on classifying source code as design pattern instances based on fixed-length inputs, such as a single class [11] [12] [13]. These methods typically work well for small or well-defined code segments where the scope is narrowly confined. However, they may fail to capture the broader context of how multiple classes or modules interact, leading to fragmented or incomplete representations of the software. In an attempt to handle larger portions of code, some methods aggregate metrics over entire modules or files—such as by summing values into a fixed-size feature vector [14]. While this strategy can reduce dimensionality and simplify processing, it tends to obscure important semantic and structural details because numerous code properties get merged into a single set of features. FeatRacer [15] addresses feature location by combining manual annotations with automated suggestions. It uses machine learning to predict missing feature recordings and guide developers, resulting in more accurate and complete feature information.

B. Large Language Models

LLMs are advanced language models that excel at understanding and processing human language [16]. They handle tasks with minimal examples, especially involving text and data tables [17]. With larger models the performance increases [18]. Generative Pre-trained Transformers (GPTs) are powerful transformer-based models with broad applications (e.g., in education, healthcare) and challenges like high computational demands, interpretability, and ethical issues [19]. Extensively studied in computer science, GPTs relates to AI, language processing, and deep learning, as Cano [20] noted. It also aids engineering design by generating novel ideas, as Gill [21] and Zhu [22] discovered.

IV. Experiment

A. Data Set of Design Pattern Instances

The data used in this work was originally collected and published by [23], and has since been adopted in various subsequent studies for benchmarking. In addition to annotating design pattern instances, the authors also provided specific versions of the corresponding source code. In this paper, we focus on instances of the *Composite* design pattern.

B. Experimental setup

The experimental setup, depicted in Figure 1, is discussed in this section.

1) Data Preparation: During this phase, we examined the available annotated design pattern instances to ensure their suitability for analysis. We identified and filtered out instances where not all associated source code files were available, as incomplete datasets could compromise the reliability of subsequent steps. Furthermore, we implemented a basic yet essential data cleaning procedure to improve the consistency of the annotations. This involved trimming unnecessary blank spaces in the Extensible Markup Language (XML) files containing the ground truth annotations, ensuring that formatting inconsistencies did not introduce errors or ambiguities.

2) Prompt Preparation: Figure 1 illustrates the structure of the prompts. The first prompt includes a general setup section that provides context for the LLM, clearly defined instructions, additional remarks, and a sample consisting of a source code snippet alongside the design pattern instance annotation.

In addition to following the prompt structure, we need to consider limiting factors in preparing the prompts A first restriction for the prompts is the available token size of the LLM. We used ChatGPT4 and the predecessor ChatGPT3.5, both with a limit of 128k tokens.

Because this represents the upper bound, our interaction with the LLM must include both prompts we provide and the expected answer, which will be much smaller than the input portion. To determine which classes to include in the provided example snippet and in the snippet for which the design pattern needs to be identified, we first identify the common root package shared by all classes participating in the ground truth annotation. For instance, the classes *a.b.c.d.ClassA* and *a.b.e.f.ClassB* share the common root package *a.b.*.

Table I provides a comprehensive overview of all remaining examples of the Composite design pattern after the Data Preparation step. For each example, it includes details about the originating project, the identified common root package, and the number of classes that (i) actively participate in the design pattern and (ii) belong to the common root package. To further optimize the input size, we have removed all comments from the classes. This reduction ensures that only essential code components remain, making the input more concise and focused for subsequent analysis and processing.

The output of this step is a collection of prompt messages. Each prompt includes a dedicated example containing a source code snippet paired with its ground truth annotation for the design pattern instance, as well as a second source code snippet from a different Java project. All permutations of the two source code snippets are considered, provided their combined token count falls within the token limit of the LLM. The order of the pairings is significant, as one snippet serves as the provided example while the other is the source code sample to be analyzed and annotated.

3) Utilization of the LLM: The actual prediction task is performed using ChatGPT3.5 and 4, though other LLMs could also be used due to their similar handling and functionality.

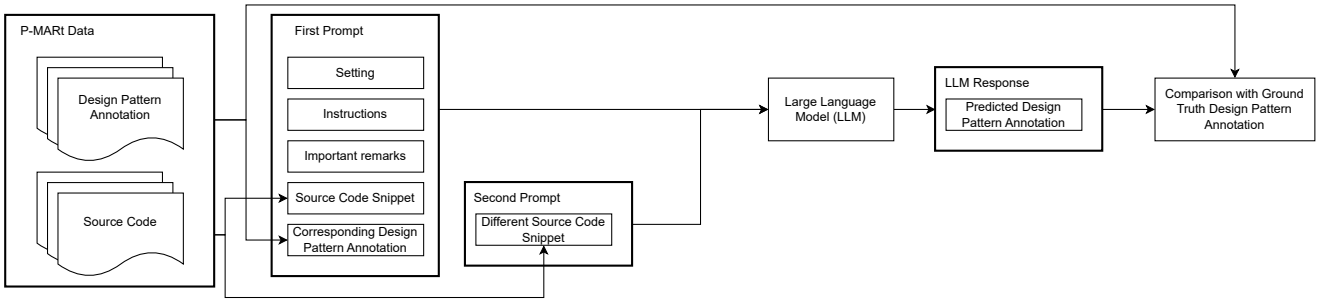

Figure 1. Schematic Overview of the Conducted Experiment.

TABLE I. INSIGHT INTO THE DESIGN PATTERN INSTANCES.

Instance	*Project*	*Common Root Package*	*# Classes Participating in Pattern instance*	*# Classes in Root Package*
4	QuickUML 2001		17	217
65	JUnit v3.7	junit	39	94
75	JHotDraw v5.1	CH.ifa.draw	35	155
98	MapperXML v1.9.7	com.taursys	29	234
129	PMD v1.8	net.sourceforge.pmd.ast	3	108
143	Software architecture design patterns in Java	src.COMPOSITE	5	5

Each prompt is presented to the LLM, and the generated output, which is expected to be in a valid XML format, is saved. The output is stored alongside the respective example and the design pattern expected to be found, ensuring traceability and alignment for subsequent analysis.

4) Post Processing: We analyzed the responses provided by the LLM and systematically extracted relevant XML annotations whenever they were available as the output of the model. In cases where the LLM did not provide any annotations, it responded with a clear answer thast no design pattern instance has been found. This allowed us to differentiate between cases where design patterns were explicitly recognized and cases where their absence was confirmed based on the LLM's output.

V. RESULTS

All prompts, LLM-generated responses, and the ground truth are available[1]. Table II presents a confusion matrix (left hand side) illustrating the roles associated with the *Composite* design pattern. Each column of the matrix corresponds to a distinct role, as well as to the additional categories *No Role* and *Hallucinated Class*. The rows follow a similar structure, with the key difference being the use of *Hallucinated Role* in place of *Hallucinated Class*. Within each cell, the matrix reports the total number of occurrences in which the LLM predicted a particular role (column) given a specific ground-truth role (row). For example, the cell at the intersection of the *Leaf* row and the *Composite* column indicates that, on four separate occasions, a class that was actually labeled as *Leaf* was misclassified by the LLM as *Composite*.

The right hand side of the table display the total number of classifications made for each role and the corresponding Precision with and without the hallucinated predictions of the LLM. Likewise, the bottom two rows (twice) of the table show the total number of ground-truth occurrences of each role, along with the corresponding Recall. The cells at the intersection of these totals represents the overall number.

Hallucination, a well-documented behavior in LLMs, involves blending prompt elements or generating information unsupported by any source. Our experiments also exhibited this phenomenon. In particular, we observed the model inventing classes not present in the code snippets—an occurrence we term hallucinated classes. Beyond that, we noted a second form of hallucination: the creation of new roles not found in any of the ground-truth examples. Both forms of hallucination are relatively straightforward to detect, as we have direct access to the complete set of valid labels and the classes embedded in the source code snippets. This comprehensive dataset enables a thorough verification process: we can systematically compare the model's predictions against the known ground-truth labels and source code elements. By doing so, it becomes evident when the model invents new classes that never appear in the provided snippets, or assigns roles not included in any of the reference examples.

We also report the results for individual prompts in Table IV and Table V. All roles—except for *No Role*— combined are the positive prediction class. We define the standard confusion matrix terms as follows: True Positive (TP): The model correctly identifies that a given source code element serves a specific role in a design pattern instance. True Negative (TN): The model correctly identifies that a given source code element does not participate in any role of the design pattern instance. False Positive (FP): The model incorrectly classifies a source code element as fulfilling a particular role, even though it does not. False Negative (FN): The model incorrectly concludes that a source code element does not fulfill any role when, in fact,

[1]https://github.com/schindlerc/LLM-Based-Design-Pattern-Detection

TABLE II. CONFUSION MATRIX CHATGPT 3.5.

							With Hallucination		Without Hallucination	
	Hallucinated Class	Client	Component	Composite	Leaf	No Role	Classification Overall	Precision	Classification Overall	Precision
Hallucinated Role	0	0	1	0	0	3	4	-	-	-
Client	38	5	1	0	2	24	70	.071	32	.156
Component	0	0	8	1	0	1	10	.800	10	.800
Composite	1	0	0	4	12	10	27	.148	26	.154
Leaf	13	0	0	4	17	32	66	.258	53	.321
No Role	0	23	5	16	185	1064	1293	.823	1293	.823
Truth overall	52	28	15	25	216	1134	1470			
Recall	-	.179	.533	.160	.008	.938				
Truth overall	-	28	14	25	216	1131			1414	
Recall	-	.179	.571	.160	.008	.941				

TABLE III. CONFUSION MATRIX GPT4.

	No Role	Client	Component	Composite	Leaf	Classification Overall	Precision
Client	13	9	1	3	8	34	.265
Component	1	0	10	0	0	11	.910
Composite	5	0	0	12	12	29	.414
Leaf	34	2	0	1	45	82	.549
No Role	1079	17	4	11	150	1261	.856
Truth overall	1132	28	15	27	215	1417	
Recall	.953	.333	.833	.522	.209		

it does have a defined role according to the ground truth.

That implies that the Precision, Recall, and F1 metrics reported focus solely on assessing the model's ability to correctly identify the roles within the design pattern instances. All classes from the code snippets that are not part of the design pattern instance serve as the negative (prediction) class, ensuring that the evaluation focuses on the roles of the design pattern. In evaluating the performance of classification models, Precision, Recall, and F1 Score are key metrics that provide insight into the model's accuracy in distinguishing between the classes. These metrics are particularly important in imbalanced datasets where a models overall accuracy might be misleading.

In the context of FP, a misalignment between the intended usage of a design pattern and its implementation can negatively impact software engineering projects, resulting in bugs, unexpected behavior, and error propagation throughout the system.

Table IV shows the different prediction runs with ChatGPT 3.5 as a row each, with the following information. *Run* is a running number to refer to the individual prediction runs with *TP, TN, FP*, and *FN* predictions, alongside the bespoken metrics. On the right side we reported the same metrics after removed the hallucination introduced by the LLM. Hallucination occurred in three runs (2, 3, and 4). Removal of the hallucination lead to increased Precision, F1 Score and Accuracy in those runs. Four runs (i.e., 11, 12, 13, and 14) did not detect a design pattern annotation but instead responded with the no design pattern found response.

Table V lists the prediction runs with ChatGPT 4, the noticeable difference to Table IV is, that no hallucination occurred. In a consequence, for each run their is only one set of metrics to be reported. In addition ChatGPT 4 has been better in 7 out of 14 runs for Precision and Accuracy, in 8 cases for F1 Score and in 9 cases for Recall.

Table VI provides additional details on each of the 14 runs and includes the corresponding design pattern instance IDs for both the example and the target, referencing the IDs in Table I. In this context, the example is the annotated instance, while the target is the instance for which only source code was provided via prompts. The subsequent columns show the number of role annotations in the ground truth (for both Example and Target) and those predicted by the LLMs ChatGPT 3.5 and 4.

We noticed that in all cases where the LLMs failed to identify a design pattern annotation, the same example (143) has been used. A notable aspect of this example is that the source code snippet includes only those classes that participate in the design pattern (see Table I), with no additional classes. It appears that providing the snippet within a broader implementation context helps the LLM better distinguish which classes are relevant to a design pattern instance.

Examining the confusion matrix shows that Precision and Recall vary considerably across different roles. The highest scores for both Precision and Recall occur for the role *Component* (keeping the *No Role* label out of scope), likely because each design pattern instance has exactly one class labeled with this role, and every successful prediction also identified exactly one such class. By contrast, performance for the other roles is poorer. As shown in Table VI, the number of classes labeled differs widely among examples. In the case of *Client*, for instance, some instances lack this role entirely.

To address RQ1, we designed two prompts for the task. The first prompt provides essential context: it describes the setting for the LLM, outlines its assumed skill set, and provides explicit instructions for problem-solving and important remarks. It also includes an example of an annotated design pattern and a corresponding code snippet. The second prompt supplies the target code snippet for which the model should output

TABLE IV. RESULTS FOR EACH PREDICTION RUN WITH CHATGPT 3.5

	With Hallucination								Without Hallucination							
Run	TP	FP	FN	TN	Precision (P)	Recall (R)	F1	Accuracy (A)	TP	FP	FN	TN	P	R	F1	A
1	4	5	36	53	.444	.100	.163	.582	4	5	36	53	.444	.100	.163	.582
2	12	17	23	117	.414	.343	.375	.763	12	**6**	23	117	**.667**	.343	**.453**	**.816**
3	3	77	37	39	.037	.075	.050	.269	3	**33**	37	39	**.083**	.075	**.079**	**.375**
4	2	2	38	54	.500	.050	.091	.583	2	**1**	38	54	**.667**	.050	**.093**	**.589**
5	1	29	16	171	.033	.059	.043	.793	1	29	16	171	.033	.059	.043	.793
6	1	3	4	0	.250	.200	.222	.125	1	3	4	0	.250	.200	.222	.125
7	3	4	2	0	.429	.600	.500	.333	3	4	2	0	.429	.600	.500	.333
8	3	1	2	0	.750	.600	.667	.500	3	1	2	0	.750	.600	.667	.500
9	3	1	2	0	.750	.600	.667	.500	3	1	2	0	.750	.600	.667	.500
10	2	3	3	0	.400	.400	.400	.250	2	3	3	0	.400	.400	.400	.250
11	0	0	17	200	0	0	0	.922	0	0	17	200	0	0	0	.922
12	0	0	3	105	0	0	0	.972	0	0	3	105	0	0	0	.972
13	0	0	29	205	0	0	0	.876	0	0	29	205	0	0	0	.876
14	0	0	35	120	0	0	0	.774	0	0	35	120	0	0	0	.774

TABLE V. RESULTS FOR EACH PREDICTION RUN WITH CHATGPT4

Run	TP	FP	FN	TN	Precision	Recall	F1	Accuracy
1	6	3	34	54	.667	.150	.245	.619
2	21	32	14	98	.396	.600	.477	.721
3	8	4	32	53	.667	.200	.308	.629
4	0	0	39	55	0	0	0	.585
5	0	0	17	200	0	0	0	.922
6	2	1	3	0	.667	.400	.500	.333
7	3	1	2	0	.750	.600	.667	.500
8	3	1	2	0	.750	.600	.667	.500
9	3	1	2	0	.750	.600	.667	.500
10	2	1	3	0	.667	.400	.500	.333
11	11	0	6	200	1.000	.647	.786	.972
12	0	0	3	105	0	0	0	.972
13	0	11	29	203	0	0	0	.835
14	17	15	18	111	.531	.486	.507	.795

the design pattern annotation. By presenting the context and an example before showing the target snippet, we ensure the LLM is both well-prepared and guided by a concrete sample.

To address RQ2, we analyzed the experimental results, which revealed varying levels of quality depending on the roles within the design pattern. Furthermore ChaGPT 4 performed better than ChatGPT 3.5, by (i) finding more annotations and (ii) having more correct annotations. Additionally, we observed substantial variability in quality across different runs, influenced by the pairing of the example code with unseen target code and the expected annotations.

In RQ3, we investigated the challenges and limitations of using an LLM-based approach for detecting design pattern instances. Our findings demonstrate that LLMs, such as ChatGPT 3.5 and 4 in our case, are capable of processing prompts containing code snippets spanning hundreds of Java classes and accurately retrieving classes along with their correct roles. However, this approach faces several limitations.

One significant constraint is the token limit of current LLMs, which restricts scalability. In our experiments, we could only include a single example of the design pattern for a given prediction in such detail, as multiple examples would exceed the context window. Additionally, some pairings of examples and target code snippets had to be excluded because their combined token count exceeded the allowable limit.

Another challenge lies in the heterogeneity of the design pattern instances, as they vary in the number of roles and the specific roles annotated in the examples, adding complexity to the task and potentially affecting the model's performance.

If the role expected to be predicted by the LLM is not present in the provided example, it becomes an unseen role, making it challenging for the models to annotate accurately.

To address the maxing out of the token size limitation of current LLMs, an effective strategy could involve reducing the size of the code snippets provided in each prompt. By pruning the code snippets—such as focusing more on the relevant sections or removing parts that are less critical for identifying the design pattern. Presenting multiple examples could help the model better generalize across varying instances of the design pattern and potentially improve its prediction accuracy.

For instance, pruning could involve removing sections of code unrelated to the design pattern being analyzed or adjusting the balance between classes directly involved in the design pattern instances and those present in the source project but not participating in the design pattern instance. This approach would allow for a richer variety of training examples without exceeding the token limit, thereby balancing the trade-off between example diversity and code snippet size.

VI. CONCLUSION AND FUTURE WORK

We have demonstrated an approach for identifying design pattern instances in large codebases comprising over 200 classes presented simultaneously, using only a single example of the design pattern. This approach operates directly on the source code without requiring modifications like abstraction or feature extraction. Our results show that the model can partially retrieve design patterns with varying levels of completeness and accuracy.

However, as discussed in the results section of the paper, the approach is not perfect, as it handles different roles with varying degrees of success. Moreover, the experiment we conducted was quite limited: it tested only one design pattern with two LLMs. Consequently, results for other design patterns may differ from those reported here, particularly because certain patterns could be either easier or more difficult to detect.

TABLE VI. COMPARISON OF THE AMOUNT OF ROLES PER EXAMPLE AND RUN.

Run	Design Pattern Instance ID		Composite				Component				Leaf				Client			
	Example (E)	Target (T)	E	T	GPT4	GPT3.5	E	T	4	3.5	E	T	4	3.5	E	T	4	3.5
1	4	65	1	2	1	1	1	1	1	1	14	37	6	6	1	0	1	1
2	65	75	2	5	4	2	1	1	1	1	37	21	29	22	0	8	19	0
3	75	65	5	2	1	10	1	1	1	1	21	37	7	17	8	0	3	52
4	143	129	1	1	-	1	1	1	-	1	2	1	-	1	1	0	-	1
5	143	98	1	1	-	5	1	1	-	1	2	22	-	15	1	5	-	9
6	129	143	1	1	1	2	1	1	1	1	1	2	1	1	0	1	0	0
7	98	143	1	1	1	1	1	1	1	1	22	2	1	1	5	1	1	4
8	65	143	2	1	1	1	1	1	1	1	37	2	1	1	0	1	1	1
9	4	143	1	1	1	1	1	1	1	1	14	2	1	1	1	1	1	1
10	75	143	5	1	1	2	1	1	1	1	21	2	1	1	8	1	0	1
11	143	4	1	1	1	-	1	1	1	-	2	14	8	-	1	1	1	-
12	143	129	1	1	-	-	1	1	-	-	2	14	-	-	1	0	-	-
13	143	98	1	1	5	-	1	1	1	-	2	22	2	-	1	5	3	-
14	143	75	1	5	5	-	1	1	1	-	2	21	25	-	1	8	1	-

However, the presented approach can be consistently applied to other design patterns without modification. In addition, the experiment was conducted as a one-shot test, providing only a single example. It would therefore be valuable to investigate how providing multiple examples might influence the model's predictive performance.

Potential directions for future work in this domain may include fine-tuning LLMs before conducting similar experiments, refining prompt design, and adopting hybrid approaches that integrate LLMs with logical reasoning or static and dynamic analysis.

REFERENCES

[1] O. Kaczor, Y.-G. Guéhéneuc, and S. Hamel, "Identification of design motifs with pattern matching algorithms," *Information and Software Technology*, vol. 52, no. 2, pp. 152–168, 2010.

[2] H. Yarahmadi and S. M. H. Hasheminejad, "Design pattern detection approaches: A systematic review of the literature," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5789–5846, Dec. 2020.

[3] F. A. Fontana, A. Caracciolo, and M. Zanoni, "Dpb: A benchmark for design pattern detection tools," in *2012 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 235–244.

[4] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE transactions on software engineering*, vol. 32, no. 11, pp. 896–909, 2006.

[5] B. B. Mayvan and A. Rasoolzadegan, "Design pattern detection based on the graph theory," *Knowledge-Based Systems*, vol. 120, pp. 211–225, 2017.

[6] R. Singh Rao and M. Gupta, "Design pattern detection by greedy algorithm using inexact graph matching," *International Journal Of Engineering And Computer Science*, vol. 2, no. 10, pp. 3658–3664, 2013.

[7] N. Bozorgvar, A. Rasoolzadegan, and A. Harati, "Probabilistic detection of gof design patterns," *The Journal of Supercomputing*, vol. 79, no. 2, pp. 1654–1682, 2023.

[8] R. Barbudo, A. Ramírez, F. Servant, and J. R. Romero, "Geml: A grammar-based evolutionary machine learning approach for design-pattern detection," *Journal of Systems and Software*, vol. 175, p. 110 919, 2021.

[9] G. Rasool and P. Mäder, "Flexible design pattern detection based on feature types," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, IEEE, 2011, pp. 243–252.

[10] H. Thaller, L. Linsbauer, and A. Egyed, "Feature maps: A comprehensible software representation for design pattern detection," in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*, IEEE, 2019, pp. 207–217.

[11] N. Nazar, A. Aleti, and Y. Zheng, "Feature-based software design pattern detection," *Journal of Systems and Software*, vol. 185, p. 111 179, 2022.

[12] A. Nacef, S. Bahroun, A. Khalfallah, and S. B. Ahmed, "Features and supervised machine learning-based method for singleton design pattern variants detection," in *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2023)*, 2023, pp. 226–237.

[13] R. Mzid, I. Rezgui, and T. Ziadi, "Attention-based method for design pattern detection," in *European Conference on Software Architecture*, Springer, 2024, pp. 86–101.

[14] S. Komolov, G. Dlamini, S. Megha, and M. Mazzara, "Towards predicting architectural design patterns: A machine learning approach," *Computers*, vol. 11, no. 10, p. 151, 2022.

[15] M. Mukelabai, K. Hermann, T. Berger, and J.-P. Steghöfer, "FeatRacer: Locating features through assisted traceability," *IEEE Trans. Softw. Eng.*, vol. 49, no. 12, pp. 5060–5083, Oct. 2023.

[16] L. Fan *et al.*, "A bibliometric review of large language models research from 2017 to 2023," *ArXiv*, 2023.

[17] W. Chen, "Large language models are few(1)-shot table reasoners," *ArXiv*, 2022.

[18] W. X. Zhao *et al.*, "A survey of large language models," *ArXiv*, 2023.

[19] G. Yenduri *et al.*, "Generative pre-trained transformer: A comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions," *ArXiv*, vol. abs/2305.10435, 2023.

[20] C. A. G. Cano, V. S. Castillo, and T. A. C. Gallego, "Unveiling the thematic landscape of generative pre-trained transformer (gpt) through bibliometric analysis," *Metaverse Basic and Applied Research*, pp. 1–8, 2023.

[21] A. S. Gill, "Chat generative pretrained transformer: Extinction of the designer or rise of an augmented designer," in *Int. Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2023, pp. 1–5.

[22] Q. Zhu and J. Luo, "Generative pre-trained transformer for design concept generation: An exploration," *Proceedings of the Design Society*, vol. 2, pp. 1825–1834, 2021.

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