Design Patterns and LLMs

Francisco Pereira, Gonçalo Borges, Tiago Mendes
Masters in Computer Science - Software Engineering
Faculdade de Ciências e Tecnologias da Universidade de Coimbra
Coimbra, Portugal
Email: {uc2022217071, uc2022215524, uc2022216857}@student.uc.pt

***Abstract*—The architectural quality of code generated by Large Language Models (LLMs) is a critical but largely unexplored dimension. While LLMs produce functional code, their autonomous adoption of established "Gang of Four" (GoF) software design patterns is unknown. This paper presents the first empirical study to quantify the spontaneous emergence of the 23 GoF patterns in LLM-generated Java code. We analyzed a 600-file corpus generated by six different LLMs from 100 neutral prompts, using a validated analysis tool.**

Our findings show a strong bias towards simplicity. Only 18.2% of files contained any GoF pattern, and only 10 of the 23 patterns were ever detected. A heavy concentration (nearly 78%) was found on just three patterns: Builder, State, and Command. Notably, 13 fundamental patterns, including Observer and Composite, were completely absent. We conclude that current LLMs do not make sophisticated, autonomous design decisions, instead reflecting common statistical patterns from their training data, which raises concerns about the maintainability of the code they produce for complex systems.

***Index Terms*—Large Language Models (LLMs) Design Patterns Gang of Four (GoF) Code Generation Code Analysis Software Engineering Empirical Study Java**

I. Introduction

Large Language Models (LLMs) have recently become powerful tools for automated code generation, assisting developers in solving programming tasks in a wide range of domains. Modern models are capable of producing syntactically correct and functionally accurate programs from natural-language prompts. While the maintainability [**?**], correctness [**?**] [**?**] [**?**] and efficiency [**?**] [**?**] of LLM-generated code, as well as LLM's ability to detect design patterns [**?**] have been widely studied, far less is known about the software design quality and architectural characteristics of such code.

In traditional software engineering, *design patterns*, as introduced by Gamma *et al.* [**?**], represent well-established solutions to recurring design problems. These patterns play a fundamental role in object-oriented programming (OOP) by promoting reusability, maintainability, and scalability. Understanding whether and how LLMs implicitly employ these patterns can reveal valuable insights into their internal reasoning tendencies and the extent to which they have internalized human design practices.

To the best of our knowledge, no prior study has systematically investigated the occurrence of GoF (Gang of Four) design patterns in code automatically generated by LLMs. Our comprehensive literature search across major digital libraries (IEEE Xplore, ACM Digital Library, SpringerLink, and arXiv) found no publications analyzing the types or frequencies of design patterns emerging from LLM-generated code.

This paper presents an empirical study aimed at identifying which of the 23 GoF design patterns are most commonly introduced by LLMs when generating Java programs without any architectural guidance. We employ a two-step methodology: (i) generating code from 100 neutral Java problem prompts using multiple LLMs, and (ii) analyzing the resulting code with the *Design Pattern Finder* tool [**?**] to detect implemented patterns. The results reveal interesting behavioral trends across models and provide the first quantitative evidence of pattern-level design emergence in LLM-generated software.

The main contributions of this study are as follows:

- We propose a systematic methodology for detecting design patterns in LLM-generated Java code.
- We validate the reliability of the DesignPatternFinder tool.
- We present and discuss empirical findings on which design patterns are most frequently generated across different LLMs.

II. Background

A. Introduction to "Gang of Four" (GoF) Design Patterns

The "Gang of Four" (GoF) design patterns represent a foundational collection of 23 reusable solutions to common, recurring problems within object-oriented software design. These patterns were formally cataloged in the seminal 1994 book, Design Patterns: Elements of Reusable Object-Oriented Software [**?**].

These patterns are not finished designs or concrete code libraries; rather, they are language-agnostic templates that describe how to solve a specific design problem in a flexible and maintainable way.

The 23 patterns are broadly classified into three distinct categories based on their purpose:

- **Creational Patterns:** These patterns abstract the instantiation process, providing mechanisms for creating objects in a manner suitable for the situation (e.g., Singleton, Factory Method, Abstract Factory).
- **Structural Patterns:** These patterns focus on class and object composition, describing how to assemble objects and classes into larger, more complex structures (e.g., Adapter, Decorator, Facade).
- **Behavioral Patterns:** These patterns concern the responsibilities of objects and the communication patterns between them (e.g., Strategy, Observer, Command).

III. Methodology

This section describes the experimental process adopted to identify design patterns in Java code generated by large language models (LLMs). The methodology consists of two main steps: (1) code generation using multiple LLMs, and (2) code analysis using the *Design Pattern Finder* tool. We also include validation procedures to ensure the reliability of the tool and the suitability of the experimental setup.

A. Overview

Figure 1 (Conceptual Pipeline) illustrates the overall workflow. A set of 100 natural-language prompts was used to request Java implementations of simple programming problems from several LLMs. The generated code was then collected, standardized, and analyzed using the *Design Pattern Finder* tool to detect occurrences of the 23 GoF design patterns. The detected patterns were aggregated and statistically summarized to derive comparative metrics across models.

B. Code Generation

1) Prompt Selection: A total of 100 short, task-oriented Java prompts were prepared. These prompts describe basic programming problems (e.g., "Write a Java event logger for a GUI application with filters" or "Write Java code for a social media feed that supports posts, likes, and comments") without imposing any architectural or design constraints. This neutral formulation ensures that the resulting designs reflect each model's intrinsic problem-solving behavior, rather than being influenced by explicit instructions about structure or patterns.

2) Model Selection: To ensure a representative and diverse evaluation, we selected some of the most popular, freely available large language models (LLMs) hosted on Hugging Face [**?**]. The chosen models come from different organizations, feature varying numbers of parameters, and include distinct

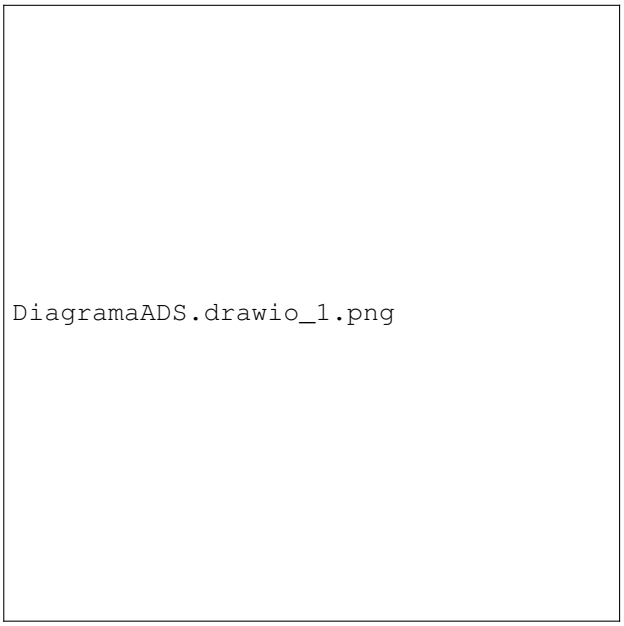

Fig. 1. Conceptual Pipeline.

fine-tuning strategies. This diversity helps capture a wide range of coding behaviors and capabilities. Specifically, we used the following models:

- Gemma 2 2B IT
- GPT-OSS 120B
- Llama 3.1 8B Instruct
- Llama 3.2 1B Instruct
- Qwen 2.5 7B Instruct
- Qwen 2.5 Coder 32B

C. Code Analysis

The generated Java files were analyzed using *Design Pattern Finder*, an analysis tool capable of detecting the 23 GoF design patterns.

For each analyzed file, the tool produces a report listing all detected patterns, their occurrences, and corresponding classes. These outputs were aggregated to compute:

- The total frequency of each pattern per model,
- The distribution of patterns by category (Creational, Structural, Behavioral),
- The proportion of files containing at least one identifiable pattern.

D. Tool Selection and Validation

The primary objective of our research is to analyze design patterns introduced or frequently used by Large Language Models (LLMs) in software development. Manually identifying these patterns across a large corpus of code is infeasible,

time-consuming, and prone to human error. Therefore, the use of an automated analysis tool is essential.

For this study, we selected the *Design Pattern Finder* (DPF) utility. The tool is designed to parse source code and identify implementations of common 'Gang of Four' (GoF) design patterns.

1) Validation Methodology: Before applying DPF to our primary (LLM-generated) dataset, it was crucial to verify its accuracy and reliability. We conducted a validation test using a curated ground truth dataset. This dataset was constructed by combining two well-documented, public GitHub repositories dedicated to GoF design pattern implementations:

- `girirajvyas/gof-design-patterns [?]`
- `RameshMF/gof-java-design-patterns [?]`

These repositories provided a total of 463 distinct Java files, each representing a clear and known implementation of one of 23 different GoF design patterns (e.g., Singleton, Factory, Adapter). We then executed the Design Pattern Finder tool on this entire dataset and compared its output against the known ground truth.

2) Validation Results: The performance of the DPF tool was evaluated using standard information retrieval metrics: Precision, Recall, and F1-Score. The results, summarized in Table **??**, demonstrate perfect accuracy.

TABLE I
OVERALL VALIDATION METRICS FOR DESIGN PATTERN FINDER

Metric	Value
Total Ground Truth Files	463
Files Detected by Tool	463
Correct Detections	463
Incorrect Detections	0
Missed Files	0
Precision	100.00%
Recall	100.00%
F1 Score	100.00%

The tool correctly identified all 463 pattern implementations without a single false positive or false negative. This perfect score extended across all 23 pattern types included in the test set. Table **??** shows a representative sample of these per-pattern results, all of which achieved **100%** across all metrics.

3) Conclusion of Validation: The **100%** accuracy, precision, and recall demonstrated in our validation test provide high confidence in the *Design Pattern Finder* tool. Based on this perfect performance on a robust ground truth dataset, we deemed the tool reliable and suitable for use in the primary analysis of our research.

TABLE II
PER-PATTERN VALIDATION METRICS (REPRESENTATIVE SAMPLE)

Pattern	Total Files	Correct	Precision	Recall
Adapter	16	16	100.00%	100.00%
Bridge	42	42	100.00%	100.00%
Builder	14	14	100.00%	100.00%
Composite	23	23	100.00%	100.00%
Decorator	24	24	100.00%	100.00%
Facade	24	24	100.00%	100.00%
Factory	101	101	100.00%	100.00%
Singleton	16	16	100.00%	100.00%
Strategy	15	15	100.00%	100.00%

Note: All 23 tested patterns (including Chain, Command, Flyweight, Observer, etc.) reported 100% on all metrics.

IV. RESULTS AND EVALUATION

The analysis was performed on a corpus of 600 generated Java files, with 100 files produced by each of the six selected LLMs. *Design Pattern Finder* was used to detect occurrences of the 23 GoF design patterns.

A. Overall Model Performance

Across the entire 600-file dataset, **109** files (**18.2%**) were found to contain at least one identifiable GoF design pattern. The remaining **491** files (**81.8%**) did not contain any detectable patterns, suggesting a tendency for LLMs to produce simpler, more straightforward implementations when not explicitly prompted for architectural complexity.

There was significant variance in pattern generation rates between the models, as detailed in Table **??**. The *GPT-OSS 120B* model had the highest detection rate, with **41%** of its files containing patterns, while *Llama 3.2 1B Instruct* and *Gemma 2 2B IT* had the lowest at **11%**. This **30-point** variance highlights that model choice is a significant factor in the architectural characteristics of generated code.

TABLE III
PER-MODEL PATTERN DETECTION SUMMARY

Model	Files Analyzed	Files w/ Patterns	Pct w/ Patterns	Files w/ Multi-Patterns	Unique Patterns
Gemma 2 IT	100	11	11.0%	1	6
GPT-OSS	100	41	41.0%	10	8
Llama 3.1 Instruct	100	12	12.0%	2	5
Llama 3.2 Instruct	100	11	11.0%	1	5
Qwen 2.5 Instruct	100	19	19.0%	2	7
Qwen 2.5 Coder Instruct	100	15	15.0%	5	7
Total	**600**	**109**	**18.2%**	**21**	**10**

B. Pattern Frequency and Distribution

Our analysis detected only **10** distinct patterns across all 600 generated files, indicating a strong bias in the solutions produced by LLMs. The distribution is highly concentrated on

the top three patterns: **Builder**, **State**, and **Command**, representing nearly **74%** of all patterns generated. The remaining patterns were much less frequent: **Factory** (**11**) and **Adapter** (**10**) formed a secondary group, while **Iterator** (6), **Template** (4), **Facade** (1), **Singleton** (1), and **Strategy** (1) were almost absent. Additionally, **13** of the 23 GoF patterns, including *Observer*, *Composite*, and *Decorator*, were not detected at all.

Table **??** shows the distribution by model. The *GPT-OSS 120B* model stood out, generating **51** pattern instances and being the only one to produce **Facade**. The Qwen models formed a middle tier with **21** instances each, while Llama 3.2 and Gemma models had the lowest rate (**12**).

C. Code Generation Volume (Lines of Code)

In addition to design patterns, we analyzed the volume of code generated by each model, measured in Lines of Code (LOC). Table **??** presents the statistical distribution of LOC for the 100 files generated by each model. This analysis helps contextualize whether models that produce more patterns also produce more code overall, and reveals interesting differences in code verbosity and complexity across models.

Across all 600 files, the corpus totaled **58,006 lines of code**, with an average of **96.68 lines per file**. However, there is substantial variation between models. The *GPT-OSS 120B* model generated the most code overall (**15,209 LOC**, mean **152.09** lines per file), while *Gemma 2 2B IT* produced the least (**6,449 LOC**, mean **64.49** lines per file), a difference of more than **2.3×** in average file length.

Looking at the median values, *GPT-OSS 120B* has the longest median LOC at **146 lines**, while *Gemma 2 2B IT* has the shortest at **61 lines**. This pattern holds when examining the interquartile ranges: larger models like *GPT-OSS 120B* show both higher medians and wider spreads, indicating more variable and generally more verbose outputs. In contrast, smaller models like *Gemma 2 2B IT* and *Qwen 2.5 7B Instruct* produce more compact and consistent code.

Notably, *GPT-OSS 120B* also exhibits the highest variability (standard deviation: **51.46**), while *Qwen 2.5 7B Instruct* is the most consistent (standard deviation: **30.01**). This suggests that larger models may adapt their output length more dynamically based on problem complexity, whereas smaller models maintain a more uniform coding style.

Figure **??** visually represents this distribution, clearly showing the substantial differences in code generation volume across models.

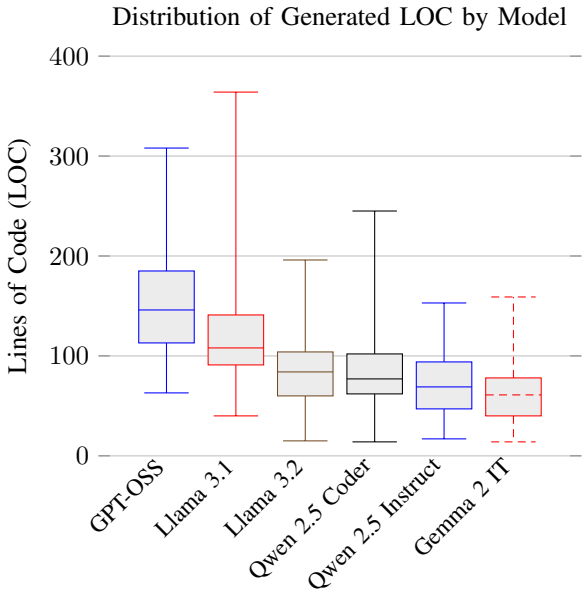

Fig. 2. Box plot of generated Lines of Code (LOC) distribution for each model, showing min, quartiles, median, and max.

The correlation between model size and code length is evident: *GPT-OSS 120B*, the largest model tested, consistently produces the longest code, while the smallest models (*Gemma 2 2B IT* and *Llama 3.2 1B Instruct*) produce the most compact implementations. Interestingly, when comparing this LOC data with the pattern detection rates from Table III, we observe that *GPT-OSS 120B* leads in both metrics (**41%** pattern detection rate and **152.09** mean LOC), suggesting that longer code may provide more opportunities for pattern implementations or that the model's tendency toward more elaborate solutions manifests in both dimensions.

D. Pattern Category Analysis

When grouping the detected patterns into their GoF categories, we observe a slight skew towards **Behavioral** (**48.9%**) and **Creational** (**42.7%**) patterns. **Structural** patterns were significantly less common, accounting for only **8.4%** of all detected instances.

Table **??** shows this breakdown by model. This suggests that LLMs are adept at generating patterns for object creation and communication, while complex structural compositions are rare. The high number for the **Creational** category is driven almost entirely by the **Builder** pattern, and the high **Behavioral** count is driven by **State** and **Command**.

V. Discussion

The results of our analysis provide several insights into the current state of LLM-generated code. We interpret these

TABLE IV
DETECTED PATTERN FREQUENCY BY MODEL

	Model						
Pattern	Gemma 2 2B IT	GPT-OSS 120B	Llama 3.1 8B Instruct	Llama 3.2 1B Instruct	Qwen 2.5 7B Instruct	Qwen 2.5 Coder Instruct	**Total**
Adapter	1	1	2	2	2	2	**10**
Builder	2	17	4	3	9	9	**44**
Command	3	9	4	2	2	4	**24**
Facade	0	1	0	0	0	0	**1**
Factory	2	3	2	2	1	1	**11**
Iterator	0	4	0	0	0	2	**6**
Singleton	0	0	0	0	1	0	**1**
State	3	14	2	3	5	2	**29**
Strategy	0	0	0	0	1	0	**1**
Template	1	2	0	0	0	1	**4**
Total	**12**	**51**	**14**	**12**	**21**	**21**	**131**

Note: The total number of patterns (131) is higher than the number of files with patterns (109) because 21 files contained multiple patterns.

TABLE V
STATISTICAL DISTRIBUTION OF GENERATED LINES OF CODE (LOC)

Model	Min	Q1 (25%)	Median (50%)	Q3 (75%)	Max
Gemma 2 2B IT	14	40	61	78	159
GPT-OSS 120B	63	113	146	185	308
Llama 3.1 8B	40	91	108	141	364
Llama 3.2 1B	15	60	84	104	196
Qwen 2.5 7B	17	47	69	94	153
Qwen 2.5 Coder	14	62	77	102	245

TABLE VI
PATTERN CATEGORY DISTRIBUTION BY MODEL

	Model						
Category	Gemma	GPT-OSS	Llama 3.1	Llama 3.2	Qwen Inst	Qwen Coder	**Total**
Creational	4	20	6	5	11	10	**56**
Structural	1	2	2	2	2	2	**11**
Behavioral	7	29	6	5	8	9	**64**
Total	**12**	**51**	**14**	**12**	**21**	**21**	**131**

findings and their implications for software engineering.

First, the low overall pattern detection rate (**18.2%**) strongly implies that LLMs favor simpler, more direct implementations over architecturally complex solutions. In **81.8%** of cases, the models generated code with no identifiable GoF patterns, opting for straightforward procedural or class-based logic. This may be desirable for simple tasks but raises concerns about scalability and maintainability for more complex, system-level prompts.

Second, a clear pattern bias exists. The heavy favoritism shown towards the **Builder** (**44** occurrences), **State** (**29**), and **Command** (**24**) patterns, coupled with the complete absence of **13** other GoF patterns (such as Observer, Composite, or Decorator), suggests that the models' training data or internal reasoning is skewed. The Builder pattern, for instance, may be common due to its frequent use in data classes and APIs (e.g., *StringBuilder*), making it a high-frequency token sequence in training corpora.

Third, model architecture and training matter. The **30-point** variance in detection rates between *GPT-OSS 120B* (**41%**) and *3.2 1B Instruct / Gemma 2 2B IT* (**11%**) demonstrates that not all models are equal in their ability or tendency to reproduce structured designs. This suggests that the scale of the model, its training data, and its fine-tuning process all have a measurable impact on the design quality of the code it produces.

Finally, the impact of training data is further highlighted by the missing patterns. The **13** undetected patterns (including common ones like **Observer**, **Composite**, and **Decorator**) are not necessarily more complex, but they may be less common in the open-source codebases used for training.

These findings suggest that while LLMs are capable of generating functional code, they cannot yet be relied upon to make sophisticated, high-level design decisions autonomously. The code they produce reflects the most common patterns found in public data, not necessarily the most appropriate design for a given problem.

VI. CONCLUSION AND FUTURE WORK

A. Conclusion

This study presented a pioneering empirical analysis of the spontaneous emergence of the 23 "Gang of Four" (GoF) design

patterns in Java code generated by six large language models (LLMs). Through a systematic methodology involving 100 neutral prompts and the rigorous validation of the *Design Pattern Finder* (DPF) static analysis tool, we obtained quantitative insights into the design tendencies of current LLMs.

The primary conclusions are clear and significant. The most prominent finding is that LLMs, when not explicitly instructed, tend to avoid architectural complexity. The vast majority of the generated code, **81.8%**, contained no identifiable GoF patterns, favoring straightforward procedural implementations. This raises questions about the scalability and maintainability of LLM-generated code for more complex problems.

Furthermore, the diversity of patterns was extremely low. Only **10** of the 23 GoF patterns were detected across all 600 files. The distribution was heavily concentrated, with the **Builder**, **State**, and **Command** patterns alone accounting for nearly **78%** of all detected instances. Surprisingly, **13** GoF patterns—including fundamental ones like **Observer**, **Composite**, and **Decorator**—were not detected at all.

The choice of LLM also has a measurable impact on design. The *GPT-OSS 120B* model, the largest one tested, exhibited the highest pattern detection rate at **41%** and also produced the longest code, with a mean of **152.09 LOC**. In contrast, smaller models like *Llama 3.2 1B* and *Gemma 2 2B IT* had the lowest rates at **11%**. These results suggest that current LLMs are not yet reliable for making sophisticated, autonomous software design decisions. Their behavior appears to reflect the statistically most common patterns found in their training data, rather than a deep understanding of design problems and the application of the most appropriate solution.

B. Future Work

The results of this study open several promising paths for future research, which are essential for understanding and improving the architectural quality of AI-generated code. This study focused on neutral prompts; a logical next step is to investigate the ability of LLMs to correctly implement design patterns, especially the **13** that were absent, when explicitly instructed to do so. Our analysis was quantitative, focusing on detection. Future research should pivot to a qualitative analysis to assess whether the detected patterns were, in fact, the most appropriate design solution for the problem proposed by the prompt.

This analysis was also limited to Java. It would be valuable to replicate this study in other object-oriented languages, such as Python or C#, to determine if the observed pattern bias is universal or specific to the Java ecosystem. The prompts used described relatively contained programming problems. Future studies should use prompts that describe more complex, larger-scale systems to investigate whether inherently more complex problems force LLMs to utilize the structural patterns that were missing in this study. Finally, a specific investigation into why common patterns like *Observer* and *Decorator* were not generated is warranted. This could involve analyzing the training corpora or fine-tuning models to verify if they can learn and apply these crucial design concepts.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley Professional, 1995.

[2] S. V, "Designpatternfinder: A tool for detecting gof design patterns in java code," https://github.com/DesignPatternFinder/DesignPatternFinder, 2020, accessed: 2024-01-01.