The Journal of Systems and Software 230 (2025) 112519

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Comparative analysis of design pattern implementation validity in LLM-based code refactoring☆

Dae-Kyoo Kim

Computer Science and Engineering, Oakland University, Rochester, MI, 48309, United States

ARTICLE INFO

Keywords:
Assessment
Code refactoring
Design patterns
Large language models (LLMs)
Metrics
Pattern implementation

ABSTRACT

Design patterns are essential in software engineering, providing proven solutions for recurring design challenges, thereby enhancing maintainability, flexibility, and reusability of code. Despite their significance, the ability of Large Language Models (LLMs) to accurately implement these patterns has not been thoroughly explored. This research introduces a novel assessment framework that combines predicate logic specifications with quantitative metrics to evaluate pattern implementation quality. Using two case studies - a Point of Sale System (POSS) and Smart Wallet System (SWS) - we assess the LLMs' capabilities in implementing design patterns including the Factory Method, Strategy, Composite, Observer, and Singleton patterns. The evaluation framework employs three metrics: Property Satisfaction Rate (PSR), Critical Property Coverage (CPC), and Pattern Implementation Quality Score (PIQS). The results demonstrate varying levels of effectiveness across the LLMs, with Claude achieving the highest average PIQS of 89.51, followed by Meta (88.98), ChatGPT (87.75), Copilot (82.69), and Gemini (71.04). These findings suggest that while LLMs show promise as refactoring tools, they are best utilized as assistive technologies rather than replacements for human developers.

1. Introduction

Design patterns represent proven solutions to common software design problems, enhancing code maintainability, flexibility, and reusability. As large language models (LLMs) become increasingly prevalent in software development, understanding their capability to correctly implement design patterns through code refactoring is crucial. This study aims to evaluate and compare the effectiveness of different LLMs in this context.

Numerous studies have extensively analyzed the impact of LLMs on various aspects of software engineering, from requirements engineering and system design to code generation and maintenance. Fan et al. (2023) surveyed applications of LLMs across the field, highlighting the potential for enhanced productivity and the challenges in ensuring output reliability. Nguyen-Duc et al. (2023) noted that while implementation receives substantial attention, areas like requirements engineering require further exploration. Marques et al. (2024) and Kim et al. (2023) found that LLMs like ChatGPT effectively improve stakeholder communication but face issues with accuracy and traceability. In code generation, Solohubov et al. (2023) and Dakhel et al. (2023) reported efficiency gains and ease in debugging with LLMs, though they also highlighted limitations in generating idiomatic practices. Ahmad et al. (2023) and Rajbhoj et al. (2024) discussed LLMs' roles in architectural design, emphasizing the benefits and challenges in human-bot collaboration. Additionally, studies on testing and quality assurance by Kirinuki and Tanno (2024) and Alshahwan et al. (2024) indicated that LLMs can achieve test coverage comparable to or better than human testers and enhance software quality while addressing issues like hallucinations in code. Lastly, Zhang et al. (2024) and Surameery and Shakor (2023) evaluated LLMs' capabilities in bug-fixing and debugging, finding significant effectiveness but also highlighting concerns about data leakage and the need for traditional validation methods. However, little research has explored their capability to apply design patterns effectively.

In this work, we assess the capability of five prominent LLMs—ChatGPT, Claude, Copilot, Gemini, and Meta – on design pattern applicability, defined as the ability of an LLM to recognize and correctly apply a suitable design pattern to a program that lacks explicit pattern implementation. We compare their performance using two case studies: the Point of Sale System (POSS) and the Smart Wallet System (SWS). For each case study, we identify commonly applicable design patterns using LLMs, formalize their properties using predicate logic for a rigorous assessment, and apply these patterns to the case studies using LLMs. The outcomes are evaluated against the formal pattern properties using a set of metrics, including the Property Satisfaction Rate (PSR), Critical Property Coverage (CPC), and Pattern Implementation Quality Score

☆ Editor: Alexander Chatzigeorgiou.
E-mail address: kim2@oakland.edu.

https://doi.org/10.1016/j.jss.2025.112519
Received 19 January 2025; Received in revised form 22 April 2025; Accepted 28 May 2025
Available online 14 June 2025

(PIQS). Findings indicate that Claude scored the highest PIQS of 89.51, followed by Meta at 88.98, ChatGPT at 87.75, Copilot at 82.69, and Gemini at 71.04.

We address the following research questions:

RQ1: How can we systematically assess the capability of LLMs in design pattern applicability? First, a systematic assessment process should be established.

RQ2: How can design patterns' properties be specified in a form that can facilitate the assessment of the quality of pattern implementation? These properties should be unambiguously defined to serve as a basis for evaluating the correctness of pattern implementation.

RQ3: What metrics should be employed to measure the quality of design pattern implementation by LLMs? These metrics should measure both the breadth and depth of adherence to the functional correctness of the patterns as implemented by the LLMs.

RQ4: What is the capability of LLMs in design pattern applicability and what implications do the findings have? The involved LLMs should be measured for their capability of pattern applicability, which should provide useful insights for practitioners.

The remainder of the paper is organized as follows. Section 2 discusses relevant work on LLMs in software engineering. Section 3 describes the proposed approach for assessing LLMs' capability in pattern application, including the metrics used in assessing the quality of pattern implementation. Section 4 presents case studies evaluating LLMs' capability of pattern applicability. Section 5 analyzes the findings from the case studies, offering valuable insights for practitioners. Section 6 addresses the research questions introduced in Section 1, drawing on the results and insights analyzed in the preceding sections. Section 7 discusses threats to the validity of findings. Section 8 concludes the paper with a discussion on future work.

2. Related work

In this section, we give an overview of research examining the role and impact of LLMs in software engineering, organized by themes and areas of application.

Recent studies have explored the intersection of LLMs and software design pattern recognition, detection, and application. The work by Pan et al. (2025) investigated LLMs' understanding of design patterns – specifically GPT-4o and LLaMA – through classification and code generation tasks such as line completion and function generation, using Python and Java codebases. Despite the use of explicit prompts, the study reported limited accuracy (approximately 38.81%), with frequent misclassifications – particularly for Singleton and Factory patterns – due to structural overlaps, highlighting the need for improved contextual understanding and model training. Schindler and Rausch (2025) explored design pattern detection using a two-prompt method with ChatGPT-3.5 and 4, showing improved precision with version 4, but also noting persistent challenges related to token limits and partial pattern implementations. Andrade et al. (2025) proposed a real-time, LLM-based framework for detecting design antipatterns and security vulnerabilities in CI/CD environments, leveraging contextual heuristics to outperform traditional static analysis tools in scalability and accuracy. Similarly, Supekar and Khande (2024) addressed the adoption gap of design patterns in development by integrating LLM-based suggestions and automated refactoring tools into IDEs, enhancing developer productivity and reducing technical debt through real-time pattern recommendations. Pandey et al. (2025) assessed the generalization capacity of LLM embeddings (e.g., RoBERTa, CodeGPT) in classifying GoF creational patterns using a k-NN classifier, achieving high F1-scores (up to 0.91) without code compilation or domain-specific pretraining. In our previous work (Kim, 2025b), we introduced a quantitative framework using Role-Based Metamodeling Language (RBML) to evaluate ChatGPT-4's capability in applying the Visitor pattern across multiple case studies, achieving high conformance (98%) and completeness (87%), while revealing nuanced model behavior—such as omitting trivial visitor implementations or selectively modifying class hierarchies based on contextual reasoning. Unlike these prior studies, which primarily focus on pattern detection, classification, or security-related implications, the current work emphasizes the application of design patterns to existing code. We introduce a formal predicate-logic-based specification of pattern properties, combined with a multi-metric evaluation framework (PSR, CPC, PIQS), enabling systematic, pattern-agnostic, and cross-model assessment of LLM-generated implementations across multiple design patterns and application domains.

Recent work has also examined the use of LLMs in code refactoring. Liu et al. (2024) investigated GPT-4 and Gemini for identifying Java refactorings, improving success rates from 15.6% to 86.7% through prompt engineering, and proposed RefactoringMirror to safely apply LLM suggestions with a 94.3% success rate. Pomian et al. (2024) introduced EM-Assist, an IntelliJ IDEA plugin that enhances extract method refactoring by filtering LLM output using static analysis and program slicing, achieving 63% recall@5 on real-world refactorings and receiving 81.3% developer approval in surveys. Similarly, Cordeiro et al. (2024) evaluated StarCoder2's refactoring capabilities, finding it outperformed developers in reducing code smells (44.36% vs. 24.27%), though human expertise remained crucial for complex, context-aware refactorings.

Several studies have provided detailed analyses of LLMs' impact on software engineering. Fan et al. (2023) surveyed LLM applications across various aspects of software engineering, from coding to analytics, highlighting both opportunities for enhanced productivity and challenges in ensuring output reliability. Nguyen-Duc et al. (2023) identified 78 research questions across 11 areas, noting that while implementation and maintenance receive substantial attention, areas like requirements engineering need further exploration. Ozkaya (2023) discussed both the benefits and risks of LLMs, emphasizing the importance of responsible AI practices and ethical considerations.

Research has shown positive results in requirements engineering and early development phases. Marques et al. (2024) evaluated ChatGPT's role in requirements engineering, finding it effective in improving stakeholder communication while noting limitations in accuracy. Kim et al. (2023) assessed ChatGPT's capabilities across various development phases, reporting over 90% success in code generation but identifying issues with traceability and consistency. White et al. (2023) introduced a systematic approach through prompt design patterns for requirements elicitation and system design.

Studies have extensively examined LLMs' capabilities in code generation and development support. Solohubov et al. (2023) demonstrated significant efficiency gains when using ChatGPT and Copilot in coding tasks. Dakhel et al. (2023) evaluated Copilot's effectiveness in solving algorithmic problems, finding that while solutions might be less diverse than human-generated ones, they were often easier to repair. Pudari and Ernst (2023) analysis of Copilot revealed strong capabilities in generating syntactically correct code but limitations in recommending idiomatic practices.

Research in architectural design has revealed both potential and challenges. Ahmad et al. (2023) explored ChatGPT's role in architecture-centric software engineering, proposing a collaborative framework for human-bot interaction in architectural design. Their work demonstrated benefits in increasing architects' productivity while highlighting concerns about response variability and bias. Rajbhoj et al. (2024) presented a systematic prompting approach for software development lifecycle integration, emphasizing LLMs' potential in bridging the skills gap.

Several studies have focused on testing and quality assurance aspects. Kirinuki and Tanno (2024) found that ChatGPT could generate test cases with coverage comparable or superior to human testers, though noting limitations in boundary testing. Alshahwan et al. (2024)

introduced Assured LLM-Based Software Engineering, employing generate-and-test methods with semantic filters to ensure code quality. Their approach addresses hallucination issues while enabling performance optimization and debugging.

Research has shown encouraging results in bug-related tasks. Zhang et al. (2024) evaluated ChatGPT's bug-fixing capabilities using both existing and new benchmarks, demonstrating notable success rates while raising concerns about data leakage. Surameery and Shakor (2023) explored ChatGPT's potential in bug prediction and debugging assistance, noting its effectiveness while emphasizing the need for traditional debugging validation. Asare et al. (2023) study of Copilot found it less prone to introducing vulnerabilities compared to human developers.

Studies have examined LLMs' impact on learning and professional development. Waseem et al. (2024) investigated ChatGPT's effectiveness in helping students understand software development tasks, noting its value in bridging skill gaps while emphasizing the need to maintain critical thinking skills. Bera et al. (2023) assessed ChatGPT's potential as an agile coach, finding it accurate in concept explanation but requiring careful integration into professional teams.

Recent work has highlighted emerging challenges and opportunities. Hassan et al. (2024) proposed developing more collaborative AI pair programmers to address limitations in existing tools, emphasizing the need for greater context awareness. Wang et al. (2024) introduced BurstGPT, a dataset for improving LLM serving systems, addressing the gap in realistic workload data for system optimization.

3. Approach

In this section, we describe the proposed method for systematically assessing the capability of LLMs in applying design patterns. Fig. 1 illustrates the overall evaluation framework for assessing LLMs' capabilities in applying design patterns. The process begins with a case study (input) consisting of a baseline program that lacks explicit pattern implementations. Each LLM is prompted to identify applicable design patterns, and the resulting suggestions are consolidated to extract a set of commonly applicable patterns for consistent comparison. For each identified pattern, a formal specification is defined using predicate logic to capture essential structural and behavioral properties. These patterns are then applied to the original program by each LLM, producing refactored implementations. The implementations are evaluated against the formal specifications using three metrics – Property Satisfaction Rate (PSR), Critical Property Coverage (CPC), and Pattern Implementation Quality Score (PIQS) – which together yield quantitative measures of implementation quality. In the figure, rectangles represent data artifacts (e.g., source programs, specifications), circles represent operations (e.g., pattern identification, application, evaluation), and cylinders represent the LLMs as tools. This structured workflow enables consistent, objective, and reproducible assessment of design pattern implementation across different LLMs, supporting systematic comparative analysis.

The framework is grounded in three key principles to ensure objectivity and rigor. First, predicate logic provides a formal foundation for specifying pattern properties, enabling unambiguous definitions and avoiding the interpretive variability of natural language descriptions. Second, the metrics – PSR, CPC, and PIQS – capture different dimensions of quality: PSR reflects the breadth of property satisfaction, CPC emphasizes the depth of critical property coverage, and PIQS integrates both into a composite score, with calibrated weights to balance comprehensiveness and significance. This layered approach allows for nuanced evaluation and model-to-model comparison. Third, the framework is designed for cross-domain applicability, as demonstrated by the retail and financial system case studies, ensuring that the methodology generalizes across different software contexts.

For each case study, we start with a program that lacks pattern implementation and use LLMs to identify applicable design patterns. In this research, we evaluate five popular LLMs: ChatGPT (ChatGPT-4o, released in May 2024), Claude (Claude 3.5 Sonnet, released in June 2024), Copilot (Microsoft 365 Copilot, released in November 2023), Gemini (Gemini 1.5, released in February 2024), and Meta (Llama 3.1 – 405B, released in July 2024). Each LLM is tasked with identifying suitable design patterns for the program, guided by the following prompt.

```
prompt: Identify applicable design patterns for the given program.
```

The patterns identified by each LLM are consolidated and further filtered to determine common patterns for a fair comparison. For each commonly identified pattern, a formal specification is developed, defining the properties of the pattern in terms of predicate logic. These commonly identified patterns are then applied to the case study using each LLM, resulting in a refactored program with the pattern implementation. The output is subsequently assessed against the pattern specifications using a set of metrics.

3.1. Applicable design patterns

In this study, we employ two case studies: the Point of Sale System (POSS) and the Smart Wallet System (SWS). These systems were deliberately selected to represent diverse domains, with POSS from the retail transaction domain and SWS from the financial management domain. Each system is analyzed by individual LLMs to identify applicable design patterns. Each pattern identified by the LLMs was manually assessed for contextual relevance and applicability to the given programs, based on established definitions and design intent. Table 1 displays the design patterns identified as applicable to POSS by each LLM, with applicable patterns marked by a checkmark. Artifacts where the patterns can be applied are specified in parentheses. For instance, ChatGPT identifies several patterns: the Factory Method pattern applicable to the $makePayment$ method, the Strategy pattern applicable to the $ByCash/ByCreditCard$ class, the Composite pattern applicable to the $Sale/SaleLineItem$ class, the Template Method pattern to the $makePayment$ method, the Decorator pattern (location not specified), and the Observer pattern applicable to $Receipt$ generation. The table clearly identifies Factory Method, Strategy, and Composite, and Observer patterns as commonly applicable to POSS. This consistency suggests a common understanding of the fundamental design principles for POSS.

Table 2 presents the design patterns applicable to SWS, highlighting commonly applicable patterns such as Factory Method, Strategy, Singleton, and Observer. Initially, Gemini did not recognize the Factory Method and Singleton patterns as applicable, and Meta did not consider the Strategy pattern applicable. Subsequently, these models were queried to reassess the applicability of these patterns to specific locations identified by other models. The following prompt was used for this reassessment, which led to their confirmation of applicability:

```
prompt: Can [design pattern] be applicable to [location]?
```

From the two case studies, five common design patterns have been identified: Factory Method, Strategy, Composite, Observer, and Singleton. Table 3 details these patterns, describing their type category, complexity, and core aspects.

The two types of prompts employed reflect different practical use cases: (1) an open-ended prompt – "Identify applicable design patterns for the given program" – used to assess the model's pattern recognition capabilities without prior bias, and (2) a targeted prompt – "Apply [design pattern] to [location] in the given program" – used to evaluate the model's ability to implement a specific pattern consistently across all models. These prompts served distinct methodological purposes: the open-ended prompt simulates exploratory scenarios where a developer seeks design guidance, while the targeted prompt simulates

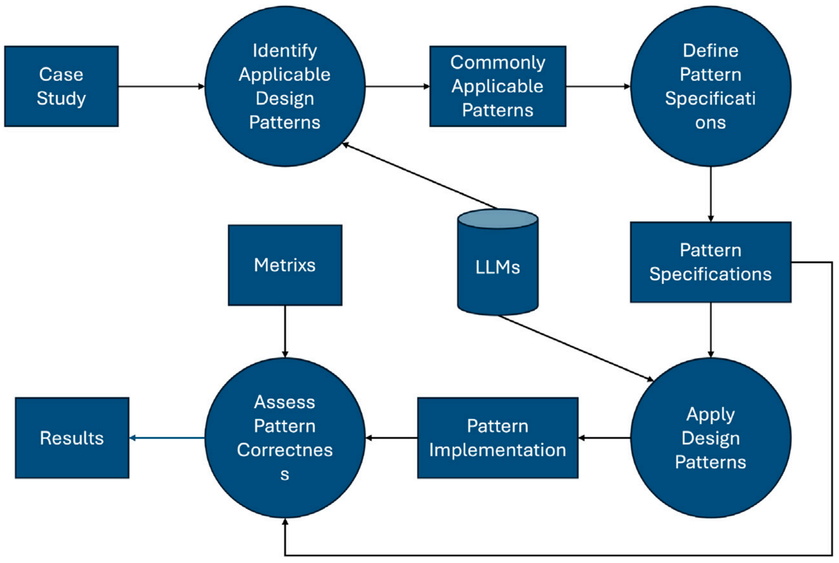
Case Study
Identify Applicable Design Patterns
Commonly Applicable Patterns
Define Pattern Specificati ons
Metrixs
LLMs
Pattern Specifications
Results
Assess Pattern Correctnes s
Pattern Implementation
Apply Design Patterns

Fig. 1. An overview of the approach.

Table 1
Design patterns identified as applicable to POSS by LLMs.

Pattern	ChatGPT	Claude	Copilot	Gemini	Meta
Factory method	✓ (makePayment method)	✓ (makePayment method)	✓ (Payment subclasses)	✓ (processSale method)	✓ (makePayment method)
Strategy	✓ (ByCash/ByCreditCard)	✓ (ByCash/ByCreditCard)	✓ (ByCash/ByCreditCard)	✓ (ByCash/ByCreditCard)	✓ (ByCash/ByCreditCard)
Composite	✓ (Sale/SaleLineItem)	✓ (Sale/SaleLineItem)	✓ (Sale/SaleLineItem)	✓ (Sale/SaleLineItem)	✓ (Sale/SaleLineItem)
Singleton	–	✓	✓	✓	–
Template method	✓ (makePayment)	–	–	–	✓ (processSale)
Decorator	✓	–	–	–	–
Command	–	–	✓	–	–
Observer	✓ (Receipt generation)	✓ (Register-Receipt)	✓ (Inventory updates)	✓ (External systems)	✓ (Inventory updates)

Table 2
Design patterns identified as applicable to SWS by LLMs.

Pattern	ChatGPT	Claude	Copilot	Gemini	Meta
Factory method	✓ (User class: addWallet method)	✓ (User class: addWallet method)	✓ (Wallet class)	–	✓ (Wallet class)
Strategy	✓ (Wallet class: Transaction methods)	✓ (Wallet class: Transaction methods)	✓ (CurrencyConverter class: ConversionStrategy)	✓ (CurrencyConverter class: conversion methods)	–
Composite	–	–	–	–	✓ (User class and Wallet objects)
Singleton	✓ (CurrencyConverter class)	✓ (CurrencyConverter class)	✓ (CurrencyConverter class)	–	✓ (CurrencyConverter class)
Decorator	✓ (Wallet class: Transaction methods)	–	✓ (Transaction class)	–	–
Command	–	–	–	–	✓ (Transaction history)
Observer	✓ (AuditLog class)	✓ (AuditLog class)	✓ (Wallet class: Transaction notifications)	✓ (Transaction notifications/balance updates)	✓ (AuditLog class)
Facade	–	✓ (User class)	–	✓ (Wallet class: account management methods)	–
Memento	–	–	–	✓ (Undo/redo transactions)	–
State	–	–	–	✓ (Wallet class behavior)	–

confirmatory use cases where a specific pattern has been selected and the developer needs help with implementation. For practitioners, the choice of prompt depends on context. The identification prompt is useful when developers are exploring design opportunities, while the application prompt is more suitable when the pattern is known and implementation assistance is required. Together, these approaches reflect two common practitioner workflows: discovering potential design opportunities and implementing confirmed design decisions.

3.2. Pattern specifications

For each of the commonly applicable patterns listed in Table 3, a formal pattern specification has been defined using predicate logic to unambiguously establish pattern properties based on the Gang of Four book (Gamma et al., 1995). Predicate logic is chosen for its ability to provide definitive ground values that determine the correctness of pattern properties.

Table 3
Commonly applicable design patterns.

Pattern type	Type category	Complexity	Core aspects
Factory method	Creational	Low	Factory method inheritance and overriding behavior, Product type hierarchy and relationships, Creator-product coupling management, Polymorphic instantiation correctness, Runtime type validation
Strategy	Behavioral	Medium	Strategy interface design and consistency, Context-strategy relationship management, Dynamic strategy switching mechanism, Algorithm encapsulation integrity, State data access and sharing
Composite	Structural	High	Recursive composition handling, Child management operations, Type safety across hierarchy, Component interface uniformity, Operation propagation through structure
Observer	Behavioral	High	Subject-observer synchronization, Event notification ordering, Circular update prevention, Observer registration management, State consistency maintenance
Singleton	Creational	Medium	Ensuring a class has only one instance, Managing global access to that instance, Lazy instantiation and thread safety, Constructor access control, Implementation of auto-disposable and thread-safe instances

3.2.1. Factory method pattern

The Factory Method pattern establishes an interface (typically an abstract class) for creating objects, termed products, while delegating the choice of which product class to instantiate to its subclasses, known as creators. This approach fosters loose coupling and supports adherence to the dependency inversion principle (Martin, 1996). The pattern involves a creator class that defines the factory method, responsible for returning an object of a specific product type. Concrete creators override this method to produce concrete products. Let E be the set of all types in the program and M be the set of all methods. We define predicates: $isAbstract(x)$ examines if element x is abstract, $isConcrete(x)$ verifies that x is concrete, $hasMethod(c, m)$ validates if class c defines method m, $returns(m, t)$ determines if method m has return type t, $implements(x, y)$ confirms an implementation relationship between concrete element x and abstract element y, $extends(x, y)$ checks an inheritance relationship between element x and element y, and $overrides(m1, m2)$ verifies method $m1$ overrides $m2$.

Table 4 outlines the pattern properties (F1-F5) and the predicates employed within these properties. The $creates(c, p)$ predicate verifies that creator class c can create product p, specifically by checking if c has a method that returns type p. The $hasFactory(c, m)$ predicate determines if class c has valid factory method m, ensuring that m returns a concrete product and follows proper factory pattern inheritance rules – either being abstract in an abstract creator class, or overriding an abstract factory method in a concrete creator. The $isProduct(x)$ predicate validates that class x fulfills the product role in the pattern, requiring it to either be abstract or, if concrete, implement an abstract product interface. Finally, the $isCreator(c)$ predicate confirms that class c properly functions as a creator in the pattern by verifying it has a valid factory method and maintains appropriate inheritance relationships – either being abstract with concrete implementations, or being concrete and implementing an abstract creator interface.

Property F1 mandates the existence of at least one abstract creator class, providing the template for object creation and setting the core structure of the pattern by deferring the specific type creation to subclasses. Property F2 requires that each abstract creator class has at least one concrete implementation, which allows the pattern to not only define interfaces but actually create objects. Property F3 states that factory methods in concrete creators must override the abstract factory method, enabling different concrete creators to produce various product types. According to Property F4, these factory methods must generate products that correctly fit into the designated product class hierarchy, ensuring type safety and proper relationships between creators and their products. Finally, Property F5 demands that any concrete product created must implement the corresponding abstract product interface, maintaining consistency and uniformity across the product hierarchy.

3.2.2. Strategy pattern

The Strategy pattern defines a family of algorithms, encapsulates each algorithm, and makes them interchangeable, allowing the algorithm to vary independently from the clients that use it. Let E be the set of all types and M be the set of all methods. We define predicates: $isAbstract(x)$ examines if element x is abstract, $isConcrete(x)$ verifies that x is concrete, $accepts(m, p)$ checks if method m takes parameter type p as an argument, $hasMethod(c, m)$ validates if class c defines method m, $calls(m1, m2)$ verifies if method $m1$ invokes method $m2$, $implements(x, y)$ confirms an implementation relationship between concrete element x and abstract element y.

Table 5 details the pattern properties (S1-S4) along with the predicates used in these properties. The $isAlgorithm(m)$ predicate validates that method m represents a strategy algorithm, being either abstract or concrete. The $algorithmMethod(m)$ confirms that m is a concrete implementation of a strategy algorithm. The $isStrategy(x)$ predicate verifies that class x fulfills the strategy role, either as an abstract interface or concrete implementation, with appropriate algorithm methods and inheritance relationships. The $hasStrategy(c, s)$ predicate checks if class c maintains a reference to strategy s by having a method that returns the strategy type. The $isSetStrategy(m)$ predicate validates that method m can accept a strategy as a parameter for strategy switching. The $isExecuteStrategy(m)$ predicate ensures that method m properly delegates to a strategy's algorithm method. The $isContext(x)$ predicate confirms that class x functions as a proper strategy context by having methods to set and execute strategies, maintaining strategy references, and properly delegating to them. Finally, the $delegates(c, s)$ predicate verifies that context c correctly delegates operations to its strategy s through appropriate method calls and relationships.

Property S1 specifies that there must be at least one abstract strategy interface, establishing the contract that all concrete strategies must adhere to and defining the strategy's behavior. Property S2 requires that each strategy interface has at least one concrete implementation, ensuring that the strategic behaviors are not only theoretical but also practically applied. Property S3 states that there must be at least one context class, a class that utilizes a strategy by maintaining a reference to a strategy object and delegating tasks to it. Finally, Property S4 mandates that all concrete strategy classes implement the algorithm method specified in the strategy interface, guaranteeing that each strategy offers a unique implementation of the method.

3.2.3. Composite pattern

The Composite pattern organizes objects into tree structures to represent recursive whole-part hierarchies, allowing clients to treat individual objects and compositions uniformly. Let E be the set of all types and M be the set of all methods. We define predicates: $isAbstract(x)$ examines if element x is abstract, $isConcrete(x)$ verifies that x is concrete, $accepts(m, p)$ checks if method m takes parameter type p as an argument, $hasMethod(c, m)$ validates if class c defines method m, $isAdd(m)$ validates if method m is an add operation, $isRemove(m)$ validates if method m is a remove operation, and $implements(x, y)$ confirms an implementation relationship between concrete element x and abstract element y.

Table 6 provides details of the pattern properties (C1-C5) and the predicates associated with these properties. The $isComponent(x)$ predicate validates that class x functions as a component in the pattern

Table 4
Factory method pattern specification.

Type	Expression	Definition
Predicates	$creates(c, p)$	$\mathtt{creates}(c \in E, p \in E) \equiv \exists m \in M \cdot \mathtt{hasMethod}(c, m) \wedge \mathtt{returns}(m, p)$
	$hasFactory(c, m)$	$\mathtt{hasFactory}(c \in E, m \in M) \equiv \mathtt{hasMethod}(c, m) \wedge \exists p \in E \cdot \mathtt{returns}(m, p) \wedge \mathtt{isConcrete}(p) \wedge ((\mathtt{isAbstract}(c) \wedge \mathtt{isAbstract}(m)) \vee (\mathtt{isConcrete}(c) \wedge \exists c_a \in E, m_a \in M \cdot \mathtt{implements}(c, c_a) \wedge \mathtt{isAbstract}(m_a) \wedge \mathtt{overrides}(m, m_a)))$
	$isProduct(x)$	$\mathtt{isProduct}(x \in E) \equiv (\mathtt{isAbstract}(x) \vee \mathtt{isConcrete}(x)) \wedge (\mathtt{isConcrete}(x) \rightarrow \exists i \in E \cdot \mathtt{isAbstract}(i) \wedge \mathtt{implements}(x, i))$
	$isCreator(c)$	$\mathtt{isCreator}(x \in E) \equiv \exists m \in M \cdot \mathtt{hasFactory}(x, m) \wedge ((\mathtt{isAbstract}(x) \wedge \exists c \in E \cdot \mathtt{isConcrete}(c) \wedge \mathtt{implements}(c, x)) \vee (\mathtt{isConcrete}(x) \wedge \exists a \in E \cdot \mathtt{isAbstract}(a) \wedge \mathtt{implements}(x, a)))$
Properties	F1	$\exists c \in E \cdot \mathtt{isAbstract}(c) \wedge \mathtt{isCreator}(c)$
	F2	$\forall a_c \in E \cdot \mathtt{isAbstract}(a_c) \wedge \mathtt{isCreator}(a_c) \rightarrow \exists c_c \in E \cdot \mathtt{isConcrete}(c_c) \wedge \mathtt{isCreator}(c_c)$
	F3	$\forall c_1, c_2 \in E, m_1, m_2 \in M \cdot \mathtt{isCreator}(c_1) \wedge \mathtt{isCreator}(c_2) \wedge \mathtt{hasFactory}(c_1, m_1) \wedge \mathtt{hasFactory}(c_2, m_2) \rightarrow \mathtt{overrides}(m_2, m_1) \vee \mathtt{implements}(m_2, m_1)$
	F4	$\forall c \in E, m \in M \cdot \mathtt{isCreator}(c) \wedge \mathtt{hasFactory}(c, m) \rightarrow \exists p \in E \cdot \mathtt{isProduct}(p) \wedge \mathtt{creates}(c, p) \wedge \mathtt{returns}(m, p)$
	F5	$\forall p \in E \cdot \mathtt{isProduct}(p) \wedge \mathtt{isConcrete}(p) \rightarrow \exists i \in E \cdot \mathtt{isProduct}(i) \wedge \mathtt{isAbstract}(i) \wedge \mathtt{implements}(p, i)$

Table 5
Strategy and algorithm specifications.

Type	Expression	Definition
Predicates	$isAlgorithm(m)$	$\mathtt{isAlgorithm}(m \in M) \equiv \mathtt{isAbstract}(m) \vee \mathtt{isConcrete}(m)$
	$algorithmMethod(m)$	$\mathtt{algorithmMethod}(m \in M) \equiv \mathtt{isAlgorithm}(m) \wedge \mathtt{isConcrete}(m)$
	$isStrategy(x)$	$\mathtt{isStrategy}(x \in E) \equiv (\mathtt{isAbstract}(x) \vee \mathtt{isConcrete}(x)) \wedge \exists m \in M \cdot \mathtt{hasMethod}(x, m) \wedge \mathtt{isAlgorithm}(m) \wedge (\mathtt{isConcrete}(x) \rightarrow \exists i \in E \cdot \mathtt{isAbstract}(i) \wedge \mathtt{implements}(x, i))$
	$hasStrategy(c, s)$	$\mathtt{hasStrategy}(c \in E, s \in E) \equiv \exists m \in M \cdot \mathtt{hasMethod}(c, m) \wedge \mathtt{returns}(m, s) \wedge \mathtt{isStrategy}(s)$
	$isSetStrategy(m)$	$\mathtt{isSetStrategy}(minM) \equiv \exists pinE \cdot \mathtt{accepts}(m, p) \wedge \mathtt{isStrategy}(p)$
	$isExecuteStrategy(m)$	$\mathtt{isExecuteStrategy}(minM) \equiv \exists sinE, m_2inM \cdot \mathtt{isStrategy}(s) \wedge \mathtt{hasMethod}(s, m_2) \wedge \mathtt{isAlgorithm}(m_2) \wedge \mathtt{calls}(m, m_2)$
	$isContext(x)$	$\mathtt{isContext}(xinE) \equiv \mathtt{isConcrete}(x) \wedge \exists m_1, m_2inM, sinE \cdot \mathtt{hasMethod}(x, m_1) \wedge \mathtt{hasMethod}(x, m_2) \wedge \mathtt{isSetStrategy}(m_1) \wedge \mathtt{isExecuteStrategy}(m_2) \wedge \mathtt{hasStrategy}(x, s) \wedge \mathtt{isAbstract}(s) \wedge \mathtt{delegates}(x, s)$
	$delegates(c, s)$	$\mathtt{delegates}(cinE, sinE) \equiv \mathtt{hasStrategy}(c, s) \wedge \exists m_1, m_2inM \cdot \mathtt{hasMethod}(c, m_1) \wedge \mathtt{hasMethod}(s, m_2) \wedge \mathtt{algorithmMethod}(m_2) \wedge \mathtt{calls}(m_1, m_2)$
Properties	S1	$\exists sinE \cdot \mathtt{isStrategy}(s) \wedge \mathtt{isAbstract}(s)$
	S2	$\forall sinE \cdot \mathtt{isStrategy}(s) \wedge \mathtt{isAbstract}(s) \rightarrow \exists cinE \cdot \mathtt{isStrategy}(c) \wedge \mathtt{isConcrete}(c) \wedge \mathtt{implements}(c, s)$
	S3	$\exists cinE \cdot \mathtt{isContext}(c)$
	S4	$\forall sinE \cdot \mathtt{isStrategy}(s) \wedge \mathtt{isConcrete}(s) \rightarrow \exists minM \cdot \mathtt{hasMethod}(s, m) \wedge \mathtt{algorithmMethod}(m)$

hierarchy, being either abstract or concrete, with concrete components implementing an abstract interface. The $hasChildren(x)$ predicate verifies that class x can manage child components through add and remove operations. The $isAddChild(m)$ and $isRemoveChild(m)$ predicates confirm that method m is a valid child remove operation that accepts a component parameter. The $containsComponent(x, y)$ predicate checks if class x can contain component y through child management capabilities. The $isComposite(x)$ predicate validates that class x functions as a composite by being a concrete component that can contain other components. Finally, the $isLeaf(x)$ predicate verifies that class x is a leaf node in the pattern by being a concrete component that does not manage children.

Property C1 ensures the existence of an abstract component type, which defines a common interface for all components within the tree structure, streamlining interactions with each element. Property C2 states that there must be at least one concrete class representing leaf nodes – objects in the composite structure without children. Property C3 requires the presence of a composite type, a concrete class designed to encapsulate other components, including both leaves and additional composites. Property C4 mandates that both composite and leaf classes adhere to this common component interface, promoting a consistent approach to handling all objects within the system. Finally, Property C5 emphasizes the necessity for uniform treatment of both individual objects and compositions of objects through the common interface, ensuring that operations on single components and their groupings are indistinguishable in practice.

3.2.4. *Observer pattern*

The Observer pattern defines a one-to-many dependency between objects so that when one object (subject) changes state, all its dependents (observers) are notified and updated automatically. Let E be the set of all types and M be the set of all methods. We define predicates: $isAbstract(x)$ examines if element x is abstract, $isConcrete(x)$ verifies that x is concrete, $reads(m, p)$ checks if method m accesses/retrieves values from parameter p without changing them, $modifies(m, p)$ verifies if method m changes the state of parameter p, $modifiesCollection(m, c, p)$ validates if method m modifies collection c with element p, $traversesCollection(m, c)$ validates if method m iterates over collection c, $increases(m, c)$ validates if method m adds elements to collection c, $decreases(m, c)$ validates if method m removes elements from collection c, $hasMethod(c, m)$ validates if class c defines method m, $calls(m1, m2)$ verifies if method $m1$ invokes method $m2$, and $implements(x, y)$ confirms an implementation relationship between concrete element x and abstract element y.

Table 7 outlines the pattern properties (O1-O4) and the predicates associated with these properties. The $isObserver(x)$ predicate validates that class x can function as an observer by having an update method. The $isUpdate(m)$ predicate confirms that method m is a valid update operation that reads but does not modify its parameters. The $isSubject(x)$ predicate verifies that class x can act as a subject with proper observer registration and notification capabilities. The $isRegisterObserver(m)$ and $isUnregisterObserver(m)$ predicates validate methods that handle observer collection management, with the former adding observers and the latter removing them. The $isNotify(m)$ predicate ensures method m properly iterates through observers and calls their update methods. The $observes(o, s)$ predicate confirms an observer-subject relationship exists. The $notifies(s, o)$ predicate verifies that subject s correctly notifies observer o through appropriate method calls. Finally, the $updates(o, s)$ predicate validates that observer o updates in response to subject s's state changes through proper update method implementation.

Table 6
Component pattern specification.

Type	Expression	Definition
Predicates	$isComponent(x)$	$\texttt{isComponent}(x \in E) \equiv (\texttt{isAbstract}(x) \vee \texttt{isConcrete}(x)) \wedge (\texttt{isConcrete}(x) \rightarrow \exists i \in E \cdot \texttt{isAbstract}(i) \wedge \texttt{implements}(x, i))$
	$hasChildren(x)$	$\texttt{hasChildren}(x \in E) \equiv \exists m_1, m_2 \in M \cdot \texttt{hasMethod}(x, m_1) \wedge \texttt{hasMethod}(x, m_2) \wedge \texttt{isAddChild}(m_1) \wedge \texttt{isRemoveChild}(m_2)$
	$isAddChild(m)$	$\texttt{isAddChild}(m \in M) \equiv \exists p \in E \cdot \texttt{accepts}(m, p) \wedge \texttt{isComponent}(p) \wedge \texttt{isAdd}(m)$
	$isRemoveChild(m)$	$\texttt{isRemoveChild}(m \in M) \equiv \exists p \in E \cdot \texttt{accepts}(m, p) \wedge \texttt{isComponent}(p) \wedge \texttt{isRemove}(m)$
	$containsComponent(x, y)$	$\texttt{containsComponent}(x, y \in E) \equiv \texttt{hasChildren}(x) \wedge \texttt{isComponent}(y) \wedge \exists m \in M \cdot \texttt{hasMethod}(x, m) \wedge \texttt{accepts}(m, y)$
	$isComposite(x)$	$\texttt{isComposite}(x \in E) \equiv \texttt{isComponent}(x) \wedge \texttt{isConcrete}(x) \wedge \exists c \in E \cdot \texttt{containsComponent}(x, c)$
	$isLeaf(x)$	$\texttt{isLeaf}(x \in E) \equiv \texttt{isComponent}(x) \wedge \texttt{isConcrete}(x) \wedge \neg\texttt{hasChildren}(x)$
Properties	C1	$\exists c \in E \cdot \texttt{isComponent}(c) \wedge \texttt{isAbstract}(c)$
	C2	$\exists l \in E \cdot \texttt{isLeaf}(l)$
	C3	$\exists m \in E \cdot \texttt{isComposite}(m)$
	C4	$\forall c \in E \cdot \texttt{isComponent}(c) \wedge \texttt{isAbstract}(c) \rightarrow (\exists l \in E \cdot \texttt{isLeaf}(l) \wedge \texttt{implements}(l, c)) \wedge (\exists m \in E \cdot \texttt{isComposite}(m) \wedge \texttt{implements}(m, c))$
	C5	$\forall c_1, c_2 \in E \cdot (\texttt{isLeaf}(c_1) \vee \texttt{isComposite}(c_1)) \wedge (\texttt{isLeaf}(c_2) \vee \texttt{isComposite}(c_2)) \rightarrow \exists i \in E \cdot \texttt{isComponent}(i) \wedge \texttt{implements}(c_1, i) \wedge \texttt{implements}(c_2, i)$

Table 7
Observer pattern specifications.

Type	Expression	Definition
Predicates	$isObserver(x)$	$\texttt{isObserver}(x \in E) \equiv (\texttt{isAbstract}(x) \vee \texttt{isConcrete}(x)) \wedge \exists m \in M \cdot \texttt{hasMethod}(x, m) \wedge \texttt{isUpdate}(m)$
	$isUpdate(m)$	$\texttt{isUpdate}(m \in M) \equiv \exists p \in E \cdot \texttt{accepts}(m, p) \wedge \texttt{reads}(m, p) \wedge \neg\texttt{modifies}(m, p)$
	$isSubject(x)$	$\texttt{isSubject}(x \in E) \equiv (\texttt{isAbstract}(x) \vee \texttt{isConcrete}(x)) \wedge \exists m_1, m_2, m_3 \in M \cdot \texttt{hasMethod}(x, m_1) \wedge \texttt{hasMethod}(x, m_2) \wedge \texttt{hasMethod}(x, m_3) \wedge \texttt{isRegisterObserver}(m_1) \wedge \texttt{isUnregisterObserver}(m_2) \wedge \texttt{isNotify}(m_3) \wedge \exists o \in E \cdot \texttt{isObserver}(o) \wedge \texttt{observes}(o, x)$
	$isRegisterObserver(m)$	$\texttt{isRegisterObserver}(m \in M, o \in E) \equiv \texttt{isObserver}(o) \wedge \texttt{accepts}(m, o) \wedge \exists c \in M \cdot \texttt{modifiesCollection}(m, c, o) \wedge \texttt{increases}(m, c)$
	$isUnregisterObserver(m)$	$\texttt{isUnregisterObserver}(m \in M, o \in E) \equiv \texttt{isObserver}(o) \wedge \texttt{accepts}(m, o) \wedge \exists c \in M \cdot \texttt{modifiesCollection}(m, c) \wedge \texttt{decreases}(m, c)$
	$isNotify(m)$	$\texttt{isNotify}(m \in M) \equiv \exists c \in E, o \in E, u \in M \cdot \texttt{traversesCollection}(m, c) \wedge \texttt{isObserver}(o) \wedge \texttt{hasMethod}(o, u) \wedge \texttt{isUpdate}(u) \wedge \texttt{calls}(m, u)$
	$observes(o, s)$	$\texttt{observes}(o \in E, s \in E) \equiv \texttt{isObserver}(o) \wedge \texttt{isSubject}(s)$
	$notifies(s, o)$	$\texttt{notifies}(s \in E, o \in E) \equiv \texttt{isSubject}(s) \wedge \texttt{isObserver}(o) \wedge \texttt{observes}(o, s) \wedge \exists m_1, m_2 \in M \cdot \texttt{hasMethod}(s, m_1) \wedge \texttt{hasMethod}(o, m_2) \wedge \texttt{isNotify}(m_1) \wedge \texttt{calls}(m_1, m_2)$
	$updates(o, s)$	$\texttt{updates}(o \in E, s \in E) \equiv \texttt{isObserver}(o) \wedge \texttt{isSubject}(s) \wedge \exists m \in M \cdot \texttt{hasMethod}(o, m) \wedge \texttt{isUpdate}(m) \wedge \texttt{accepts}(m, s)$
Properties	O1	$\exists s \in E \cdot \texttt{isSubject}(s) \wedge \texttt{isAbstract}(s)$
	O2	$\exists o \in E \cdot \texttt{isObserver}(o) \wedge \texttt{isAbstract}(o)$
	O3	$\forall s \in E, o \in E \cdot \texttt{isSubject}(s) \wedge \texttt{isObserver}(o) \wedge \texttt{observes}(o, s) \rightarrow \texttt{notifies}(s, o)$
	O4	$\forall s \in E, o \in E \cdot \texttt{isSubject}(s) \wedge \texttt{isObserver}(o) \wedge \texttt{notifies}(s, o) \rightarrow \texttt{updates}(o, s)$

Property O1 ensures that there is at least one abstract class defining the interface for subjects, which includes mechanisms for attaching and notifying observers of any changes. Property O2 requires the existence of at least one abstract class or interface that outlines how observers will update in response to notifications from the subject, establishing a standard method for observers to follow. Property O3 specifies that when a subject's state changes, it is obliged to notify all registered observers about the change, ensuring that all observers are kept informed of state modifications. Finally, Property O4 mandates that observers, upon receiving notifications, must update their state to reflect the new information provided by the subject.

3.2.5. Singleton pattern

The Singleton pattern ensures that a class has only one instance throughout its execution and provides a global access point to that instance. Let E be the set of all types, M be the set of all methods, and F be the set of all fields. We define predicates: $isPrivate(x)$ examines if element x has private access, $isStatic(x)$ verifies that x is static, $fieldType(f, t)$ verifies if field f is declared with type x, $belongsTo(f, c)$ verifies that field f belongs to class c, $hasConstructor(c, m)$ validates if class c defines constructor m, $accessesField(m, f)$ checks if method m includes any operations that read from or write to field f, $returns(m, t)$ determines if method m has return type t, and $calls(m1, m2)$ verifies if method $m1$ invokes method $m2$.

Table 8 lists the pattern properties (G1) and the predicates related to these properties. The $isSingleton(x)$ predicate validates that class x implements the complete singleton structure by having both a private constructor and proper instance management method. The $hasPrivateConstructor(x)$ predicate verifies that all constructors in class x are private, preventing external instantiation. The $hasInstanceMethod(x)$ predicate confirms that class x has a static method that returns the singleton instance and properly manages access to the static instance field. The $hasStaticInstance(x, f)$ predicate validates that class x has a properly encapsulated static field f that holds the singleton instance, ensuring it is both private and static with the correct type.

Property G1 stipulates that the constructor must be private to prevent direct instantiation from outside the class. This restriction enforces the singleton requirement by ensuring that the class can only be instantiated internally, maintaining control over how and when the instance is created.

3.3. Metrics

In this section, we define a set of metrics to measure the quality of pattern implementation.

Property Satisfaction Rate (PSR). The Property Satisfaction Rate provides a fundamental measure of how well an implementation adheres to the specified pattern properties. This metric calculates the percentage of satisfied properties relative to the total number of defined properties for a given pattern. A PSR above 90% indicates excellent conformance, suggesting a highly accurate implementation that captures the essential characteristics of the design pattern. Scores between 70%–89%

Table 8
Singleton pattern specifications.

Type	Expression	Definition
Predicates	$isSingleton(x)$	$\texttt{isSingleton}(x \in E) \equiv$ $\texttt{isConcrete}(x) \wedge \texttt{hasPrivateConstructor}(x) \wedge \texttt{hasInstanceMethod}(x)$
	$hasPrivateConstructor(x)$	$\texttt{hasPrivateConstructor}(x \in E) \equiv \forall m \in M \cdot \texttt{hasConstructor}(x, m) \rightarrow \texttt{isPrivate}(m)$
	$hasInstanceMethod(x)$	$\texttt{hasInstanceMethod}(x \in E) \equiv \exists m \in M \cdot \texttt{hasMethod}(x, m) \wedge \texttt{isStatic}(m) \wedge$ $\texttt{returns}(m, x) \wedge \exists f \in F \cdot \texttt{hasStaticInstance}(x, f) \wedge \texttt{accessesField}(m, f)$
	$hasStaticInstance(x, f)$	$\texttt{hasStaticInstance}(x \in E, f \in F) \equiv$ $\texttt{isStatic}(f) \wedge \texttt{isPrivate}(f) \wedge \texttt{fieldType}(f, x) \wedge \texttt{belongsTo}(f, x)$
Properties	G1	$\exists x \in E \cdot \texttt{isSingleton}(x)$

represent good conformance, while scores between 50%–69% indicate moderate conformance that may require improvements. Implementations scoring below 50% exhibit poor conformance and likely need significant revision.

$$PSR = \frac{\text{Number of satisfied properties}}{\text{Total number of properties}} \times 100 \tag{1}$$

Critical Property Coverage (CPC). The Critical Property Coverage metric introduces a weighted evaluation system that acknowledges the varying importance of different pattern properties. This sophisticated measure assigns higher weights to properties that are fundamental to the pattern's structure and behavior (weight of 3), moderate weights to properties governing relationships between components (weight of 2), and lower weights to supporting properties (weight of 1). This weighted approach provides a more nuanced assessment of pattern conformance by emphasizing the most crucial aspects of the design pattern implementation.

$$CPC = \left(\frac{\sum(w_i \times s_i)}{\sum w_i} \right) \times 100 \tag{2}$$

where w_i is the weight of property i, and s_i is the satisfaction status of property i (1 if satisfied, 0 if not).

Pattern Implementation Quality Score (PIQS). The Pattern Implementation Quality Score represents a comprehensive evaluation of pattern implementation quality, combining both PSR and CPC. This composite score provides a balanced measure that considers both the breadth of property satisfaction and the relative importance of each satisfied property. We assign a higher weight to the PSR component (0.6) to emphasize the importance of satisfying a comprehensive range of pattern properties, with the CPC component's weight (0.4) reflecting the complementary role of critical pattern-specific properties in ensuring structural and behavioral correctness. The choice of weights is inherently subjective and could be adjusted based on specific project requirements or organizational priorities,

$$\text{PIQS} = (\text{PSR} \times 0.6) + (\text{CPC} \times 0.4) \tag{3}$$

Property weights. Property weights are assigned for each design pattern based on their criticality to pattern implementation. Table 9 presents the weights assigned to the properties of the commonly applicable patterns listed in Table 3. Properties that define essential structural relationships or behavioral guarantees are considered critical and receive a weight of 3, such as the Factory Method's polymorphic object creation (F3) or the Composite pattern's uniform object treatment (C5). Properties that ensure proper pattern usage and relationships are deemed important and assigned a weight of 2, such as the Factory Method's existence of abstract creator classes (F1) or the Observer's existence of abstract subject classes (O1). Supporting properties that enhance implementation quality are given a weight of 1. While these weights reflect one approach to assessing pattern implementation criticality, they are subjective and can be adjusted based on specific project needs, or different interpretations of pattern importance.

The weights used in this study are established based on a combination of pattern-specific considerations (e.g., structural roles and behavioral guarantees) and general software engineering principles (e.g., cohesion, coupling, and separation of concerns). Critical properties are assigned higher weights because they define essential structural relationships or behavioral requirements fundamental to the correctness of a given pattern. Important properties, which govern component interactions and design intent, are assigned moderate weights, while supporting properties receive lower weights due to their more peripheral influence on overall pattern quality.

For practitioners adopting the framework, we recommend adjusting weights according to project-specific goals. For example, in safety-critical systems, properties related to behavioral correctness may warrant higher weights, whereas in maintainability-focused projects, structural integrity and modularity may be prioritized.

While the absolute metric scores may vary under different weighting schemes – as explicitly shown in the formulation of each metric – the framework is designed to make these dependencies transparent. Practitioners can readily interpret how weight adjustments affect evaluations and calibrate the framework accordingly. As noted earlier, weights used in the CPS metric can be customized based on project needs, whereas the weights used in the PIQS metric would typically remain stable, since they are primarily driven by quality attributes intrinsic to the design pattern being applied.

4. Case studies

To evaluate the performance of LLMs in applying design patterns, we use two Java-based programs – the Point of Sale System (POSS) and the Smart Wallet System (SWS). These programs were deliberately selected to ensure diversity, with POSS representing the retail transaction domain and SWS representing the financial management domain. This allows us to assess LLMs' pattern implementation capabilities across varying levels of domain complexity.

Table 10 provides detailed information about POSS and SWS, including their structural characteristics and complexity metrics. For POSS, it details a traditional retail transaction system comprising eleven classes (`POS`, `Register`, `Sale`, `Payment`, `Receipt`, `Item`, `SaleLineItem`, `ItemInventory`, `ByCash`, `ByCreditCard`, `PointOfSaleSystem`) with no interfaces. This system has 257 lines of code (LoC), implements 30 methods, and exhibits a cyclomatic complexity (CC) of 18, indicating moderate structural complexity. It features a simple inheritance hierarchy where `ByCash` and `ByCreditCard` classes extend the `Payment` class. It implements key retail functionalities, including inventory control, multi-method payment processing (cash and credit), and receipt generation. SWS represents a modern financial application with six classes (`User`, `Wallet`, `Transaction`, `CurrencyConverter`, `AuditLog`), similarly without interfaces. With 154 lines of code and 16 methods, it demonstrates a slightly lower cyclomatic complexity of 15 and lacks inheritance relationships. It supports operations such as multi-currency wallet management, fund transfers, real-time currency conversion, and transaction auditing. Both systems are implemented without interfaces and provide distinct operational domains—retail and finance—serving as diverse testbeds for evaluating the robustness and generalizability of LLM-assisted design pattern applications. Cyclomatic complexity

Table 9
Property weights of design patterns.

Pattern	Property	Weight	Justification
Factory method	F1. There exists at least one abstract creator class	2	Core structural requirement enabling the pattern
	F2. Every abstract creator has at least one concrete implementation	3	Essential for pattern usability and instantiation
	F3. Factory methods in concrete creators override the abstract factory method	3	Critical for polymorphic object creation
	F4. Factory methods create products of the correct type	3	Fundamental to type safety and pattern correctness
	F5. Concrete products implement their corresponding abstract product interfaces	2	Important for product hierarchy but builds on F4
Strategy	S1. There exists at least one abstract strategy interface	3	Core structural element enabling strategy operations
	S2. Every strategy interface has at least one concrete implementation	3	Essential for providing algorithm implementations
	S3. There exists at least one context class	2	Important for strategy management and execution
	S4. Concrete strategies implement the algorithm method	3	Fundamental to providing strategy behavior
Composite	C1. There exists an abstract component type	3	Core structural element enabling uniform treatment
	C2. There exists a leaf type	2	Important but simpler than composite structure
	C3. There exists a composite type	3	Critical for managing hierarchical structures
	C4. Both composite and leaf classes implement the component interface	3	Essential for uniform object treatment
	C5. Uniform treatment of composite and leaf objects	3	Fundamental to pattern's purpose
Observer	O1. There exists at least one abstract subject	2	Core structural element enabling observation
	O2. There exists at least one abstract observer	3	Essential for defining observer interface
	O3. Subjects notify all registered observers of changes	3	Critical for maintaining consistency
	O4. Observers update in response to subject notifications	3	Fundamental to pattern operation
Singleton	G1. Constructor is private to prevent direct instantiation	3	Core structural requirement

Table 10
Details of POSS and SWS.

Program name	Domain	Classes	LoC	Methods	CC	Inheritance hierarchy	Features
Point of sale system	Retail transaction	11 (POS, Register, Sale, Payment, Receipt, Item, SaleLineItem, ItemInventory, ByCash, ByCreditCard, PointOfSaleSystem)	257	30	18	Payment → ByCash, ByCreditCard	Handles retail sales transactions including inventory management, processes multiple payment types (cash/credit), manages product inventory and stock levels, generates transaction receipts, coordinates between sales, inventory, and payment subsystems
Smart wallet system	Financial management	6 (User, Wallet, Transaction, CurrencyConverter, AuditLog)	154	16	15	None	Manages multiple currency wallets, facilitates deposits and payments within wallets, converts between different currencies using live exchange rates, maintains a detailed transaction history and audit logs, supports user authentication, and logs significant user actions for security and tracking

LoC: Line of Code, CC: Cyclomatic Complexity.

was estimated by identifying control flow decision points – such as conditional branches, loops, and switch statements – according to the standard formula $M = E - N + 2$, where E denotes the number of edges and N the number of nodes in the control flow graph. The table's analysis of both systems' initial states provides a baseline for evaluating the impact and effectiveness of subsequent design pattern implementations by different LLMs.

While the case study systems are modest in size, they are deliberately selected to reflect realistic software design scenarios with manageable complexity, enabling focused and interpretable evaluation of LLM capabilities. These systems incorporate diverse architectural elements and design responsibilities, making them suitable for assessing pattern identification and application in a controlled setting.

These two programs were provided to LLMs along with the following prompt to facilitate the application of design patterns:

```
prompt: Apply [design pattern] to [location] in the given program.
```

Here, "location" refers to specific areas identified in Tables 1 and 2. The programs that implement design patterns produced by LLMs can be found in a GitHub repository, as referenced in Kim (2025a).

4.1. POSS

Table 11 presents a detailed analysis of POSS following the application of four design patterns – Factory Method, Strategy, Composite, and Observer, as identified in Table 1. It details the specific classes modified or created, the interfaces introduced, and the design patterns implemented by each LLM, illustrating the varied approaches to design pattern implementation in POSS. Additionally, it indicates whether the resulting programs are executable, reflecting the ability of each LLM to handle details meticulously and produce functional outcomes. Looking at the classes column, all LLMs maintained core components like POS, Sale, SaleLineItem, Item, and ItemInventory, but each added their own variations in payment processing and inventory management components. For instance, some (e.g., Meta) included specialized classes (e.g., PaymentProcessor) for payment processing or additional management functionality (e.g., SaleComponent), showing different approaches to system organization. The interfaces column reveals interesting variations in how each LLM approached abstraction. ChatGPT implemented PaymentStrategy, InventoryObserver, and InventorySubject interfaces, while Claude included a PaymentFactory interface. Meta's implementation stood out by introducing a unique SaleComponent interface alongside the more common payment and inventory-related interfaces. Additionally, it is worth noting that Meta is the only model that incorporates a Context component within the Strategy pattern, highlighting its unique approach to this design principle.

Regarding executability, only the programs generated by ChatGPT and Claude were executable without errors. The programs from the other models encountered issues that prevented execution. Copilot produced code with errors in the `POS` and `Receipt` classes. In

Table 11
Analysis of POSS after pattern application by individual LLMs.

Model	Classes	Interfaces	Patterns	Executable?
ChatGPT	POS, Sale, SaleLineItem, Item, ItemInventory, PaymentStrategy, PaymentFactory, InventoryObserver, InventorySubject	PaymentStrategy, InventoryObserver, InventorySubject	Factory, Strategy, Composite, Observer	Y
Claude	POS, Sale, SaleLineItem, Item, ItemInventory, PaymentStrategy, PaymentFactory, InventoryObserver, Register	PaymentStrategy, PaymentFactory, InventoryObserver	Factory, Strategy, Composite, Observer	Y
Copilot	POS, Sale, SaleLineItem, Item, ItemInventory, Payment, PaymentFactory, InventoryObserver, Register, Receipt	Payment, InventoryObserver	Factory, Strategy, Composite, Observer	N
Gemini	POS, Sale, SaleLineItem, Item, ItemInventory, Payment, PaymentFactory, InventoryObserver, Register	Payment, InventoryObserver	Factory, Strategy, Composite, Observer	N
Meta	POS, Sale, SaleLineItem, Item, ItemInventory, Payment, PaymentFactory, InventoryObserver, PaymentProcessor, SaleComponent	Payment, InventoryObserver, SaleComponent	Factory, Strategy, Composite, Observer	N

the `POS` class, the method call `r.updateInventory(i, quantity)` failed because the `Register` class defines a method `update(Item, int)` instead. In the `Receipt` class, the expression `df.format(payment.amount)` failed due to the absence of the `amount` attribute in the `Payment` class. Gemini's implementation exhibited errors across four classes – `InventoryObserver`, `ItemInventory`, `Register`, and `SaleLineItem` – due to the use of an undefined `Item` class, resulting in multiple method and field reference failures. Meta encountered errors in the driver class `RefactoredPOSMeta`, where it attempted to access the private `components` field of the `Sale` class and referenced undefined fields such as `item` and `quantity` in the `Item` class.

Table 12 provides a model evaluation for POSS after applying design patterns by LLMs. Each model's adherence to pattern properties for the Factory Method, Strategy, Composite, and Observer patterns is assessed. The table details each pattern's properties and how well they were satisfied by the model, providing specific usage locations and comments on the implementation. For example, ChatGPT partially satisfied the Factory Method properties, fully implemented the Strategy pattern, and partially satisfied the Observer pattern. Under the Factory Method, ChatGPT's implementation used a static factory instead of an abstract creator class, which is noted as a deviation from the ideal pattern implementation. This detailed comparison reveals varying levels of success across the different models, with some achieving full implementation of certain patterns while others showed partial satisfaction or missing important aspects of the pattern requirements. The table serves as a comparative analysis tool to understand how each LLM fares in implementing specific design patterns, highlighting strengths and weaknesses in their approach to refactoring code based on established software design principles.

Table 13 offers a quantitative analysis of LLMs' effectiveness in implementing design patterns, detailed by PSR, CPC, and PIQS. The table delineates these metrics for individual design patterns and aggregates them to derive average values across all patterns. The results highlight the distinct performance levels of the LLMs – Meta excels, achieving perfect scores across all metrics for all patterns, culminating in an average PIQS of 100. ChatGPT also performs robustly, especially excelling in Strategy, Composite, and Observer patterns, with an average PIQS of 95.46. In contrast, Claude scores moderately with an average PIQS of 89.12, Copilot displays mixed results with an average PIQS of 74.46, and Gemini lags behind, notably struggling with the Observer pattern, reflected in its lower average PIQS of 65.14.

4.2. SWS

Table 14 presents a detailed analysis of SWS following the application of four design patterns – Factory Method, Strategy, Singleton, and Observer, as identified in Table 2. The analysis reveals that while all models maintained the core classes (User, Wallet, Transaction, CurrencyConverter, AuditLog), they differed in their interface implementations – for example, ChatGPT used Observer and TransactionStrategy interfaces, while Claude introduced additional WalletFactory interfaces. It is worth noting that Meta is the only model that utilized the built-in observer implementation from the Java standard library.

With respect to executability, only the program produced by Claude was executable without errors. ChatGPT's implementation failed at runtime due to a `NullPointerException` caused by a null `exchangeRates` map in the `CurrencyConverter` class during a `put()` operation. Copilot's code introduced a type mismatch in the `RefactoredSWSCopilot` class by assigning the result of `addObserver()` – a method with a `void` return type – to a variable of type `AuditLog`. Gemini's implementation showed multiple issues across several classes – `DefaultWalletFactory`, `DepositStrategy`, `PaymentStrategy`, and `RefactoredSWSGemini` – including constructor mismatches and repeated attempts to access private fields (`balance`, `transactions`, and `currency`) in the `Wallet` class, as well as unauthorized access to the private `auditLog` field in the `User` class. Finally, Meta's implementation in the `RefactoredSWSMeta` class failed due to a call to `user.authenticate(‘‘123456’’)`, where the `User` class does not define an `authenticate()` method. These issues illustrate the challenges in ensuring that the implementations are not only correct in applying design patterns but also syntactically and semantically correct in their specific programming contexts.

Table 15 provides an evaluation of how LLMs implemented the Factory Method, Strategy, Singleton, and Observer patterns in SWS. It reveals that while most models fully satisfied the Strategy and Singleton patterns, there were varying levels of success with the Factory Method pattern, with common issues like missing abstract creators or incomplete factory inheritance hierarchies, and missing abstract subject for Observer pattern. More specifically, Meta fully implement the Factory Method pattern, ensuring payment processing capabilities, while Gemini shows partial implementation of the Observer pattern, indicating potential gaps in event monitoring and response functionalities.

Table 16 offers a quantitative analysis detailing the performance of LLMs in implementing the Factory Method, Strategy, Singleton, and Observer patterns in SWS using PSR, CPC, and PIQS. The data reveals a consistently high performance across all models in the Strategy and Singleton patterns, achieving 100% in all metrics. However, variability is noted in the implementation of the Factory Method and Observer patterns. Copilot stands out with the highest average PIQS of 90.92, demonstrating robustness particularly in the Observer pattern, while Claude also shows strong performance with an average PIQS of 89.89. ChatGPT and Meta display moderate effectiveness, with average PIQS scores of 80.05 and 77.97, respectively. In particular, ChatGPT faced challenges with the Factory Method pattern. Gemini, however, shows

Table 12
Evaluation of design pattern implementation in POSS by LLMs.

Model	Pattern	Properties	Satisfaction	Usage location	Comments
ChatGPT	Factory method	F1–F5	Partially satisfied	PaymentFactory. getPaymentMethod()	F1: Uses static factory instead of abstract creator class.
	Strategy	S1–S4	All satisfied	PaymentStrategy, ByCash, ByCreditCard	Fully implemented.
	Composite	C1–C5	All satisfied	SaleComponent, Sale (composite), SaleLineItem (leaf)	Fully implemented.
	Observer	O1–O4	Partially satisfied	ItemInventory (subject), StockAlert (observer)	Fully implemented.
Claude	Factory method	F1–F5	Partially satisfied	PaymentFactory	F4: Missing clear creator-product coupling relationship.
	Strategy	S1–S4	All satisfied	PaymentStrategy	Fully implemented.
	Composite	C1–C5	All satisfied	SaleComponent hierarchy	Fully implemented.
	Observer	O1–O4	Partially satisfied	ItemInventory, inventory observers	O1: ItemInventory implemented as concrete class instead of abstract.
Copilot	Factory method	F1–F5	Partially satisfied	PaymentFactory. getPaymentMethod()	F1: Implementation uses static factory method instead of proper abstract class.
	Strategy	S1–S4	All satisfied	Payment, payment methods	Fully implemented.
	Composite	C1–C5	Partially satisfied	SaleComponent hierarchy	C1, C4, C5: Fundamental component hierarchy missing.
	Observer	O1–O4	Partially satisfied	ItemInventory, stock observers	O1: Uses concrete subject instead of abstract class.
Gemini	Factory method	F1–F5	Partially satisfied	PaymentFactory	Fully implemented.
	Strategy	S1–S4	All satisfied	Payment	Fully implemented.
	Composite	C1–C5	Partially satisfied	SaleComponent	C1, C4, C5: Fundamental component hierarchy missing.
	Observer	O1–O4	Partially satisfied	ItemInventory, inventory observers	O1: Missing proper abstract classes for subject.
Meta	Factory method	F1–F5	All satisfied	PaymentFactory	Fully implemented.
	Strategy	S1–S4	All satisfied	Payment, concrete payments	Fully implemented.
	Composite	C1–C5	All satisfied	SaleComponent, Sale, and SaleLineItem	Fully implemented.
	Observer	O1–O4	All satisfied	ItemInventory	Fully implemented.

Table 13
Evaluation metrics for pattern implementation in POSS across LLMs.

Pattern	Metric	ChatGPT	Claude	Copilot	Gemini	Meta
Factory method	PSR	80.00	80.00	80.00	100.00	100.00
	CPC	84.62	76.92	84.62	100.00	100.00
	PIQS	81.85	78.77	81.85	100.00	100.00
Strategy	PSR	100.00	100.00	100.00	100.00	100.00
	CPC	100.00	100.00	100.00	100.00	100.00
	PIQS	100.00	100.00	100.00	100.00	100.00
Composite	PSR	100.00	100.00	40.00	40.00	100.00
	CPC	100.00	35.71	35.71	100.00	100.00
	PIQS	100.00	38.29	38.29	100.00	100.00
Observer	PSR	100.00	75.00	75.00	25.00	100.00
	CPC	100.00	81.82	81.82	18.18	100.00
	PIQS	100.00	77.73	77.73	22.27	100.00
Average	**PSR**	**95**	**88.75**	**73.75**	**66.25**	**100**
	CPC	**96.15**	**89.69**	**75.54**	**63.47**	**100**
	PIQS	**95.46**	**89.12**	**74.46**	**65.14**	**100**

lower performance across the board, particularly struggling with the Observer pattern, reflected in its lowest average PIQS of 76.94. This analysis highlights areas where each model excels or requires improvement, providing insights into their respective capabilities in adapting and applying complex design principles within a financial management context.

5. Analysis

In this section, we examine the findings detailed in Section 4. Table 17 presents the average evaluation metrics for pattern implementation in the Point of Sale System (POSS) and the Smart Wallet System (SWS) across various Large Language Models (LLMs). This table elucidates a distinct hierarchy of capabilities among the LLMs. Claude leads with

Table 14
Analysis of SWS after pattern application by individual LLMs.

Model	Classes	Interfaces	Patterns	Executable?
ChatGPT	User, Wallet, Transaction, CurrencyConverter, AuditLog	Observer, TransactionStrategy	Factory, Strategy, Singleton, Observer	N
Claude	User, Wallet, Transaction, CurrencyConverter, AuditLog, Subject	Observer, TransactionStrategy, WalletFactory	Factory, Strategy, Singleton, Observer	Y
Copilot	User, Wallet, Transaction, CurrencyConverter, AuditLog	TransactionObserver, TransactionStrategy, WalletFactory	Factory, Strategy, Singleton, Observer	N
Gemini	User, Wallet, Transaction, CurrencyConverter, AuditLog	WalletFactory, TransactionStrategy, AuditLogObserver	Factory, Strategy, Singleton, Observer	N
Meta	User, Wallet, Transaction, CurrencyConverter, AuditLog	TransactionStrategy, Observable, Observer	Factory, Strategy, Singleton, Observer	N

Table 15
Evaluation of design pattern implementation in SWS by LLMs.

Model	Pattern	Properties	Satisfaction	Usage location	Comments
ChatGPT	Factory method	F1–F5	Partially satisfied	User.addWallet()	F1: Missing abstract creator; F4: addWallet() not returning a wallet; F5: No abstract-concrete factory inheritance hierarchy.
	Strategy	S1–S4	All satisfied	Wallet. processTransaction()	Fully implemented.
	Singleton	G1	All satisfied	CurrencyConverter. getInstance()	Fully implemented.
	Observer	O1–O4	Partially satisfied	AuditLog.update()	O1: Missing abstract subject.
Claude	Factory method	F1–F5	All satisfied	User.addWallet()	F5: Missing abstract product.
	Strategy	S1–S4	All satisfied	Wallet. executeTransaction()	Fully implemented.
	Singleton	G1	All satisfied	CurrencyConverter. getInstance()	Fully implemented.
	Observer	O1–O4	All satisfied	User. notifyObservers()	Fully implemented.
Copilot	Factory method	F1–F5	All satisfied	WalletFactory. createWallet()	F5: Missing abstract product.
	Strategy	S1–S4	All satisfied	Wallet. performTransaction()	Fully implemented.
	Singleton	G1	All satisfied	CurrencyConverter. getInstance()	Fully implemented.
	Observer	O1–O4	Partially satisfied	AuditLog.notify()	Fully implemented.
Gemini	Factory method	F1–F5	Partially satisfied	DefaultWalletFactory. createWallet()	F5: Missing abstract product.
	Strategy	S1–S4	All satisfied	Wallet.addFunds(), Wallet.makePayment()	Fully implemented.
	Singleton	G1	All satisfied	CurrencyConverter. getInstance()	Fully implemented.
	Observer	O1–O4	Partially satisfied	AuditLog.logAction()	O1: Missing abstract subject. O3: Notification not implemented. O4: Update not implemented.
Meta	Factory method	F1–F5	Partially satisfied	User.addWallet()	F1: Missing abstract factory. F5: No abstract-concrete factory inheritance hierarchy.
	Strategy	S1–S4	All satisfied	Wallet. executeTransaction()	Fully implemented.
	Singleton	G1	All satisfied	CurrencyConverter. getInstance()	Fully implemented.
	Observer	O1–O4	All satisfied	AuditLog. logAction()	O2: Missing abstract observer; O4: Update not implemented.

an exemplary average Pattern Implementation Quality Score (PIQS) of 89.51, closely followed by Meta, which exhibits substantial variability with an average PIQS of 88.98. ChatGPT also shows strong performance, achieving an average PIQS of 87.75. Copilot displays a broader range of outcomes, indicated by its average PIQS of 82.69. Gemini, on the other hand, shows the most constrained capabilities, with the lowest average PIQS of 71.04. It is noteworthy that LLMs generally encountered greater challenges with the Factory Method and Observer patterns in SWS compared to POSS. In particular, ChatGPT showed the most significant discrepancy in the Factory Method pattern, achieving a PIQS of 81.85 in POSS but only 42.46 in SWS, marking the lowest performance among the models. This suggests that the specific context or complexity of the software system may significantly influence the effectiveness of LLMs in applying these patterns. Gemini consistently

Table 16
Evaluation metrics for pattern implementation in SWS across LLMs.

Pattern	Metric	ChatGPT	Claude	Copilot	Gemini	Meta
Factory method	PSR	40.00	80.00	60.00	80.00	60.00
	CPC	46.15	84.62	69.23	84.62	69.23
	PIQS	42.46	81.85	63.69	81.85	63.69
Strategy	PSR	100.00	100.00	100.00	100.00	100.00
	CPC	100.00	100.00	100.00	100.00	100.00
	PIQS	100.00	100.00	100.00	100.00	100.00
Singleton	PSR	100.00	100.00	100.00	100.00	100.00
	CPC	100.00	100.00	100.00	100.00	100.00
	PIQS	100.00	100.00	100.00	100.00	100.00
Observer	PSR	75.00	75.00	100.00	25.00	50.00
	CPC	81.82	81.82	100.00	27.27	45.45
	PIQS	77.73	77.73	100.00	25.91	48.18
Average	**PSR**	**78.75**	**88.75**	**90.00**	**76.25**	**77.50**
	CPC	**81.99**	**91.61**	**92.31**	**77.97**	**78.67**
	PIQS	**80.05**	**89.89**	**90.92**	**76.94**	**77.97**

Table 17
Average evaluation metrics for pattern implementation in POSS and SWS across LLMs.

Project	Metric	ChatGPT	Claude	Copilot	Gemini	Meta
POSS	IPQS	95.46	89.12	74.46	65.14	100.00
SWS	IPQS	80.05	89.89	90.92	76.94	77.97
Average	**IPQS**	**87.75**	**89.51**	**82.69**	**71.04**	**88.98**

faced difficulties with the Observer pattern in both POSS and SWS, recording the lowest PIQS among the models in both systems, which underscores its challenges with this pattern. This analysis clearly delineates the relative strengths and weaknesses of each LLM in applying design patterns to complex software systems.

The comparative analysis presented in Table 17 reveals several notable insights about LLM capabilities in design pattern implementation across different software contexts. Claude's leading position with an average PIQS of 89.51 demonstrates a high level of consistency across both POSS and SWS, suggesting reliably strong pattern implementation capabilities regardless of application context. Meta shows interesting performance variation, achieving perfect scores (PIQS: 100.00) in POSS but notably lower scores (PIQS: 77.97) in SWS, indicating potential domain-specific strengths and limitations. ChatGPT's performance pattern is reversed, showing stronger results in SWS (PIQS: 95.46) compared to POSS (PIQS: 80.05), which suggests its capabilities may be better suited to certain architectural contexts. Copilot demonstrates more balanced but moderate performance (average PIQS: 82.69), while Gemini's consistently lower scores (average PIQS: 71.04) across both domains indicate more fundamental limitations in pattern implementation capabilities. The variation in performance between POSS and SWS implementations across all LLMs suggests that domain contexts and specific pattern requirements (e.g., Composite pattern) significantly influence implementation quality. This finding has important implications for practitioners, indicating that LLM selection for design pattern implementation should consider both the specific domain context and the particular patterns being implemented. Furthermore, the consistent performance gaps between LLMs highlight the importance of understanding each model's strengths and limitations when incorporating them into software design workflows.

6. Discussion

In this section, we address the research questions posed in Section 1 based on the findings in previous sections.

***RQ1*:. How can we systematically assess the capability of LLMs in design pattern applicability?** The research established a systematic assessment approach through a structured evaluation framework. The process begins with pattern identification by LLMs and proceeds through a rigorous validation pipeline. As demonstrated in Fig. 1, the approach involves initial pattern identification, consolidation of commonly identified patterns, development of formal specifications, pattern implementation by LLMs, and assessment against those specifications using defined metrics. This systematic process ensures objectivity by relying on formal specifications rather than subjective criteria, employing quantitative metrics that enable consistent comparison across different LLMs, and supporting validation through multiple case studies across different domains.

***RQ2*:. How can design patterns' properties be specified in a form that can facilitate the assessment of the quality of pattern implementation?** The research demonstrated that predicate logic provides an effective formal specification mechanism for design pattern properties. This approach enables unambiguous definition of pattern characteristics through precise logical predicates that formally define relationships between components and explicitly state pattern constraints. Each pattern specification includes core predicates defining key concepts, property definitions using those predicates, formal relationships between components, and behavioral constraints. Using the Factory Method pattern as an example, the specification includes base predicates like $isAbstract(x)$ and $isConcrete(x)$, derived predicates such as $creates(c, p)$, and properties F1-F5 defining pattern requirements. This formal approach enables objective validation of implementations and consistent assessment across different LLMs.

***RQ3*:. What metrics should be employed to measure the quality of design pattern implementation by LLMs?** The research established three complementary metrics that together provide a thorough assessment of pattern implementation quality. The Property Satisfaction Rate (PSR) measures the basic percentage of satisfied pattern properties, with scores above 90% indicating excellent implementation and scores below 50% indicating poor implementation. The Critical Property Coverage (CPC) provides a weighted evaluation that considers property importance, with weights assigned based on whether properties are critical (3), important (2), or supporting (1). The Pattern Implementation Quality Score (PIQS) combines PSR and CPC into a composite metric, with weights of 0.6 and 0.4 respectively, to provide a balanced measure of both breadth and depth of implementation quality.

***RQ4*:. What is the capability of LLMs in design pattern applicability and what implications do the findings have?** The analysis revealed a clear hierarchy of capabilities across different LLMs. Claude emerged as the most consistent performer with an average PIQS of 89.51, followed by Meta at 88.98 and ChatGPT at 87.75. Copilot showed more variable results with a PIQS of 82.69, while Gemini demonstrated more limited capabilities with a PIQS of 71.04. While all LLMs demonstrated strong performance with the Strategy pattern, they showed varying capabilities with more complex patterns like Factory Method and Observer. Claude achieved the highest consistency across both POSS and SWS domains, while Meta and ChatGPT showed domain-specific strengths - Meta excelled in POSS (PIQS: 100.00) but performed lower in SWS (PIQS: 77.97), while ChatGPT showed the opposite pattern. Gemini showed consistent limitations across domains.

In this study, we employed a straightforward prompting approach with minimal engineering, using simple directives for both pattern identification (e.g., *"Identify applicable design patterns for the given program"*) and pattern application (e.g., *"Apply [design pattern] to [location] in the given program"*) to ensure consistency and comparability across LLMs. While this basic approach was intentionally chosen to evaluate LLMs' inherent capabilities without optimization through specialized prompting techniques, the findings suggest that prompt engineering could significantly impact both pattern identification and implementation quality. For example, Table 2 shows variation in pattern identification across LLMs, with some models requiring follow-up prompting to recognize patterns initially missed. Additionally, performance discrepancies observed in Table 17—such as Meta's perfect PIQS in POSS (100.00) but substantially lower in SWS (77.97), and ChatGPT's inverse trend—indicate that domain-specific prompt tailoring might yield better outcomes. Prompts incorporating contextual knowledge about the system (e.g., retail transaction flows for POSS or financial operations for SWS) could help guide models toward more appropriate design decisions. The consistently lower performance by Gemini across both domains (average PIQS: 71.04) further suggests that some models may benefit more substantially from prompt refinement to compensate for inherent limitations in their ability to reason about design structure. Despite these limitations, LLMs demonstrated reasonable performance even under minimal prompting. Claude's average PIQS of 89.51 suggests that strong results are possible without extensive prompt engineering; however, the gap from perfect scores indicates room for improvement. Techniques such as structured pattern templates (e.g., Kim (2025b)) that provide context, input examples, and expected outputs may further improve both pattern identification accuracy and implementation quality. Exploring such techniques represents an important direction for future work in optimizing LLM-assisted software design and refactoring.

These findings have important implications: practitioners should maintain appropriate validation processes (e.g., testing protocols) while leveraging LLMs; researchers should focus on improving abstract class hierarchies and inheritance relationships in design patterns due to frequent violations of pattern properties involving abstract elements; and tool developers should prioritize pattern validation capabilities (e.g., verifying pattern-specific properties and constraints). One potential approach is fine-tuning LLMs on curated datasets that emphasize the correct use of abstract interfaces and inheritance structures in the context of design patterns. Additionally, specialized prompting techniques – such as structured prompt design (Kim, 2025b) – may help guide models toward proper abstract class usage by explicitly framing inheritance relationships or providing representative examples within the prompt context. This may include clear instructions about required type hierarchies, or supplying skeleton code that establishes the expected abstract framework. These strategies merit further exploration in future work aimed at improving the structural accuracy and reliability of LLM-generated pattern implementations. The performance variations suggest that LLM selection should consider both domain context and specific pattern requirements, highlighting the value of a combined human-LLM approach to software design and the importance of understanding each model's strengths and limitations.

7. Threats to validity

In this section, we discuss potential threats to the validity of findings across three dimensions – internal, construct, and external validity.

Internal validity. Internal validity concerns primarily relate to the completeness and soundness of pattern specifications and evaluation process. While we derived pattern specifications from established pattern literature (Gamma et al., 1995), there may be subtle pattern properties or variations we failed to capture. To mitigate this concern, we conducted extensive literature reviews to ensure pattern specifications aligned with canonical pattern definitions and employed predicate logic to formalize pattern properties, reducing ambiguity. However, some pattern nuances, particularly those involving complex behavioral interactions (e.g., Observer pattern), may not be fully represented in pattern specifications. Also, the evaluation process itself could introduce bias through the interpretation of LLM outputs and their mapping to formal specifications.

External validity. The generalizability of findings faces several limitations that affect external validity. The evaluation relied on only two case studies (POSS and SWS), which may not fully represent the diverse spectrum of software systems encountered in practice. The selection of five specific design patterns, while fundamental, covers only a subset of the broader range of design patterns. The LLMs evaluated represent a snapshot in time, and their rapidly evolving capabilities mean the findings in this work may not reflect their current or future performance. Domain-specific variations in pattern implementation practices may not be fully captured by the approach. To mitigate these concerns, we intentionally selected case studies from diverse domains and chose design patterns that are commonly applicable to these case studies. We have documented the specific versions and capabilities of LLMs tested to facilitate future comparisons and explicitly acknowledged the contextual limitations of findings. Future research should expand the evaluation to more diverse case studies, additional design patterns, and newer LLM versions to strengthen the external validity of findings and their generalizability to broader software engineering contexts.

Construct validity. The primary threat to construct validity stems from metric construction and weighting scheme. The weights assigned to different pattern properties and the balance between PSR (0.6) and CPC (0.4) in calculating PIQS reflect subjective judgments about their relative importance in pattern implementation quality. Alternative weighting schemes could yield different model rankings. For instance, an equal weighting scheme would treat all properties as equally important, potentially simplifying the evaluation but ignoring structural criticality. Domain-specific weighting could prioritize properties differently depending on the application context (e.g., safety-critical systems might emphasize Observer notification guarantees more heavily). Empirical or data-driven approaches could assign weights based on expert consensus, frequency of violation in real-world systems, or observed defect rates. Additionally, machine learning–based techniques could infer optimal weights by aligning implementation features with expert evaluations. The potential impact of such alternatives is acknowledged, and their exploration is left for future work. The relationship between metrics and the underlying quality of pattern implementation they aim to measure may be imperfect, as different weight assignments could potentially lead to different rankings of LLM performance. Furthermore, the chosen weights may not reflect the true importance of different pattern aspects across all implementation contexts.

8. Conclusion

This research conducted a systematic evaluation of LLMs' capabilities in implementing design patterns through code refactoring. Through rigorous assessment using predicate logic specifications and quantitative metrics, we found varying levels of effectiveness across different LLMs – with Claude achieving the highest average PIQS of 89.51, followed by Meta at 88.98 and ChatGPT at 87.75. While LLMs demonstrated strong performance with simpler patterns like Strategy, they showed consistent difficulties with patterns requiring complex abstract hierarchies and inheritance relationships.

The findings suggest that while LLMs show promise as refactoring tools, they are best used as assistive technologies rather than replacements for human developers. Their effectiveness varies based on both the specific pattern being implemented and the domain context, indicating the need for careful consideration in their application.

Future work should focus on expanding the evaluation to cover more design patterns and diverse application domains to better understand LLMs' capabilities and limitations. Research is also needed to improve LLMs' handling of abstract class hierarchies and inheritance relationships, potentially through specialized prompting techniques or fine-tuning approaches. Additionally, as LLM technology continues to evolve rapidly, ongoing assessment of newer models will be crucial for understanding how these tools can best support software design and maintenance activities.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared the link to my data.

References

Ahmad, A., Waseem, M., Liang, P., Fahmideh, M., Aktar, M.S., Mikkonen, T., 2023. Towards Human-Bot collaborative software architecting with ChatGPT. In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering. Oulu, Finland, pp. 279–285.

Alshahwan, N., Harman, M., Harper, I., Marginean, A., Sengupta, S., Wang, E., 2024. Assured offline LLM-based software engineering. In: Proceedings of the ACM/IEEE 2nd International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering. Lisbon, Portugal, pp. 7–12.

Andrade, R., Torres, J., Ortiz-Garcés, I., 2025. Enhancing security in software design patterns and antipatterns: A framework for LLM-based detection. Electronics 14 (3), 586.

Asare, O., Nagappan, M., Asokan, N., 2023. Is GitHub's copilot as bad as humans at introducing vulnerabilities in code? Empir. Softw. Eng. 28 (6), 129.

Bera, P., Wautelet, Y., Poels, G., 2023. On the use of ChatGPT to support agile software development. In: Proceedings of the 2nd International Workshop on Agile Methods for Information Systems Engineering. Zaragoza, Spain, pp. 1–9.

Cordeiro, J., Noei, S., Zou, Y., 2024. An empirical study on the code refactoring capability of large language models. arXiv preprint arXiv:2411.02320.

Dakhel, A.M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M.C., Jiang, Z.M.J., 2023. GitHub Copilot AI pair programmer: Asset or liability? J. Syst. Softw. 203, 111734.

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., Zhang, J.M., 2023. Large language models for software engineering: Survey and open problems. In: Proceedings of IEEE/ACM International Conference on Software Engineering: Future of Software Engineering. Melbourne, Australia, pp. 31–53.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, MA.

Hassan, A.E., Oliva, G.A., Lin, D., Chen, B., Ming, Z., et al., 2024. Rethinking software engineering in the foundation model era: From task-driven AI copilots to goal-driven AI pair programmers. arXiv preprint arXiv:2404.10225.

Kim, D.-K., 2025a. Design pattern applications. Available at: https://github.com/hanbyul1/Design-Pattern-Applications.

Kim, D.-K., 2025b. Improving software development traceability with structured prompting. J. Comput. Inf. Syst. 1–21.

Kim, D.-K., Chen, J., Ming, H., Lu, L., 2023. Assessment of ChatGPT's proficiency in software development. In: Proceedings of the 21st International Conference on Software Engineering Research & Practice.

Kirinuki, H., Tanno, H., 2024. ChatGPT and human synergy in Black-Box testing: A comparative analysis. Online: https://arxiv.org/abs/2401.13924.

Liu, B., Jiang, Y., Zhang, Y., Niu, N., Li, G., Liu, H., 2024. An empirical study on the potential of LLMs in automated software refactoring. arXiv preprint arXiv:2411.04444.

Marques, N., Silva, R.R., Bernardino, J., 2024. Using ChatGPT in software requirements engineering: A comprehensive review. Futur. Internet 16 (6), 180.

Martin, R.C., 1996. The dependency inversion principle. C++ Rep. 8 (6), 61–66.

Nguyen-Duc, A., Cabrero-Daniel, B., Przybylek, A., Arora, C., Khanna, D., Herda, T., Rafiq, U., Melegati, J., Guerra, E., Kemell, K.-K., et al., 2023. Generative artificial intelligence for software engineering–A research agenda. arXiv preprint arXiv:2310.18648.

Ozkaya, I., 2023. Application of large language models to software engineering tasks: Opportunities, risks, and implications. IEEE Softw. 40 (3), 4–8.

Pan, Z., Song, X., Wang, Y., Cao, R., Li, B., Li, Y., Liu, H., 2025. Do code LLMs understand design patterns?. arXiv preprint arXiv:2501.04835.

Pandey, S.K., Chand, S., Horkoff, J., Staron, M., Ochodek, M., Durisic, D., 2025. Design pattern recognition: a study of large language models. Empir. Softw. Eng. 30 (3), 69.

Pomian, D., Bellur, A., Dilhara, M., Kurbatova, Z., Bogomolov, E., Bryksin, T., Dig, D., 2024. Next-generation refactoring: Combining llm insights and ide capabilities for extract method. In: 2024 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 275–287.

Pudari, R., Ernst, N.A., 2023. From Copilot to Pilot: Towards AI supported software development. arXiv preprint arXiv:2303.04142.

Rajbhoj, A., Somase, A., Kulkarni, P., Kulkarni, V., 2024. Accelerating software development using generative AI: ChatGPT case study. In: Proceedings of the 17th Innovations in Software Engineering Conference. Bangalore, India, pp. 1–11.

Schindler, C., Rausch, A., 2025. LLM-based design pattern detection. arXiv preprint arXiv:2502.18458.

Solohubov, I., Moroz, A., Tiahunova, M.Y., Kyrychek, H.H., Skrupsky, S., 2023. Accelerating software development with AI: Exploring the impact of ChatGPT and GitHub copilot. In: Proceedings of the 11th Workshop on Cloud Technologies in Education. Kryvyi Rih, Ukraine, pp. 76–86.

Supekar, V., Khande, R., 2024. Improving software engineering practices: AI-driven adoption of design patterns. In: The 2nd International Conference on Advanced Computing & Communication Technologies (ICACCTech). pp. 768–774.

Surameery, N.M.S., Shakor, M.Y., 2023. Use chat GPT to solve programming bugs. Int. J. Inf. Technol. Comput. Eng. 3 (01), 17–22.

Wang, Y., Chen, Y., Li, Z., Tang, Z., Guo, R., Wang, X., Wang, Q., Zhou, A.C., Chu, X., 2024. Towards efficient and reliable LLM serving: A real-world workload study. arXiv preprint arXiv:2401.17644.

Waseem, M., Das, T., Ahmad, A., Liang, P., Fahmideh, M., Mikkonen, T., 2024. ChatGPT as a software development Bot: A project-based study. In: Proceedings of International Conference on Evaluation of Novel Approaches to Software Engineering. Angers, France.

White, J., Hays, S., Fu, Q., Spencer-Smith, J., Schmidt, D.C., 2023. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. Online: https://arxiv.org/abs/2303.07839.

Zhang, Q., Zhang, T., Zhai, J., Fang, C., Yu, B., Sun, W., Chen, Z., 2024. A critical review of large language model on software engineering: An example from ChatGPT and automated program repair. arXiv preprint arXiv:2310.08879.