Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.062552

ARTICLE

Quantitative Assessment of Generative Large Language Models on Design Pattern Application

Dae-Kyoo Kim*

Department of Computer Science and Engineering, Oakland University, 115 Library Dr., Rochester, MI 48309, USA

*Corresponding Author: Dae-Kyoo Kim. Email: kim2@oakland.edu

Received: 20 December 2024; Accepted: 01 February 2025; Published: 06 March 2025

ABSTRACT: Design patterns offer reusable solutions for common software issues, enhancing quality. The advent of generative large language models (LLMs) marks progress in software development, but their efficacy in applying design patterns is not fully assessed. The recent introduction of generative large language models (LLMs) like ChatGPT and CoPilot has demonstrated significant promise in software development. They assist with a variety of tasks including code generation, modeling, bug fixing, and testing, leading to enhanced efficiency and productivity. Although initial uses of these LLMs have had a positive effect on software development, their potential influence on the application of design patterns remains unexplored. This study introduces a method to quantify LLMs' ability to implement design patterns, using Role-Based Metamodeling Language (RBML) for a rigorous specification of the pattern's problem, solution, and transformation rules. The method evaluates the pattern applicability of a software application using the pattern's problem specification. If deemed applicable, the application is input to the LLM for pattern application. The resulting application is assessed for conformance to the pattern's solution specification and for completeness against the pattern's transformation rules. Evaluating the method with ChatGPT 4 across three applications reveals ChatGPT's high proficiency, achieving averages of 98% in conformance and 87% in completeness, thereby demonstrating the effectiveness of the method. Using RBML, this study confirms that LLMs, specifically ChatGPT 4, have great potential in effective and efficient application of design patterns with high conformance and completeness. This opens avenues for further integrating LLMs into complex software engineering processes.

KEYWORDS: Design patterns; large language models; pattern application; pattern-based refactoring; quantitative assessment

1 Introduction

Design patterns [1] offer proven solutions for recurring design problems in software development, enhancing software quality such as reusability, maintainability, and scalability. In practice, design patterns are used by interpreting their abstract description in the context of the application under development. However, the abstract nature of pattern descriptions can make it difficult to have a clear interpretation in their application, which might lead to obstacles in attaining the expected benefits of the pattern [2].

The recent advent of generative large language models (LLMs) such as ChatGPT [3], Gemini [4], and CoPilot [5] has shown great potential in software development, providing support in various tasks such as code generation [6], modeling [7], bug fixing [8], and testing [9], which leads to improved efficiency and productivity. While the initial use of these LLMs indicate a positive impact on software development [10], their potential on design pattern application has not been explored.

In this work, we present a quantifiable approach to evaluate the capability of LLMs in applying design patterns, focusing on pattern conformance and completeness. A prerequisite for this approach is the rigorous specification of design patterns, which is necessary to define precise pattern properties, serving as a quantitative measure, while facilitating checking the presence of pattern properties in UML models, which are the representation of programs used in this work to evaluate pattern conformance. There have been several techniques for specifying design patterns, which can be categorized into formal methods-based approaches and UML-based approaches. Formal methods-based approaches (e.g., [11–14]) make use of formal specification techniques to specify design patterns. While these techniques have, by virtue of formalism, strong support for reasoning and verifying pattern properties, it is difficult to use formalized pattern properties in checking their presence in UML models. There have also been efforts (e.g., [15–17]) to specify design patterns using the UML, a widely accepted modeling language. A major benefit of these approaches is that because of the wide acceptance of the UML, these approaches can be easily adopted. However, these approaches suffer from high complexity in representation. In this work, we adopt Role-Based Metamodeling Language (RBML) [18], a UML-based pattern specification technique, to formalize design patterns. RBML defines a design pattern in terms of roles, which capture pattern participants in a way simliar to UML models, which facilitates the evaluation of pattern conformance in this work.

In this work, we present a quantifiable approach to evaluating the capability of large language models (LLMs) in applying design patterns, focusing on pattern conformance and completeness. A key prerequisite for this approach is the rigorous specification of design patterns, which enables the definition of precise pattern properties that serve as quantitative measures and facilitate the evaluation of pattern conformance. This is achieved by checking for the presence of these properties in UML models, which represent the programs under evaluation. Various techniques have been proposed for specifying design patterns, broadly categorized into formal methods-based approaches and UML-based approaches. Formal methods-based approaches (e.g., [11–14]) use formal specification techniques, providing strong support for reasoning and verifying pattern properties. However, applying these formalized pattern properties to check their presence in UML models proves challenging. On the other hand, UML-based approaches (e.g., [15–17]) specify design patterns using UML, a widely accepted modeling language. While these approaches are more accessible due to the popularity of UML, they often suffer from high representational complexity. In this work, we adopt the Role-Based Metamodeling Language (RBML) [18], a UML-based pattern specification technique, to formalize design patterns. RBML defines design patterns in terms of roles, capturing pattern participants in a manner similar to UML models, thereby simplifying the evaluation of pattern conformance and facilitating analysis.

We define a design pattern in terms of problem specification, solution specification, and transformation rules. The problem specification is used to check the pattern applicability of the program under consideration. If the pattern is applicable, the program is input into the LLM, which applies the pattern and produces a refactored version. This refactored program is then evaluated for conformance to the applied pattern using the pattern's solution specification and checked for the completeness of pattern realization against the pattern's transformation rules. We evalaute the approach using the Visitor pattern applied to three case studies in ChatGPT 4. The evaluation results show that ChatGPT can apply design patterns with an average of 98% pattern conformance and 87% pattern completeness, demonstrating the effectiveness of the approach.

The remainder of the paper is organized as follows: Section 2 discusses the relevant literature on utilizing LLMs in software engineering. Section 3 provides an overview of RBML using the Visitor pattern as an example. Section 4 details the proposed approach, illustrating the application of the Visitor pattern to a software application in ChatGPT. Section 5 presents two additional case studies in which the Visitor pattern is applied to other applications. Finally, Section 6 concludes the study and discusses avenues for future work.

2 Related Work

In this section, we review relevant work on evaluating LLMs in software engineering.

Several studies have examined LLMs' capabilities in general software development and code synthesis. Kim et al. [10] assessed ChatGPT's proficiency across various development phases, finding it could generate over 90% of code while noting limitations in traceability. White et al. [19] developed systematic prompt design techniques, introducing patterns for requirements elicitation and code quality. Dakhel et al. [20] evaluated GitHub Copilot's capabilities in algorithmic problem-solving, finding effective but sometimes inconsistent solutions. Solohubov et al. [21] demonstrated significant efficiency gains with AI tools, while Nascimento et al. [22] compared ChatGPT against human programmers on Leetcode, revealing varying performance across different problem types.

Research on code quality and maintenance has produced significant findings. Zhang et al. [8] evaluated ChatGPT's bug-fixing capabilities, showing successful repair patterns through various prompting strategies. In a follow-up study, Zhang et al. [8] introduced EvalGPTFix, a benchmark for assessing LLM-based program repair. Kirinuki et al. [9] found ChatGPT generated test cases comparable to human testers, though with limitations in boundary testing. Surameery et al. [23] explored ChatGPT's debugging capabilities, while Asare et al. [24] investigated Copilot's potential for introducing vulnerabilities. Alshahwan et al. [25] proposed an assured LLMSE approach using semantic filters to validate code changes.

A significant body of work has focused on software architecture and requirements engineering. Ahmad et al. [7] explored ChatGPT's role in architecture-centric software engineering, developing frameworks for requirements articulation and microservices architecture design. Marques et al. [26] evaluated ChatGPT's effectiveness in requirements engineering, highlighting improved stakeholder communication. Rajbhoj et al. [27] investigated integrating generative AI across the software development lifecycle, while Ozkaya [6] discussed both benefits and risks of LLMs in software engineering tasks.

Studies have examined LLMs' impact on software development processes and methodologies. Bera et al. [28] assessed ChatGPT's capability as an agile coach, recommending cautious integration into teams. Felizardo et al. [29] demonstrated ChatGPT's potential in systematic literature reviews, while Özpolat et al. [30] investigated its role in automating development tasks. Champa et al. [31] analyzed the DevGPT dataset [32] to understand how developers utilize ChatGPT in practice.

Recent research has also explored collaborative and educational aspects of LLMs. Hassan et al. [33] proposed developing AI pair programmers that work contextually with human developers. Pudari et al. [34] analyzed Copilot's adherence to programming best practices. Waseem et al. [35] investigated ChatGPT's effectiveness in helping students understand software development tasks while warning against over-reliance.

Several comprehensive studies have examined broader implications and practical applications. Fan et al. [36] surveyed LLM applications across software engineering domains. Nguyen-Duc et al. [37] identified 78 research questions across 11 areas in generative AI for software engineering. Rahmaniar [38] discussed ChatGPT's potential to enhance software engineering efficiency, while Wang et al. [39] introduced BurstGPT, a dataset capturing real-world LLM usage patterns. Sridhara et al. [40] evaluated ChatGPT's performance across fifteen distinct software engineering tasks, finding varying effectiveness across different types of activities.

While these studies provide valuable insights into LLMs' potential across various software engineering tasks, they highlight both opportunities and limitations in areas such as code generation, bug fixing, testing, and architectural design. However, the systematic application of design patterns—a crucial aspect of software engineering—remains unexplored. Our work addresses this gap by providing a quantitative framework for evaluating LLMs' pattern application abilities, using RBML for rigorous pattern specification, and demonstrating practical effectiveness through multiple case studies.

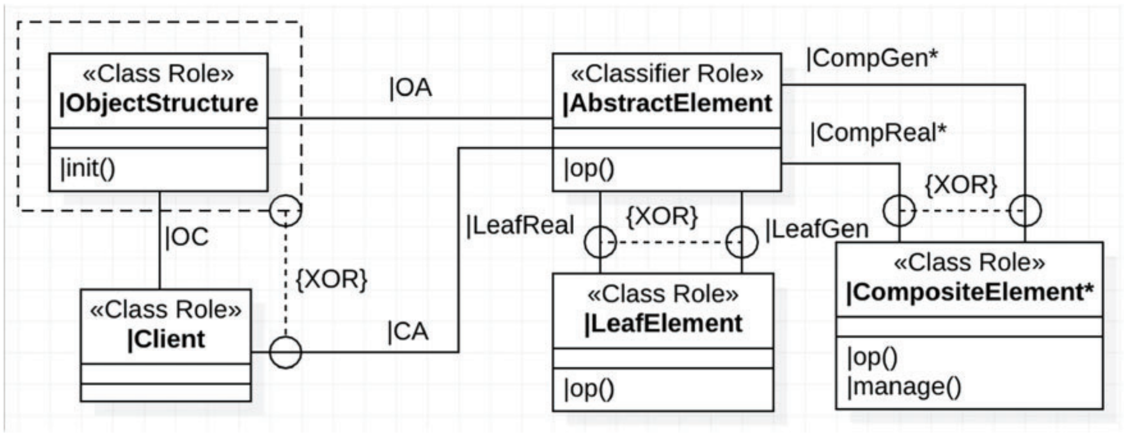
3 Design Pattern Specifications in RBML

RBML is a UML-based notation for specifying design patterns at the meta-model level, supporting their application at the model level [18]. RBML captures the variability of design patterns through roles which are enacted by UML model elements. Each role defines a set of properties that the model elements must exhibit to comply with the role. Roles are established based on a UML metaclass, representing a subset of the instances of that base metaclass.

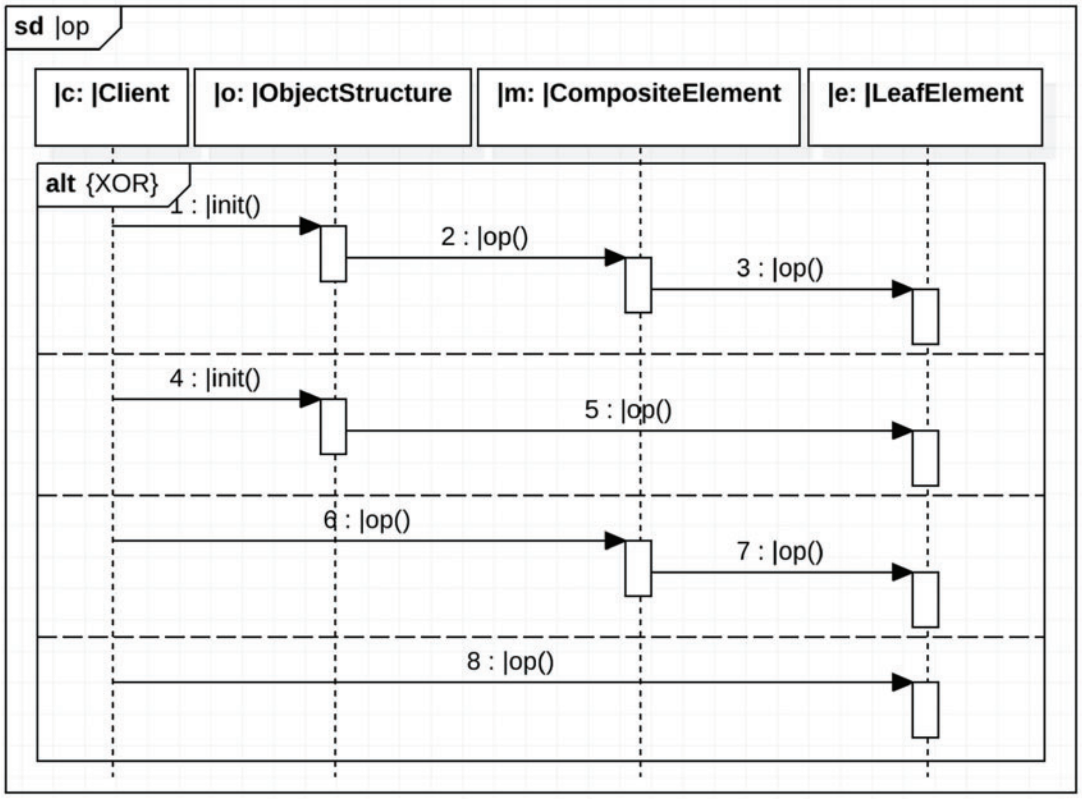
RBML allows for the rigorous specification of design patterns, facilitating the quantitative assessment of pattern realizations. RBML pattern specifications are defined in terms of problem specification, solution specification, and transformation rules. The problem specification captures the problem domain of the pattern, defining the applicability of the pattern. A software application is considered pattern applicable if it satisfies the properties of the problem specification. The solution specification describes the pattern's solution domain, establishing criteria for conformance to the pattern. A software model is deemed to be conformant to the pattern if it satisfies the properties of the solution specification. Transformation rules define the process of refactoring a problem model into a solution model based on the mapping between the problem specification and the solution specification.

A pattern specification is defined by roles that encapsulate the pattern's variability. A single role or a combination of roles constitutes a pattern property, which extends the capacity to capture the pattern's variability, and a pattern specification may comprise multiple such properties. An application model is deemed to satisfy the pattern specification if it adheres to all the defined properties. There are two types of pattern specifications—a Structural Pattern Specification (SPS) that delineates the pattern's structural properties, and an Interaction Pattern Specification (IPS) that outlines the pattern's behavioral properties.

In this study, we employ the Visitor pattern [1] to illustrate our approach. This pattern is selected due to its complexity among the Gang of Four (GoF) patterns, making it an ideal candidate to demonstrate the capabilities of LLMs in applying design patterns. Fig. 1 shows the problem specification for the Visitor pattern. The SPS defines the structure of the pattern's problem domain, including roles such as $|ObjectStructure$, $|AbstractElement$, $|Client$, $|LeafElement$, and $|CompositeElement$. Each role is linked to a base metaclass, specified within <<>> above the role name. $\{XOR\}$ in the diagram specifies exclusive-or constraints, illustrating the structural variability of the pattern's problem domain. The $\{XOR\}$ constraint between the $|ObjectStructure$ block (which includes its relationship role $|OC$ with the $|Client$ role and relationship role $|OA$ with the $|AbstractElement$ role) and the relationship role $|CA$ indicates two variations—(i) the $|Client$ is associated with the $|AbstractElement$ via $|ObjectStructure$, or (ii) the $|Client$ is directly associated with the $|AbstractElement$ without an intervening object structure. The $\{XOR\}$ constraint between the $|LeafReal$ and $|LeafGen$ roles stipulates that only one of these roles should be realized. Similarly, the $\{XOR\}$ constraint between $|CompReal$ and $|CompGen$ enforces the same logic. The asterisk (*) symbol next to the $|CompReal$ role, the $|CompGen$ role, and the $|CompositeElement$ indicates that these roles are optional, introducing additional variability. Specifically, it implies that the presence of composite elements is not mandatory.



(a) SPS

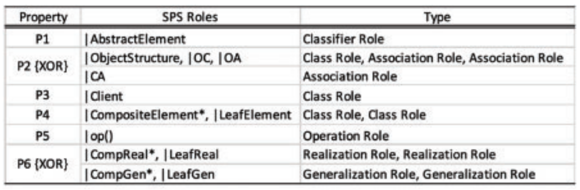


(b) IPS

Figure 1: Visitor problem specification

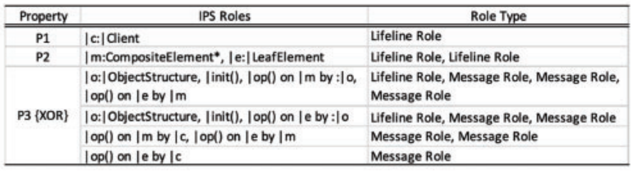
The IPS details the interaction behaviors within the pattern's problem domain, represented by lifeline roles assumed by objects of classes fulfilling SPS roles. For example, the $|o : |ObjectStructure$ lifeline role is played by an object of the class fulfilling the $|ObjectStructure$ SPS role. The alt combined fragment with the $\{XOR\}$ constraint outlines four interaction variations aligned with the $\{XOR\}$ constraint near the $|Client$ role in the SPS. The first two cases specify the participation of the object structure in the interactions—the first case describes the object structure interacting with elements via composite elements, and the second case involves the object structure interacting directly with leaf elements when composite elements are absent. The remaining two cases delineate scenarios without the object structure—in the third case, the client interacts with elements via composite elements, while the fourth case describes the client interacting directly with leaf elements in the absence of composite elements.

A single role or a group of roles forms a pattern property. Fig. 2 illustrates the properties of the Visitor problem specification, where Fig. 2a displays SPS properties and Fig. 2b shows IPS properties. In the tables, the first column shows labels for properties, the second column specifies roles involved in the property, and the third column describes the base metaclasses for the involved roles. Six SPS properties are defined in Fig. 2a. P2 specifies two structural variations, as indicated by the $\{XOR\}$ constraint near $|Client$ in Fig. 1a. The property is satisfied if either of the two variations is realized. P4 is fulfilled if the $|LeafElement$ role, along with the optional $|CompositeElement*$ role, is present. P6 addresses another $\{XOR\}$ constraint related to the two $\{XOR\}$ constraints under $|AbstractElement$ in Fig. 1a and is satisfied if either $(|CompReal, |LeafReal)$ or $(|CompGen*, |LeafGen)$ exists, where the first component is linked to the $\{XOR\}$ constraint over the relationships with $|CompositeElement$ and the second with $|LeafElement$. P1, P3, and P5, each involving only one role, cover the corresponding constraints of that role. For the IPS, three properties are defined in Fig. 2b. P3, detailing four $\{XOR\}$ cases in the alt fragment in Fig. 1b, is satisfied if any one case occurs. P2 is met if the $|e : |LeafElement$ role, along with the optional $|m : |CompositeElement*$ role, is achieved. P1, containing only a single role, is satisfied when that role is realized.



Property	SPS Roles	Type
P1	|AbstractElement	Classifier Role
P2 {XOR}	|ObjectStructure, |OC, |OA	Class Role, Association Role, Association Role
	|CA	Association Role
P3	|Client	Class Role
P4	|CompositeElement*, |LeafElement	Class Role, Class Role
P5	|op()	Operation Role
P6 {XOR}	|CompReal*, |LeafReal	Realization Role, Realization Role
	|CompGen*, |LeafGen	Generalization Role, Generalization Role

(a) SPS Properties



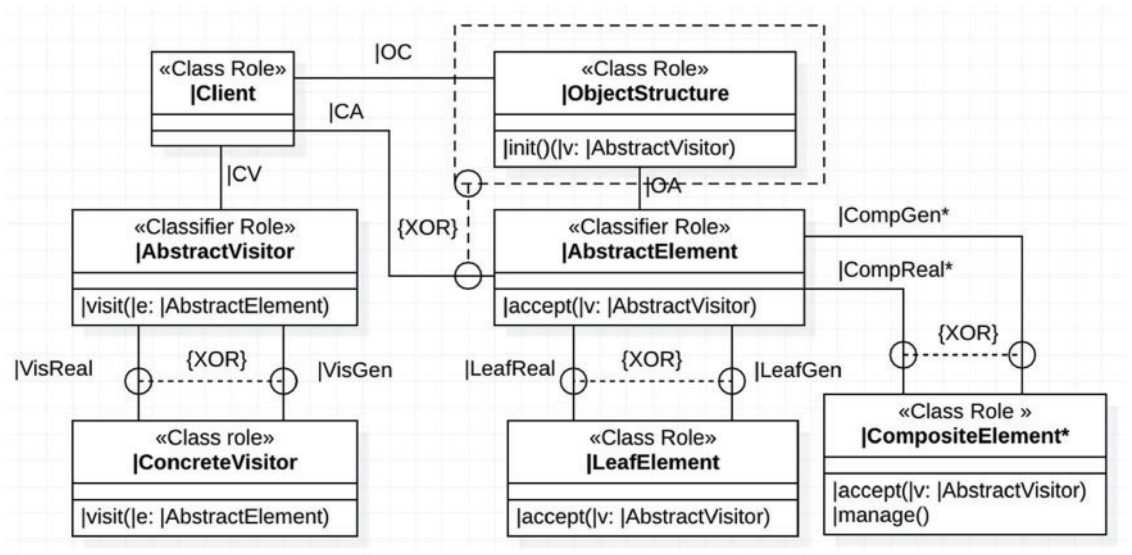
Property	IPS Roles	Role Type
P1	|c:|Client	Lifeline Role
P2	|m:CompositeElement*, |e:|LeafElement	Lifeline Role, Lifeline Role
P3 {XOR}	|o:|ObjectStructure, |init(), |op() on |m by :|o, |op() on |e by |m	Lifeline Role, Message Role, Message Role, Message Role
	|o:|ObjectStructure, |init(), |op() on |e by :|o	Lifeline Role, Message Role, Message Role
	|op() on |m by |c, |op() on |e by |m	Message Role, Message Role
	|op() on |e by |c	Message Role

(b) IPS Properties

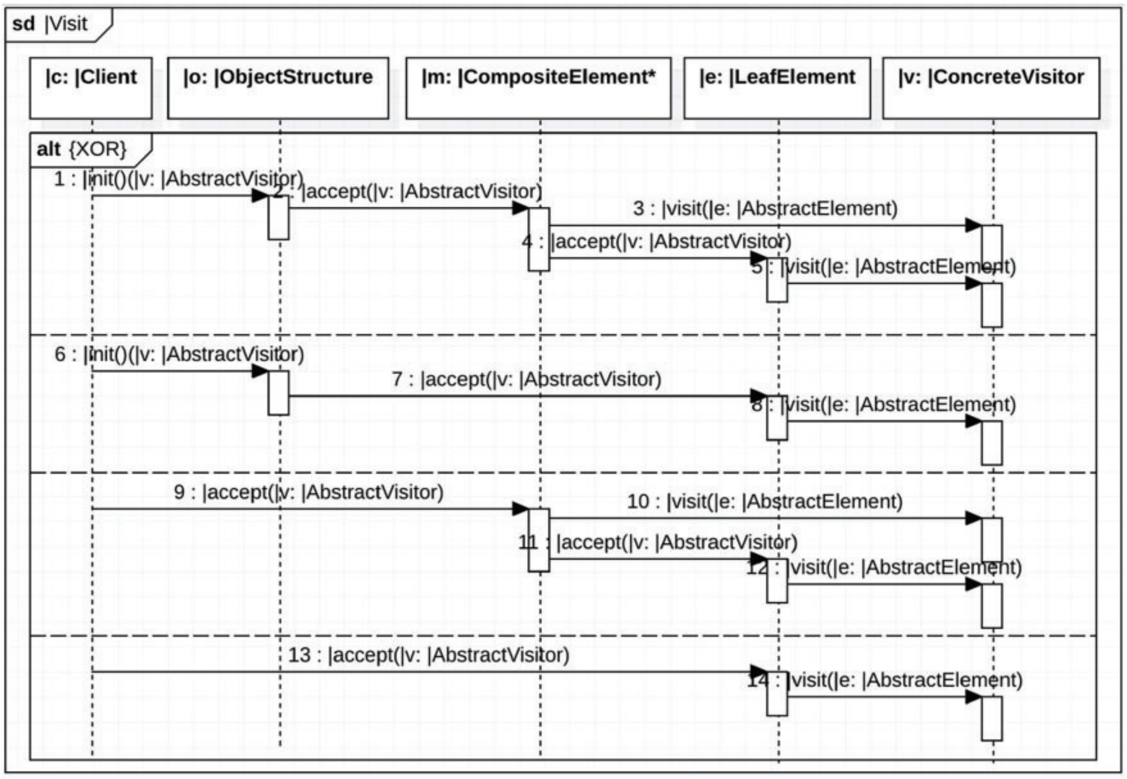
Figure 2: Visitor problem properties

Fig. 3 illustrates the solution specification of the Visitor pattern. The SPS defines the solution structure which encompasses $|AbstractVisitor$ and $|ConcreteVisitor$, along with the roles specified in the problem SPS. $\{XOR\}$ in the diagram specifies exclusive-or constraints, illustrating the structural variability of the pattern's solution domain. The IPS delineates the solution behaviors of the pattern through four variations specified in the alt fragment. Each variation describes the double dispatch mechanism where the client or the object structure conveys the visitor through the $|accept()$ message sent to an individual component, which in turn, invites the visitor by passing itself via the $visit()$ message sent to the visitor.

Fig. 4 illustrates the solution properties of the Visitor pattern, with Fig. 4a detailing the SPS properties and Fig. 4b outlining the IPS properties. In Fig. 4a, properties P2, P6, P7, and P8 relate to specific visitor-related aspects, while P13 defines the $\{XOR\}$ constraint governing the relationships between $|AbstractVisitor$ and $|ConcreteVisitor$, as depicted in Fig. 3a. In Fig. 4b, P3 identifies the participating visitor, and P4 details the four variations of double-dispatch interactions, as specified in the alt fragment in Fig. 3b.

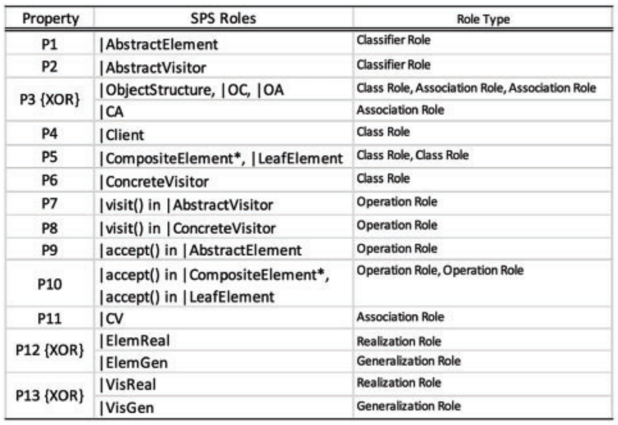


(a) SPS


sd |Visit
|c: |Client
|o: |ObjectStructure
|m: |CompositeElement*
|e: |LeafElement
|v: |ConcreteVisitor
alt {XOR}
1 : |init()(|v: |AbstractVisitor)
2 : |accept(|v: |AbstractVisitor)
3 : |visit(|e: |AbstractElement)
4 : |accept(|v: |AbstractVisitor)
5 : |visit(|e: |AbstractElement)
6 : |init()(|v: |AbstractVisitor)
7 : |accept(|v: |AbstractVisitor)
8 : |visit(|e: |AbstractElement)
9 : |accept(|v: |AbstractVisitor)
10 : |visit(|e: |AbstractElement)
11 : |accept(|v: |AbstractVisitor)
12 : |visit(|e: |AbstractElement)
13 : |accept(|v: |AbstractVisitor)
14 : |visit(|e: |AbstractElement)

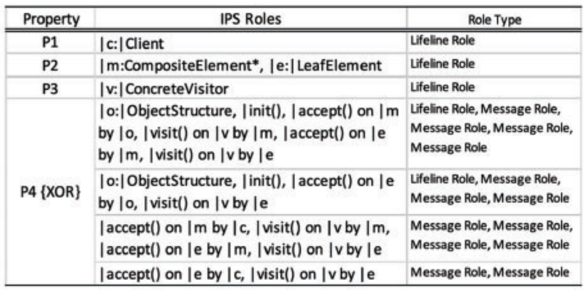
(b) IPS

Figure 3: Visitor solution specification



Property	SPS Roles	Role Type
P1	|AbstractElement	Classifier Role
P2	|AbstractVisitor	Classifier Role
P3 {XOR}	|ObjectStructure, |OC, |OA	Class Role, Association Role, Association Role
	|CA	Association Role
P4	|Client	Class Role
P5	|CompositeElement*, |LeafElement	Class Role, Class Role
P6	|ConcreteVisitor	Class Role
P7	|visit() in |AbstractVisitor	Operation Role
P8	|visit() in |ConcreteVisitor	Operation Role
P9	|accept() in |AbstractElement	Operation Role
P10	|accept() in |CompositeElement*, |accept() in |LeafElement	Operation Role, Operation Role
P11	|CV	Association Role
P12 {XOR}	|ElemReal	Realization Role
	|ElemGen	Generalization Role
P13 {XOR}	|VisReal	Realization Role
	|VisGen	Generalization Role

(a) SPS Properties



Property	IPS Roles	Role Type
P1	|c:|Client	Lifeline Role
P2	|m:CompositeElement*, |e:|LeafElement	Lifeline Role
P3	|v:|ConcreteVisitor	Lifeline Role
P4 {XOR}	|o:|ObjectStructure, |init(), |accept() on |m by |o, |visit() on |v by |m, |accept() on |e by |m, |visit() on |v by |e	Lifeline Role, Message Role, Message Role, Message Role, Message Role
	|o:|ObjectStructure, |init(), |accept() on |e by |o, |visit() on |v by |e	Lifeline Role, Message Role, Message Role, Message Role
	|accept() on |m by |c, |visit() on |v by |m, |accept() on |e by |m, |visit() on |v by |e	Message Role, Message Role, Message Role, Message Role
	|accept() on |e by |c, |visit() on |v by |e	Message Role, Message Role

(b) IPS Properties

Figure 4: Visitor solution properties

The problem specification and solution specification of a pattern are mapped to establish transformation rules. These rules define the necessary conditions that must be met during the transformation from a problem model to a solution model when the pattern is applied. Let $\widetilde{e}$ represent an element e in the problem domain, and let $\widehat{e}$ represent a corresponding element e in the solution domain. Then, the following mapping is defined between the problem SPS and the solution SPS of the Visitor pattern:

$[|\widetilde{Client} \mapsto |\widehat{Client}$,

$|\widetilde{ObjectStructure} \mapsto |\widehat{ObjectStructure}$,

$|\widetilde{AbstractElement} \mapsto |\widehat{AbstractElement}$,

$|\widetilde{CompositeElement} \mapsto |\widehat{CompositeElement}$,

$|\widetilde{LeafElement} \mapsto |\widehat{LeafElement}]$

Based on the mapping, let R represent the set of elements that fulfill the role $|R$. Then, $\widetilde{op()}$ is the set of operations fulfilling the role $|\widetilde{op()}$, and $\widehat{visit()}$ is the set of operations fulfilling the role $|\widehat{visit()}$. The function $getVisit(c)$ produces a visit operation $v \in \widehat{visit()}$ for class c. Similarly, $getOperations(c)$ returns the set of operations in class c, and $getConcreteVisitor(o)$ provides a concrete visitor $cv \in \widehat{ConcreteVisitor}$ associated with operation o. Using these functions, we define the following SPS transformation rules:

S1. For every concrete element $e \in \widetilde{CompositeElement} \cup \widetilde{LeafElement}$, there exists a corresponding visit operation $o \in \widehat{visit()}$ in every abstract visitor class $v \in \widehat{AbstractVisitor}$.
$\forall e : \widetilde{\text{CompositeElement}} \cup \widetilde{\text{LeafElement}}, \exists o : \widehat{\text{visit()}} \cdot$
$o = getVisit(e) \wedge (\forall v : \widehat{\text{AbstractVisitor}} \cdot o \in \text{getOperations}(\text{v}))$

S2. For every operation $p \in \widetilde{op()}$, there exists a corresponding concrete visitor class $c \in \widehat{ConcreteVisitor}$.
$\forall p : \widetilde{\text{op}}(), \exists c : \widehat{\text{ConcreteVisitor}} \cdot c = getConcreteVisitor(p)$

S3. The operations of $\widetilde{op()}$ no longer exist in classes of $\widehat{AbstractElement}$ and $\widehat{ConcreteElement}$.
$\forall p : \widetilde{\text{op}}(), \forall e : \widehat{\text{AbstractElement}} \cup \widehat{\text{ConcreteElement}} \cdot p \notin getOperations(e)$

For IPS transformation, the following defines the mapping between problem IPS roles and solution IPS roles:

$[|c : \widetilde{|Client} \mapsto |c : \widehat{|Client}$,

$|o : |\widetilde{ObjectStructure} \mapsto |o : |\widehat{ObjectStructure}$,

$|m : |\widetilde{CompositeElement} \mapsto |m : |\widehat{CompositeElement}$,

$|e : |\widehat{LeafElement} \mapsto |e : |\widehat{LeafElement}]$

Let us denote $|m()[: |caller, : |callee]$ as a message role where $: |caller$ calls $|m()$ on $: |callee$. Then, $m()[caller, callee]$ is the set of messages playing the $|m()[: |caller, : |callee]$ role. In the context of IPS mapping, $|\widetilde{op()}$ represents the union of message sets $|op()[|o, |m] \cup |op()[|m, |e] \cup |op()[|o, |e] \cup |op()[|c, |m] \cup |op()[|m, |e] \cup |op()[|c, |e]$. Similarly, $|\widehat{accept()}$ represents the union of $|accept()[|o, |m] \cup |accept()[|m, |e] \cup |accept()[|o, |e] \cup |accept()[|c, |m] \cup |accept()[|m, |e] \cup |accept()[|c, |e]$. $seq(s)$ denotes the set of all elements for sequence diagram s. The function $match(\widehat{m1}, \widetilde{m2})$ returns true if the caller and callee of $m1$ play the roles that correspond to the caller and callee of $m2$. With these definitions, we can establish the following IPS transformation rules:

I1. For every message $op \in \widetilde{op()}$, there exists a corresponding message $accept \in \widehat{accept()}$ whose caller and callee matches those of op.
$\forall op : \widetilde{\mathrm{op}()}, \exists accept \in \widehat{accept()} \cdot match(op, accept)$

I2. The messages of $\widetilde{op()}$ no longer exist in the solution sequence diagram $\widehat{s}$.
$\forall m : \widetilde{\mathrm{op}()} \cdot m \notin seq(\widehat{s})$

4 Quantifying LLMs' Capability on Design Pattern Application

In this section, we describe the proposed approach for quantitatively assessing LLMs in design pattern application using RBML pattern specifications. Fig. 5 illustrates the process where rectangles represent data, ellipses denote operations, diamonds indicate conditions, and arrows depict control flow. The process involves the following steps:

1. A program that does not yet implement the intended design pattern is considered, referred to as the problem program. To ensure a fair assessment, we focus on programs that are suitable candidates for the intended pattern application, as those not conforming to the pattern may not adequately demonstrate the LLM's full potential.
2. The problem program is reverse-engineered into a model, termed the problem model.
3. The problem model is checked for pattern applicability against the problem specification of the desired pattern. The degree of applicability is quantified as $\frac{p'}{p} \times 100$ where p' represents the number of satisfied problem properties and p denotes the total number of problem properties. Each problem property is assigned equal weight.
4. After confirming pattern applicability, the problem program is input into the LLM to apply the pattern with the following prompt:
`prompt: Apply [Target Pattern] to the given program.`
5. The resulting program from the LLM, to which the pattern has been applied, is reverse-engineered into what is termed the solution model.
6. The solution model is evaluated for its conformance to the pattern's solution specification. The degree of conformance is quantified as $\frac{s'}{s} \times 100$ where s' represents the number of satisfied solution properties and s denotes the total number of solution properties. Each solution property is assigned equal weight.
7. The solution model is checked for completeness regarding the pattern transformation rules. The degree of completeness is quantified as $\frac{t'}{t} \times 100$ where t' represents the number of satisfied transformation rules and t denotes the total number of transformation rules. Each transformation rule is assigned equal weight.

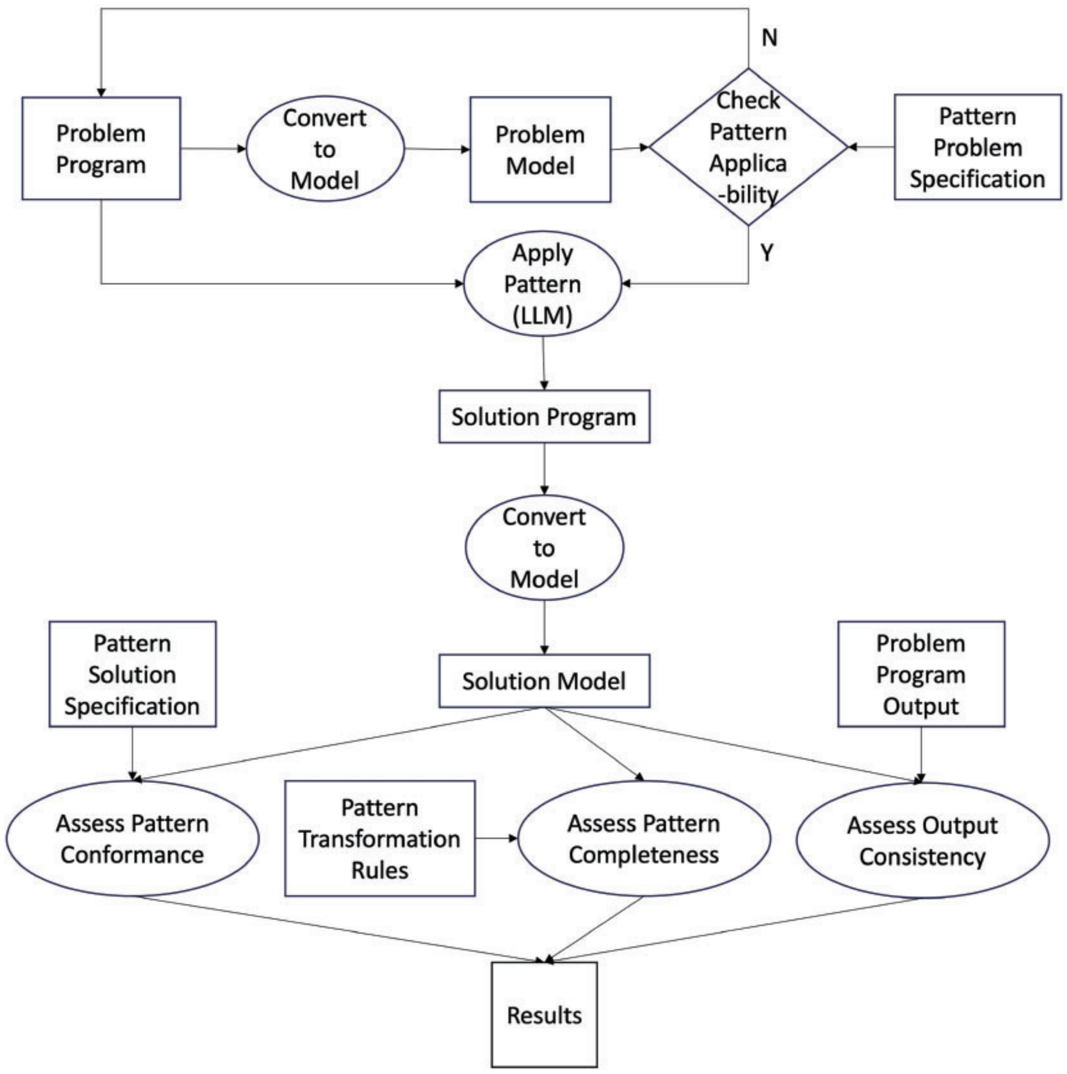

N
Problem Program
Convert to Model
Problem Model
Check Pattern Applica -bility
Pattern Problem Specification
Y
Apply Pattern (LLM)
Solution Program
Convert to Model
Pattern Solution Specification
Solution Model
Problem Program Output
Assess Pattern Conformance
Pattern Transformation Rules
Assess Pattern Completeness
Assess Output Consistency
Results

Figure 5: Quantitative assessment process

To demonstrate the approach, we use the Visitor pattern described in Section 3, applied to a drawing application, which is one of the three case studies conducted in this work. The other two case studies are presented in Section 5. The source code of the applications used in the case studies, both before and after pattern application, is available on GitHub [41]. The drawing application is designed for rendering a variety of graphical objects such as points, lines, rectangles, and composite images, which can be composed of multiple shapes. In a drawing session, each object is capable of being drawn individually, allowing for a complex assembly of shapes to create detailed pictures. The application's functionality caters to structuring and manipulating these objects to form a visual representation.

The application is reverse-engineered to derive its model which is used to check for pattern applicability. Fig. 6 illustrates the reverse-engineered model, referred to as the problem model. The model includes the $VisitorDrawingProblem$ class serving as the driver class, the $ObjectStructure$ class capturing the structure of the drawing objects, and a hierarchy of object classes such as $Picture$, $Rectangle$, $Line$, and $Point$. Composite relationships within this hierarchy allow for the creation of complex structures where a picture may contain other objects, including other pictures. The sequence diagram details the drawing behavior for rendering a picture that encompasses a line connected by two points. The client interacts with the object structure, invoking the $draw()$ operation recursively across the composite arrangement of elements.

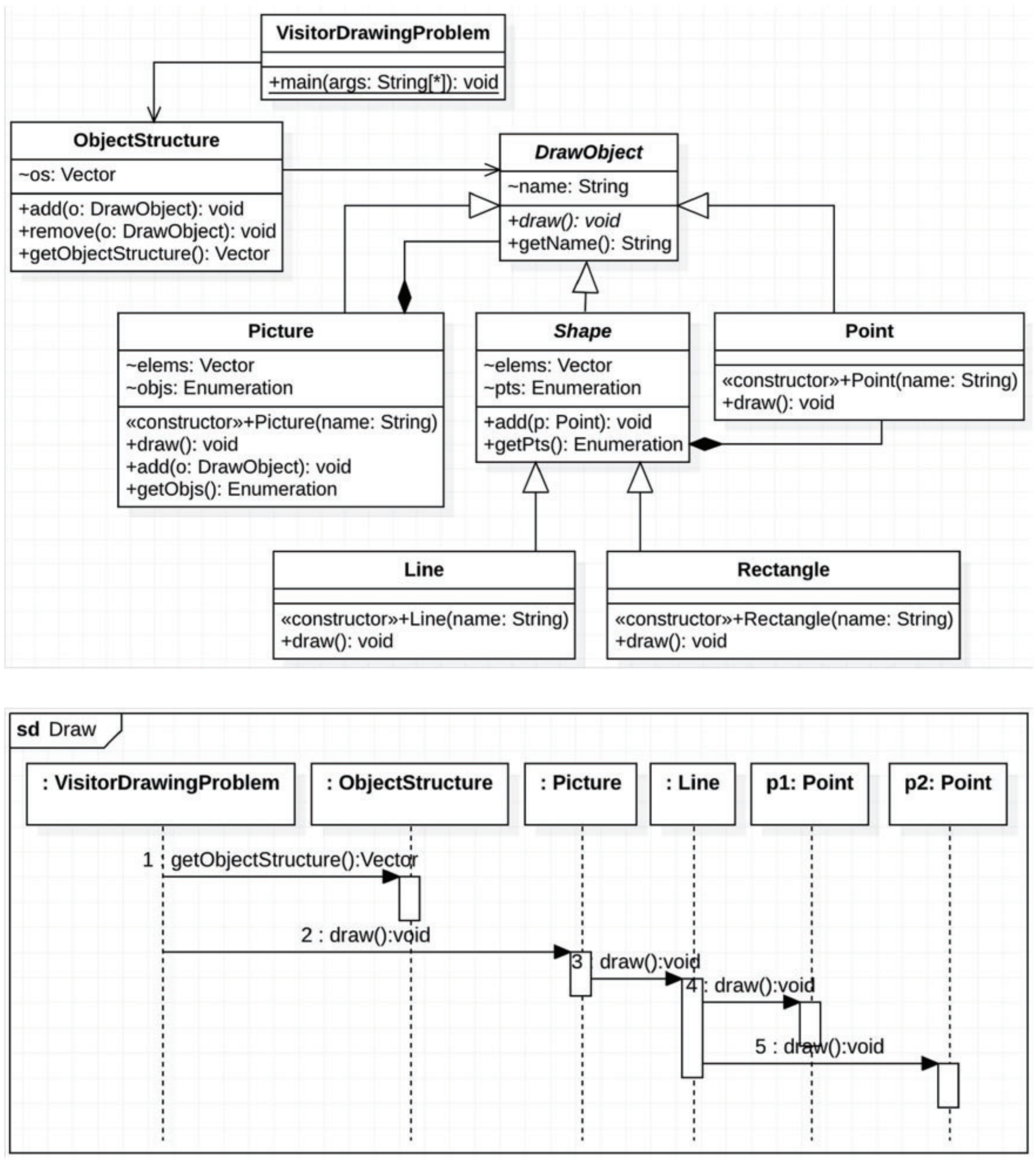


Figure 6: Drawing objects problem model

The problem model is evaluated for pattern applicability against the problem specification of the pattern. The model is considered pattern-applicable if it satisfies the properties defined in the problem specification. Pattern properties are specified in terms of roles and are evaluated based on the mapping of the roles to the model elements in the problem model. Fig. 7 presents the mapping of SPS roles to class diagram elements. In the mapping, the $|AbstractElement$ role corresponds to two abstract classes, namely $DrawObject$ and $Shape$, demonstrating the role's variability, where more than one class can fulfill the role. Similarly, the

$|CompositeElement$ role corresponds to three concrete classes, $Picture$, $Rectangle$, and $Line$, which are composite objects.

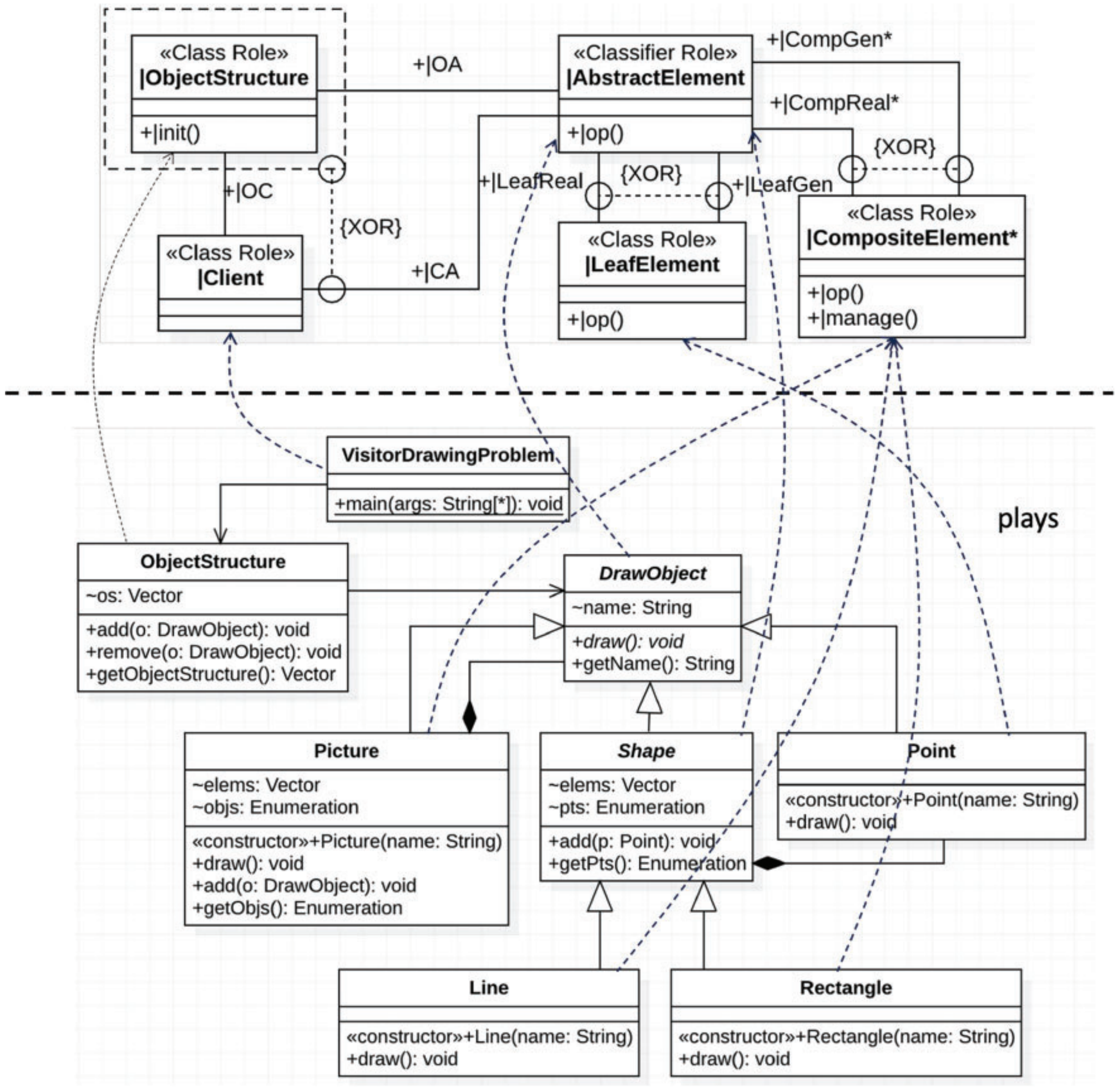


Figure 7: Visitor problem SPS mapping to drawing objects problem class diagram

Fig. 8 shows the mapping of IPS roles to sequence diagram elements. In the figure, the $|m:|CompositeElement$ role is fulfilled by two lifelines, $:Picture$ and $:Line$, demonstrating the role's variability. Similarly, the $|e:|LeafElement$ role is played by two other lifelines, $p1:Point$ and $p2:Point$. These mappings illustrate the third case in the alt fragment of the IPS. Note that although the $:ObjectStructure$ lifeline fulfills the $|o:|ObjectStructure$ role, the $getObjectStructure()$ operation in the $|ObjectStructure$ role does not delegate the $draw()$ message to drawing objects, rendering the first and second cases in the alt fragment invalid.

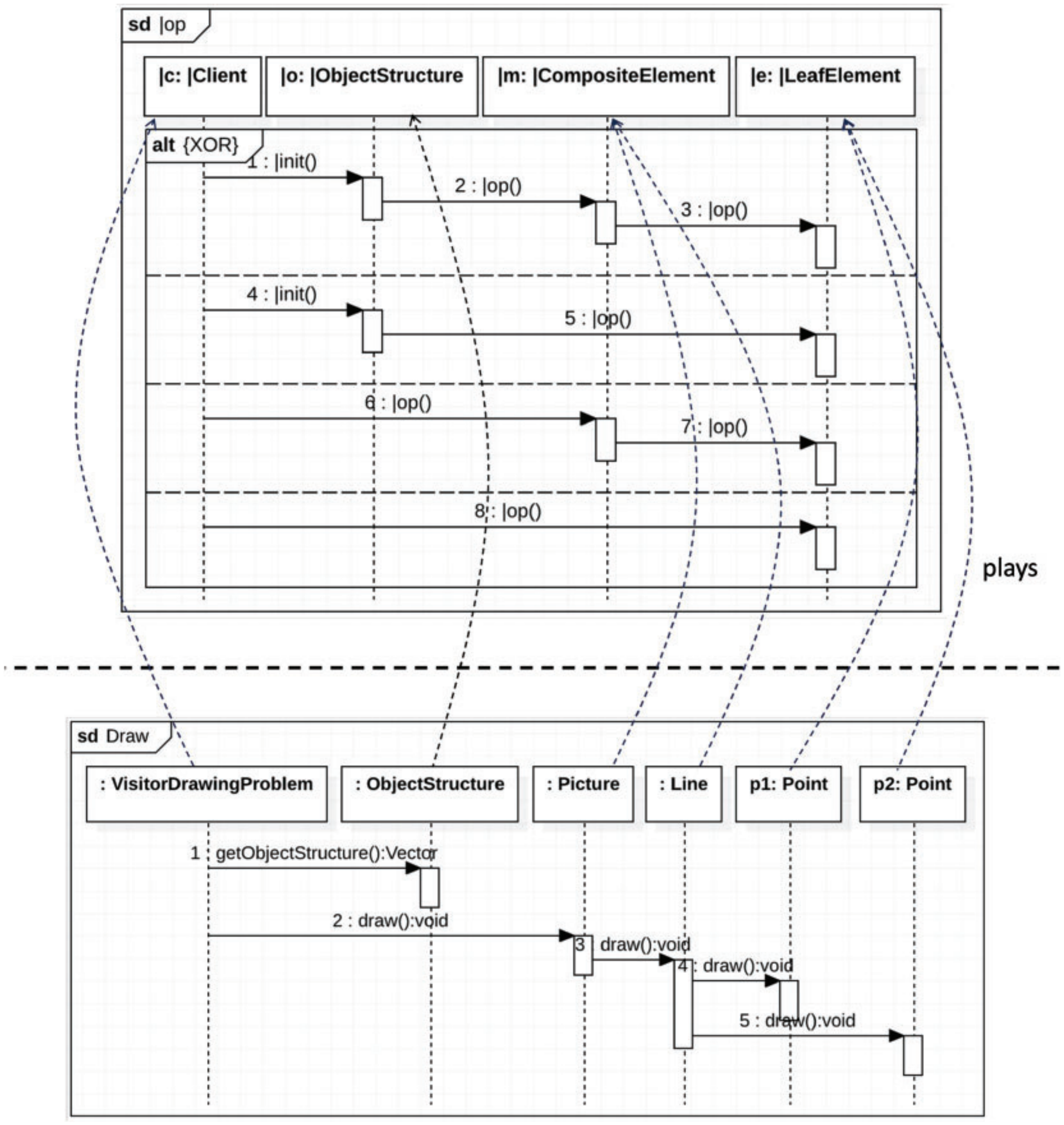
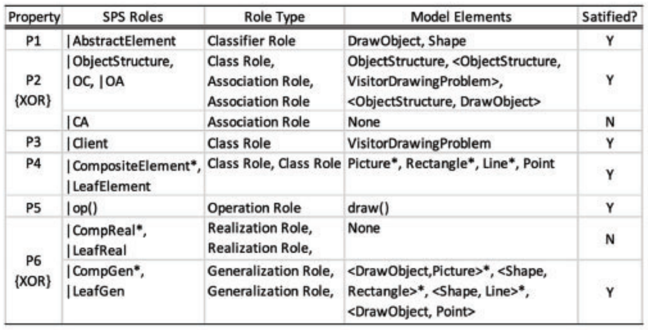


Figure 8: Visitor problem IPS mapping to *Draw* problem sequence diagram

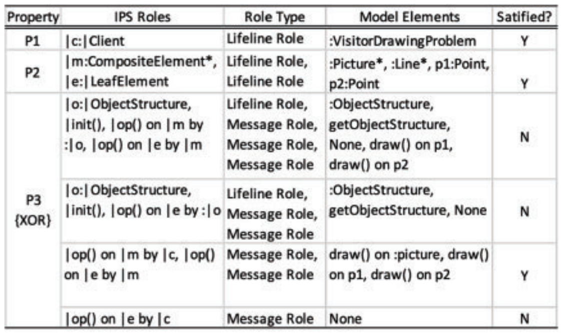
Based on the mappings in Figs. 7 and 8, the pattern applicability of the problem model is evaluated as depicted in Fig. 9. In the figure, pattern properties are evaluated based on the roles involved (SPS/IPS Roles), the base metaclasses of these roles, and the model elements that enact these roles. The base metaclass of an involved role dictates that model elements playing the role must be instances of the specified metaclass. If this condition is not met, the roles cannot be properly enacted, and consequently, the pattern property cannot be satisfied. Fig. 9a indicates that all six SPS properties are satisfied, demonstrating 100% SPS applicability. Property $P1$ involves the $|AbstractElement$ role, whose base type is the $Classifier$ metaclass, and is satisfied by the $DrawObject$ and $Shape$ classes which are instances of the $Classifier$ metaclass, denoted as Y. Property $P2$ involves two exclusive-or cases and is satisfied by the first case, which captures the relationship between the $VisitorDrawingProblem$ client and the $ObjectStructure$ class. This case includes the $|ObjectStructure$, $|OC$, and $|OA$ roles, whose base types are the $Class$, $Association$, and $Association$ metaclasses, respectively. The $|ObjectStructure$ role is fulfilled by the $ObjectStructure$ class, which is an instance of the $Class$ metaclass. The $|OC$ role is enacted by the <

$ObjectStructure, VisitorDrawingProblem >$ association, an instance of the $Association$ metaclass, and the $|OA$ role is fulfilled by the $< ObjectStructure, DrawObject >$ association, also an instance of the $Association$ metaclass. The second case does not satisfy the property as there are no model elements playing the $|CA$ role. Property $P3$ pertains to the $|Client$ role, whose base type is the $Class$ metaclass. This property is satisfied by the $VisitorDrawingProblem$ class, which is an instance of the $Class$ metaclass, and is therefore denoted as Y. Property $P4$ involves the $|CompositeElement^*$ and $|LeafElement$ roles, both of which have the base type of the $Class$ metaclass. The $|CompositeElement^*$ role is played by the $Picture^*$, $Rectangle^*$, and $Line^*$ classes, each an instance of the $Class$ metaclass. Similarly, the $|LeafElement$ role is enacted by the $Line$ class. Consequently, the property is satisfied, denoted as Y. Property $P5$ relates to the $|opt()$ role, which is associated with the $|Operation$ metaclass. This property is fulfilled by the $draw()$ operation, an instance of the $Operation$ metaclass, thereby denoted as Y. Property $P6$ involves two exclusive-or scenarios, with the second case being satisfied through the generalization hierarchy among elements, represented by the $|CompGen^*$ and $|LeafGen$ roles. Both roles are based on the $Generalization$ metaclass. The $|CompGen^*$ role is played by the generalization relationships $< DrawObject, Picture >^*$, $< Shape, Rectangle >^*$, and $< Shape, Line >^*$, each an instance of the $Generalization$ metaclass. The $|LeafGen$ role is satisfied by the $< DrawObject, Point >$ generalization relationship, also an instance of the $|Generalization$ metaclass. Fig. 9b demonstrates that all three IPS properties are satisfied, leading to 100% IPS applicability. Property $P1$ involves the $|c : |Client$ role whose base type is the $Lifeline$ metaclass, and is satisfied by the $: VisitorDrawingProblem$ lifeline which is an instance of the $Lifeline$ metaclass, denoted as Y. Property $P2$ relates to the $|m : |CompositeElement^*$ and $|e : |LeafElement$ roles, both of which have the base type of the $Lifeline$ metaclass. The $|m : |CompositeElement^*$ role is played by the $: Picture^*$ and $: Line^*$ lifelines which are instances of the $Lifeline$ metaclass, denoted as Y. Property $P3$, which involves four exclusive-or cases, is satisfied by the third case where the client directly handles the $draw()$ calls on the drawing objects. The first case is not met, as there is no model element fulfilling the $|op()$ role on $|m$ called by $: |o$; the second case is not met because there is no operation enacting the $|op()$ role on $|e$ called by $: |o$; and the fourth case is not met as there is no operation fulfilling the $|op()$ role on $|e$ called by $|c$.



Property	SPS Roles	Role Type	Model Elements	Satified?
P1	|AbstractElement	Classifier Role	DrawObject, Shape	Y
P2 {XOR}	|ObjectStructure, |OC, |OA	Class Role, Association Role, Association Role	ObjectStructure, <ObjectStructure, VisitorDrawingProblem>, <ObjectStructure, DrawObject>	Y
	|CA	Association Role	None	N
P3	|Client	Class Role	VisitorDrawingProblem	Y
P4	|CompositeElement*, |LeafElement	Class Role, Class Role	Picture*, Rectangle*, Line*, Point	Y
P5	|op()	Operation Role	draw()	Y
P6 {XOR}	|CompReal*, |LeafReal	Realization Role, Realization Role,	None	N
	|CompGen*, |LeafGen	Generalization Role, Generalization Role,	<DrawObject,Picture>*, <Shape, Rectangle>*, <Shape, Line>*, <DrawObject, Point>	Y

(a) Visitor SPS Applicability



Property	IPS Roles	Role Type	Model Elements	Satified?
P1	|c:|Client	Lifeline Role	:VisitorDrawingProblem	Y
P2	|m:CompositeElement*, |e:|LeafElement	Lifeline Role, Lifeline Role	:Picture*, :Line*, p1:Point, p2:Point	Y
P3 {XOR}	|o:|ObjectStructure, |init(), |op() on |m by :|o, |op() on |e by |m	Lifeline Role, Message Role, Message Role, Message Role	:ObjectStructure, getObjectStructure, None, draw() on p1, draw() on p2	N
	|o:|ObjectStructure, |init(), |op() on |e by :|o	Lifeline Role, Message Role, Message Role	:ObjectStructure, getObjectStructure, None	N
	|op() on |m by |c, |op() on |e by |m	Message Role, Message Role	draw() on :picture, draw() on p1, draw() on p2	Y
	|op() on |e by |c	Message Role	None	N

(b) Visitor IPS Applicability

Figure 9: Visitor pattern applicability of drawing objects problem model

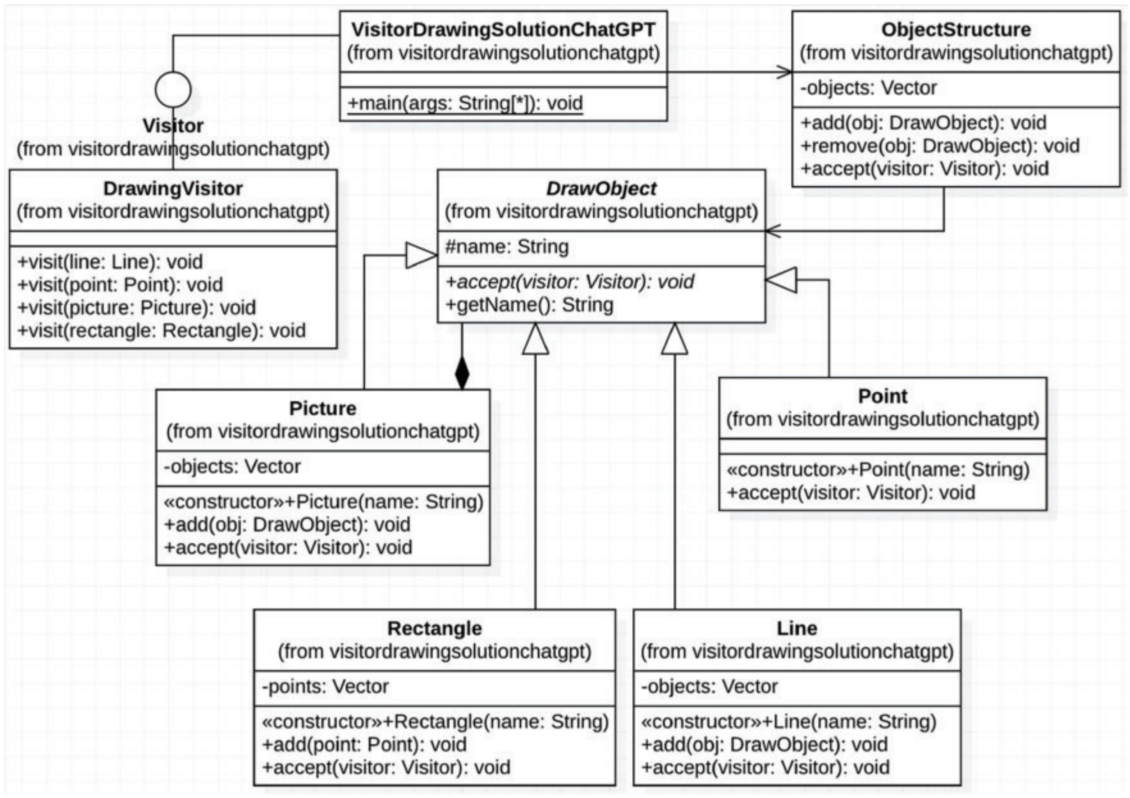
After ensuring pattern applicability, the problem model is input to the LLM for the application of the desired pattern. In this work, we chose ChatGPT 4 for the LLM, due to its increasing popularity in software engineering as discussed in Section 2. ChatGPT is instructed with the following prompt without any further instructions such as the information about the pattern or the context of the program.

```
Prompt: "Apply the Visitor design pattern to the given program."
```

ChatGPT produces a refactored program with the Visitor pattern applied. The resulting program is reverse-engineered to create the corresponding solution model as shown in Fig. 10. The solution class diagram retains most of the classes from the problem model with the exception of the $Shape$ class which was removed during the transformation. Upon further analysis of the deletion, ChatGPT indicates that the class was removed because it was not directly related to the Visitor pattern. Indeed, the removal did not affect the application's external behaviors, and can be seen as a design simplification. However, since the class played an integral part in the element hierarchy—fulfilling the $|CompositeElement$ role in the problem domain and the corresponding role in the solution domain—it should have been included in the transformation. This particular instance demonstrates a case where the LLM made an unexpected decision on its own, highlighting the need for human verification of an LLM's output. The pattern application has removed the $draw()$ operation from the drawing classes and introduced a new $accept()$ operation to them. The transformation has also introduced a visitor hierarchy that includes the $Visitor$ interface and the $DrawingVisitor$ class, encapsulating the semantics of the removed $draw()$ operations. The solution sequence diagram exhibits behaviors quite distinct from the problem sequence diagram with the new $visitor : DrawingVisitor$ lifeline and new message calls to $accept()$ and $visit()$ operations which together realize the double dispatch behavior, which is a key characteristic of the Visitor pattern.

The solution model is checked for its conformance to the solution specification of the Visitor pattern. Pattern conformance is evaluated based on how elements in the solution model correspond the pattern solution roles. Fig. 11 displays the correspondence of the elements in the solution class diagram to the solution SPS roles. In this alignment, the new $Visitor$ interface and the $DrawingVisitor$ class fulfill the $|AbstractVisitor$ and $|ConcreteVisitor$ roles, respectively. For other classes present in the problem class diagram, their mappings remain unchanged.

Fig. 12 presents the alignment of elements in the solution sequence diagram with their respective IPS roles. A notable aspect of this alignment is the correspondence of the $visitor : DrawingVisitor$ lifeline with the $|v : |ConcreteVisitor$ role, enabling the double dispatch behavior through $accept()$ and $visit()$ methods among composite elements, leaf elements, and the visitor.


VisitorDrawingSolutionChatGPT
(from visitordrawingsolutionchatgpt)
+main(args: String[*]): void
ObjectStructure
(from visitordrawingsolutionchatgpt)
-objects: Vector
+add(obj: DrawObject): void
+remove(obj: DrawObject): void
+accept(visitor: Visitor): void
Visitor
(from visitordrawingsolutionchatgpt)
DrawingVisitor
(from visitordrawingsolutionchatgpt)
+visit(line: Line): void
+visit(point: Point): void
+visit(picture: Picture): void
+visit(rectangle: Rectangle): void
DrawObject
(from visitordrawingsolutionchatgpt)
#name: String
+accept(visitor: Visitor): void
+getName(): String
Picture
(from visitordrawingsolutionchatgpt)
-objects: Vector
«constructor»+Picture(name: String)
+add(obj: DrawObject): void
+accept(visitor: Visitor): void
Point
(from visitordrawingsolutionchatgpt)
«constructor»+Point(name: String)
+accept(visitor: Visitor): void
Rectangle
(from visitordrawingsolutionchatgpt)
-points: Vector
«constructor»+Rectangle(name: String)
+add(point: Point): void
+accept(visitor: Visitor): void
Line
(from visitordrawingsolutionchatgpt)
-objects: Vector
«constructor»+Line(name: String)
+add(obj: DrawObject): void
+accept(visitor: Visitor): void

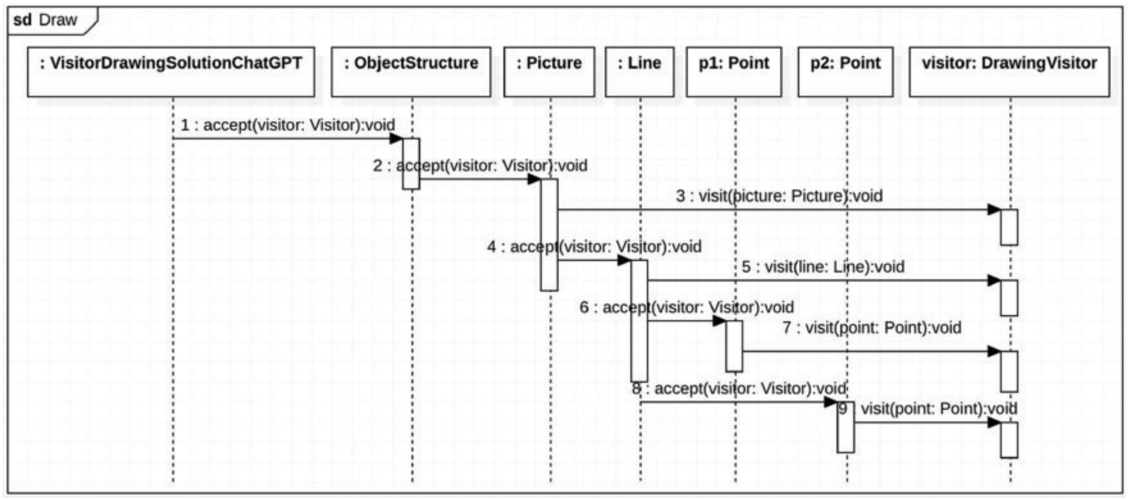

sd Draw
: VisitorDrawingSolutionChatGPT
: ObjectStructure
: Picture
: Line
p1: Point
p2: Point
visitor: DrawingVisitor
1 : accept(visitor: Visitor):void
2 : accept(visitor: Visitor):void
3 : visit(picture: Picture):void
4 : accept(visitor: Visitor):void
5 : visit(line: Line):void
6 : accept(visitor: Visitor):void
7 : visit(point: Point):void
8 : accept(visitor: Visitor):void
9 : visit(point: Point):void

Figure 10: Drawing objects solution model with visitor pattern applied by ChatGPT

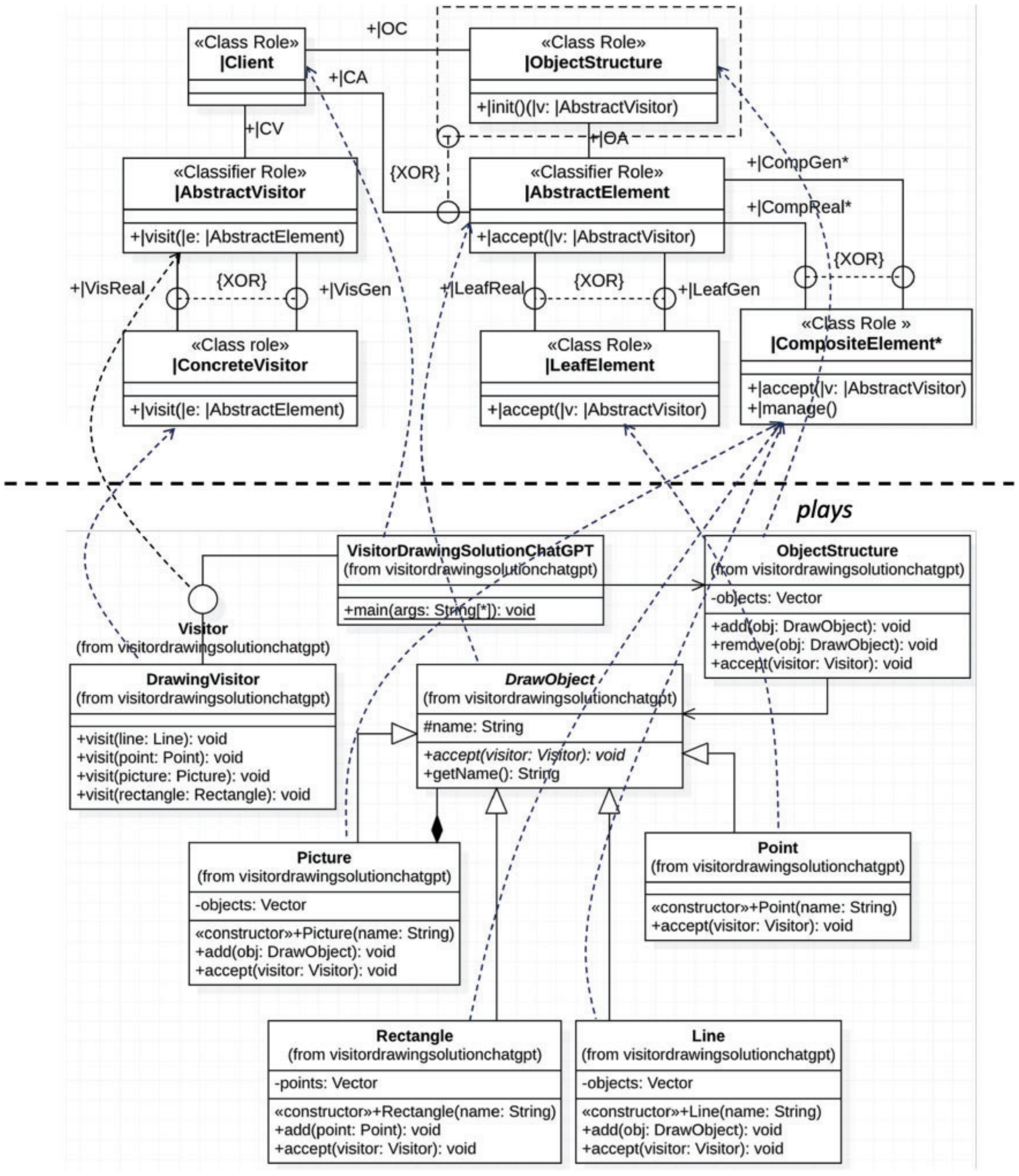

«Class Role»
|Client
+|OC
«Class Role»
|ObjectStructure
+|init()(|v: |AbstractVisitor)
+|CA
+|CV
+|OA
{XOR}
«Classifier Role»
|AbstractVisitor
+|visit(|e: |AbstractElement)
«Classifier Role»
|AbstractElement
+|accept(|v: |AbstractVisitor)
+|CompGen*
+|CompReal*
{XOR}
+|VisReal
{XOR}
+|VisGen
+|LeafReal
{XOR}
+|LeafGen
«Class role»
|ConcreteVisitor
+|visit(|e: |AbstractElement)
«Class Role»
|LeafElement
+|accept(|v: |AbstractVisitor)
«Class Role »
|CompositeElement*
+|accept(|v: |AbstractVisitor)
+|manage()
plays
VisitorDrawingSolutionChatGPT
(from visitordrawingsolutionchatgpt)
+main(args: String[*]): void
ObjectStructure
(from visitordrawingsolutionchatgpt)
-objects: Vector
+add(obj: DrawObject): void
+remove(obj: DrawObject): void
+accept(visitor: Visitor): void
Visitor
(from visitordrawingsolutionchatgpt)
DrawingVisitor
(from visitordrawingsolutionchatgpt)
+visit(line: Line): void
+visit(point: Point): void
+visit(picture: Picture): void
+visit(rectangle: Rectangle): void
DrawObject
(from visitordrawingsolutionchatgpt)
#name: String
+accept(visitor: Visitor): void
+getName(): String
Picture
(from visitordrawingsolutionchatgpt)
-objects: Vector
«constructor»+Picture(name: String)
+add(obj: DrawObject): void
+accept(visitor: Visitor): void
Point
(from visitordrawingsolutionchatgpt)
«constructor»+Point(name: String)
+accept(visitor: Visitor): void
Rectangle
(from visitordrawingsolutionchatgpt)
-points: Vector
«constructor»+Rectangle(name: String)
+add(point: Point): void
+accept(visitor: Visitor): void
Line
(from visitordrawingsolutionchatgpt)
-objects: Vector
«constructor»+Line(name: String)
+add(obj: DrawObject): void
+accept(visitor: Visitor): void

Figure 11: Visitor solution SPS conformance of drawing objects solution class diagram

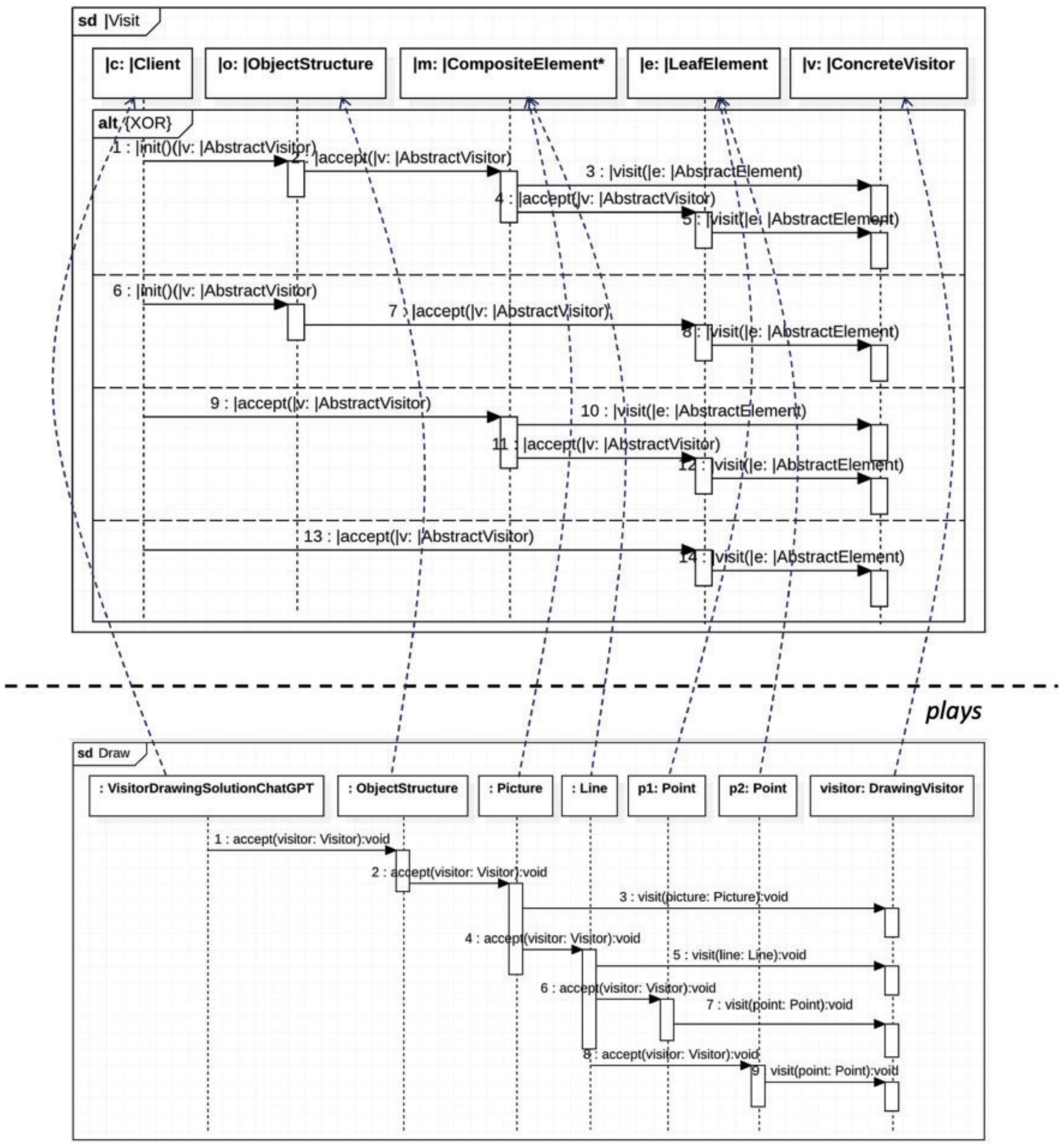
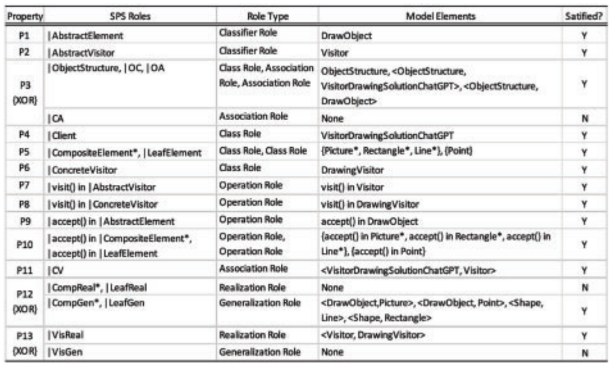

sd |Visit
|c: |Client
|o: |ObjectStructure
|m: |CompositeElement*
|e: |LeafElement
|v: |ConcreteVisitor
alt {XOR}
1 : |init()(|v: |AbstractVisitor)
2 : |accept(|v: |AbstractVisitor)
3 : |visit(|e: |AbstractElement)
4 : |accept(|v: |AbstractVisitor)
5 : |visit(|e: |AbstractElement)
6 : |init()(|v: |AbstractVisitor)
7 : |accept(|v: |AbstractVisitor)
8 : |visit(|e: |AbstractElement)
9 : |accept(|v: |AbstractVisitor)
10 : |visit(|e: |AbstractElement)
11 : |accept(|v: |AbstractVisitor)
12 : |visit(|e: |AbstractElement)
13 : |accept(|v: |AbstractVisitor)
14 : |visit(|e: |AbstractElement)
plays
sd Draw
: VisitorDrawingSolutionChatGPT
: ObjectStructure
: Picture
: Line
p1: Point
p2: Point
visitor: DrawingVisitor
1 : accept(visitor: Visitor):void
2 : accept(visitor: Visitor):void
3 : visit(picture: Picture):void
4 : accept(visitor: Visitor):void
5 : visit(line: Line):void
6 : accept(visitor: Visitor):void
7 : visit(point: Point):void
8 : accept(visitor: Visitor):void
9 : visit(point: Point):void

Figure 12: Visitor solution IPS conformance of *Draw* solution sequence diagram

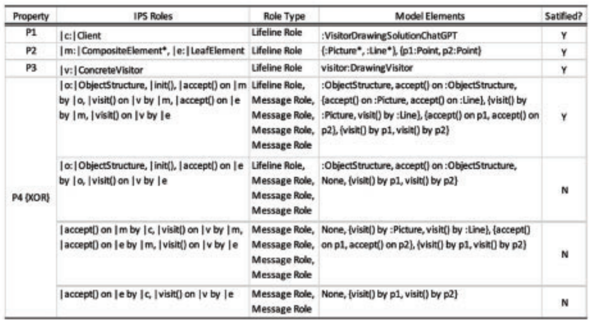
Based on the mapping in Figs. 11 and 12, the conformance of the solution model to the solution specification of the Visitor pattern is evaluated as depicted in Fig. 13. Fig. 13a illustrates the SPS conformance where all thirteen SPS properties are satisfied. P1 is satisfied by the $DrawObject$ class fulfilling the $|AbstractElement$ role; P2 is satisfied by the $Visitor$ class enacting the $|AbstractVisitor$ role; P3 is satisfied by the first case where the client accesses drawing objects via the object structure, with the second case failing due to the absence of model elements that can fulfill the $|CA$ role; P4 is satisfied by the $VisitorDrawingSolutionChatGPT$ class playing the $|Client$ role; P5 is satisfied by the $Picture^*$, $Rectangle^*$, and $Line^*$ classes fulfilling the $|CompositeElement^*$ role, and the $Point$ class enacting the $|LeafElement$ role; P6 is satisfied by the $DrawingVisitor$ class fulfilling the $|ConcreteVisitor$ role; P7 is satisfied by the $visit()$ operation in the $Visitor$ class, which plays the $|visit()$ role within the

$|AbstractVisitor$ role; P8 is satisfied by the $visit()$ operation in the $DrawingVisitor$ class, which plays the $|visit()$ role in the $|ConcreteVisitor$ role; P9 is satisfied by the $accept()$ operation in the $DrawObject$ class, which enacts the $|accept()$ role in the $|AbstractElement$ role; P10 is satisfied by the $accept()$ operation in the $Picture^*$, $Rectangle^*$, and $Line^*$ classes, which play the $|accept()$ role in the $|CompositeElement^*$ role, and the $accept()$ operation in the $Point$ class, which fulfills the $|accept()$ role in the $|LeafElement$ role; P11 is satisfied by the $< VisitorDrawingSolutionChatGPT, Visitor >$ relationship, which enacts the $|CV$ role; P12 is satisfied by the second case, which captures the generalization relationships among the drawing object hierarchy, with the first case failing due to the absence of model elements that can fulfill the $|CompReal^*$ and $|LeafReal$ roles; P13 is satisfied by the first case capturing the realization relationship in the visitor hierarchy, with the second case failing due to the absence of model elements that can play the $|VisGen$ role. This results in full SPS conformance. Fig. 13b confirms the IPS conformance where all four IPS properties are satisfied. P1 is satisfied by the $: VisitorDrawingSolutionChatGPT$ lifeline, which plays the $|c : |Client$ role; P2 is satisfied by the $: Picture^*$ and $: Line^*$ lifelines enacting the $|m : |CompositeElement^*$ role and the $p1 : Point$ and $p2 : Point$ lifelines fulfilling the $|e : |LeafElement$ role; P3 is satisfied by the $visitor : DrawingVisitor$ lifeline playing the $|v : |ConcreteVisitor$ role; P4, which involves four exclusive-or cases, is satisfied by the first case where the object structure facilitates the recursive call to the $accept()$ operation on composing objects. The second case fails due to the absence of model elements fulfilling the $|init()$ role, and the third case fails due to the absence of model elements playing the $|accept()$ role on $|m$ called by $|c$. This leads to full IPS conformance, resulting in 100% conformance ($\frac{13+4}{13+4} \times 100$) to the pattern's solution specification.



Property	SPS Roles	Role Type	Model Elements	Satified?
P1	|AbstractElement	Classifier Role	DrawObject	Y
P2	|AbstractVisitor	Classifier Role	Visitor	Y
P3 (XOR)	|ObjectStructure, |OC, |OA	Class Role, Association Role, Association Role	ObjectStructure, <ObjectStructure, VisitorDrawingSolutionChatGPT>, <ObjectStructure, DrawObject>	Y
	|CA	Association Role	None	N
P4	|Client	Class Role	VisitorDrawingSolutionChatGPT	Y
P5	|CompositeElement*, |LeafElement	Class Role, Class Role	{Picture*, Rectangle*, Line*}, {Point}	Y
P6	|ConcreteVisitor	Class Role	DrawingVisitor	Y
P7	|visit() in |AbstractVisitor	Operation Role	visit() in Visitor	Y
P8	|visit() in |ConcreteVisitor	Operation Role	visit() in DrawingVisitor	Y
P9	|accept() in |AbstractElement	Operation Role	accept() in DrawObject	Y
P10	|accept() in |CompositeElement*, |accept() in |LeafElement	Operation Role, Operation Role	{accept() in Picture*, accept() in Rectangle*, accept() in Line*}, {accept() in Point}	Y
P11	|CV	Association Role	<VisitorDrawingSolutionChatGPT, Visitor>	Y
P12 (XOR)	|CompReal*, |LeafReal	Realization Role	None	N
	|CompGen*, |LeafGen	Generalization Role	<DrawObject,Picture>, <DrawObject, Point>, <Shape, Line>, <Shape, Rectangle>	Y
P13 (XOR)	|VisReal	Realization Role	<Visitor, DrawingVisitor>	Y
	|VisGen	Generalization Role	None	N

(a) Solution SPS Conformance

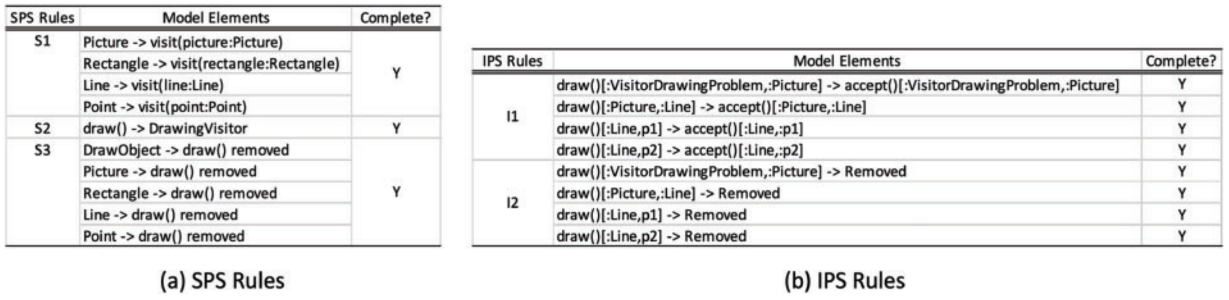


Property	IPS Roles	Role Type	Model Elements	Satified?
P1	|c:|Client	Lifeline Role	:VisitorDrawingSolutionChatGPT	Y
P2	|m:|CompositeElement*, |e:|LeafElement	Lifeline Role	{:Picture*, :Line*}, {p1:Point, p2:Point}	Y
P3	|v:|ConcreteVisitor	Lifeline Role	visitor:DrawingVisitor	Y
P4 (XOR)	|o:|ObjectStructure, |init(), |accept() on |m by |o, |visit() on |v by |m, |accept() on |e by |m, |visit() on |v by |e	Lifeline Role, Message Role, Message Role, Message Role, Message Role	:ObjectStructure, accept() on :ObjectStructure, {accept() on :Picture, accept() on :Line}, {visit() by :Picture, visit() by :Line}, {accept() on p1, accept() on p2}, {visit() by p1, visit() by p2}	Y
	|o:|ObjectStructure, |init(), |accept() on |e by |o, |visit() on |v by |e	Lifeline Role, Message Role, Message Role, Message Role	:ObjectStructure, accept() on :ObjectStructure, None, {visit() by p1, visit() by p2}	N
	|accept() on |m by |c, |visit() on |v by |m, |accept() on |e by |m, |visit() on |v by |e	Message Role, Message Role, Message Role, Message Role	None, {visit() by :Picture, visit() by :Line}, {accept() on p1, accept() on p2}, {visit() by p1, visit() by p2}	N
	|accept() on |e by |c, |visit() on |v by |e	Message Role, Message Role	None, {visit() by p1, visit() by p2}	N

(b) Solution IPS Conformance

Figure 13: Visitor pattern conformance of drawing objects solution model

Pattern conformance ensures that the model possesses the pattern properties. However, it does not necessarily indicate that the model has a complete realization of the pattern. For instance, if the elements in the object structure involve more than one common operation across the element hierarchy, there should be a separate concrete visitor class and visit operation for each operation. If not all common operations are covered, the pattern is not fully realized. The completeness of the pattern realization can be checked using the pattern's transformation rules, as presented in Section 3. Fig. 14 demonstrates the completeness of the pattern realization, checking the enforcement of transformation rules. Fig. 14a indicates that all three SPS rules are fully enforced over all relevant elements, leading to a complete realization of the SPS. S1 requires the creation of a visit operation for every drawing element, and the table confirms that a corresponding visit operation exists for all four elements. If any element had been omitted, the enforcement of the rule would have been incomplete. Fig. 14b shows that both of the two IPS rules are enforced over all relevant lifelines, resulting in a complete realization of the IPS. Overall, all 5 transformation rules ($\frac{3+2}{3+2} \times 100$) are fully enforced, leading to 100% completeness in the pattern realization.



SPS Rules	Model Elements	Complete?
S1	Picture -> visit(picture:Picture)	Y
	Rectangle -> visit(rectangle:Rectangle)	
	Line -> visit(line:Line)	
	Point -> visit(point:Point)	
S2	draw() -> DrawingVisitor	Y
S3	DrawObject -> draw() removed	Y
	Picture -> draw() removed	
	Rectangle -> draw() removed	
	Line -> draw() removed	
	Point -> draw() removed	

(a) SPS Rules

IPS Rules	Model Elements	Complete?
I1	draw()[:VisitorDrawingProblem,:Picture] -> accept()[:VisitorDrawingProblem,:Picture]	Y
	draw()[:Picture,:Line] -> accept()[:Picture,:Line]	Y
	draw()[:Line,p1] -> accept()[:Line,:p1]	Y
	draw()[:Line,p2] -> accept()[:Line,:p2]	Y
I2	draw()[:VisitorDrawingProblem,:Picture] -> Removed	Y
	draw()[:Picture,:Line] -> Removed	Y
	draw()[:Line,p1] -> Removed	Y
	draw()[:Line,p2] -> Removed	Y

(b) IPS Rules

Figure 14: Visitor pattern completeness of drawing objects solution model

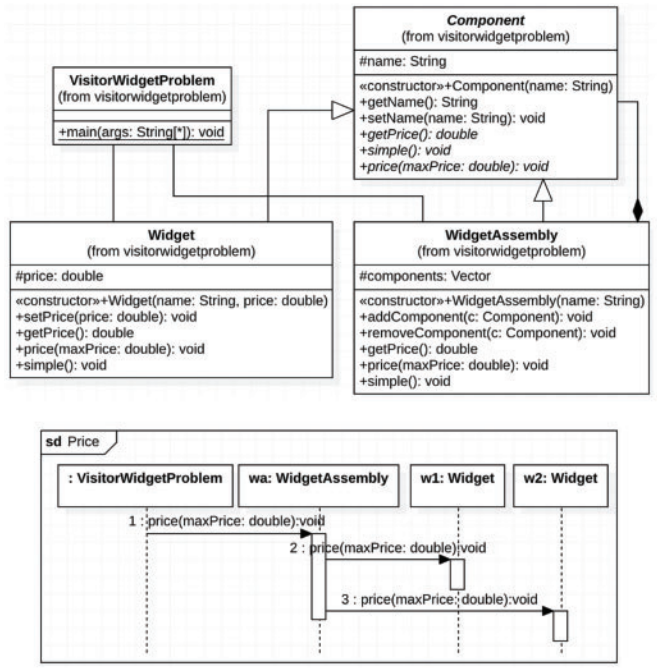
5 Case Studies

In this section, we present two additional case studies to evaluate the approach, applying the Visitor pattern to a widget application and a node application using ChatGPT. These applications are small in size, written in Java. We deliberately chose small applications for two main reasons—i) to adapt to LLMs' limitations on processing large inputs, and ii) to examine how the LLM applies the pattern comprehensively across the application. The source code for these applications, both before and after pattern application, is available on GitHub [41].

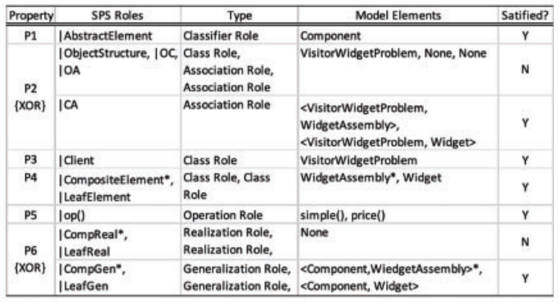
5.1 Widget Application

The widget application is concerned with building a widget assembly consisting of a set of widgets. A widget assembly may contain other widget assemblies as well. After a widget assembly is built, the application processes each comprising widget and widget assembly to display their names and determine the price of individual widgets, which is used for computing the total price of the widget equipment. Fig. 15 shows the widget problem model reserve-engineered from the application and its applicability to the Visitor pattern's problem specification. In the model, the $simple()$ operation is used to display the name of composing components and the $price()$ operation is used to access the price of components. The sequence diagram captures the $price()$ behavior, and a similar diagram can be specified for the $simple()$ behavior. Fig. 15a,b shows that the model exhibits applicability to the Visitor pattern in both SPS and IPS. In the SPS applicability in Fig. 15b, P1 is satisfied by the $Component$ class fulfilling the $|AbstractElement$ role; P2 is satisfied by the second case where no separate object structure exists, while the first case fails due to the absence of model elements that can play the $|OC$ and $|OA$ roles; P3 is satisfied by the $VisitorWidgetProblem$ class enacting the $|Client$ role; P4 is satisfied by the $WidgetAssembly^*$ class fulfilling the $|CompositeElement^*$ role and the $Widget$ class playing the $|LeafElement$ role; P5 is satisfied by the $simple()$ and $price()$ operations, both of which enact the $|op()$ role; P6 is satisfied by the second case, which captures a generalization hierarchy among component elements. The first case fails due to the absence of model elements that can fulfill the $|CompReal^*$ and $|LeafReal$ roles.

With respect to IPS applicability in Fig. 15c, P1 is satisfied by the $: VisitorDrawingProblem$ lifeline playing the $|c : |Client$ role; P2 is satisfied by the $wa : WidgetAssembly^*$ lifeline enacting the $|m : |CompositeElement^*$ role and the $w1 : Widget$ and $w2 : Widget$ lifelines fulfilling the $|e : |LeafElement$ role; P3 is fulfilled by the third case, which involves the client delegating the $price()$ message through composite components (widget assemblies), bypassing the object structure. The first case fails due to the absence of model elements that can play the $|o : |ObjectStructure$ and $|init()$ roles. Similarly, the second case fails due to the absence of model elements that can enact the $|o : |ObjectStructure$ and $|init()$ roles. The fourth case fails due to the absence of model elements that can fulfill the $|op()$ role on $|e$ called by $|c$.

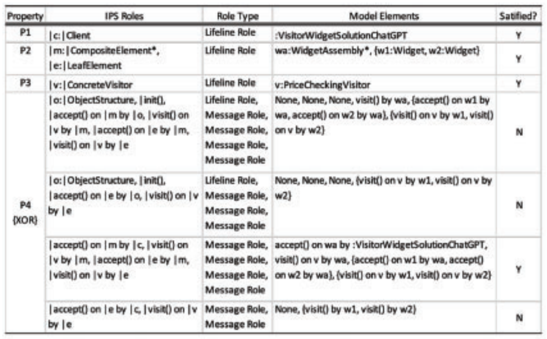

Component
(from visitorwidgetproblem)
#name: String
«constructor»+Component(name: String)
+getName(): String
+setName(name: String): void
+getPrice(): double
+simple(): void
+price(maxPrice: double): void
VisitorWidgetProblem
(from visitorwidgetproblem)
+main(args: String[*]): void
Widget
(from visitorwidgetproblem)
#price: double
«constructor»+Widget(name: String, price: double)
+setPrice(price: double): void
+getPrice(): double
+price(maxPrice: double): void
+simple(): void
WidgetAssembly
(from visitorwidgetproblem)
#components: Vector
«constructor»+WidgetAssembly(name: String)
+addComponent(c: Component): void
+removeComponent(c: Component): void
+getPrice(): double
+price(maxPrice: double): void
+simple(): void
sd Price
: VisitorWidgetProblem
wa: WidgetAssembly
w1: Widget
w2: Widget
1 : price(maxPrice: double):void
2 : price(maxPrice: double):void
3 : price(maxPrice: double):void

(a) Widget Problem Model



Property	SPS Roles	Type	Model Elements	Satified?
P1	|AbstractElement	Classifier Role	Component	Y
P2 {XOR}	|ObjectStructure, |OC, |OA	Class Role, Association Role, Association Role	VisitorWidgetProblem, None, None	N
	|CA	Association Role	<VisitorWidgetProblem, WidgetAssembly>, <VisitorWidgetProblem, Widget>	Y
P3	|Client	Class Role	VisitorWidgetProblem	Y
P4	|CompositeElement*, |LeafElement	Class Role, Class Role	WidgetAssembly*, Widget	Y
P5	|op()	Operation Role	simple(), price()	Y
P6 {XOR}	|CompReal*, |LeafReal	Realization Role, Realization Role,	None	N
	|CompGen*, |LeafGen	Generalization Role, Generalization Role,	<Component,WiedgetAssembly>*, <Component, Widget>	Y

(b) Visitor Problem SPS Applicability

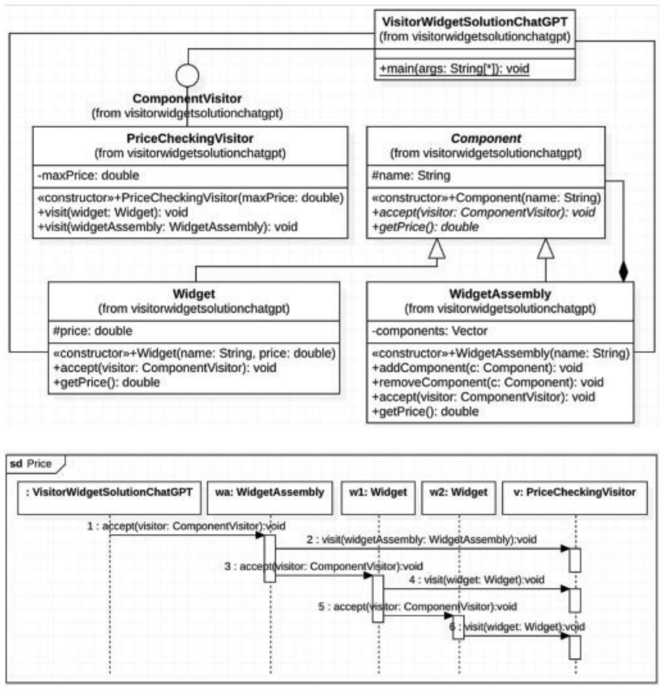


Property	IPS Roles	Role Type	Model Elements	Satified?
P1	|c:|Client	Lifeline Role	:VisitorWidgetSolutionChatGPT	Y
P2	|m:|CompositeElement*, |e:|LeafElement	Lifeline Role	wa:WidgetAssembly*, {w1:Widget, w2:Widget}	Y
P3	|v:|ConcreteVisitor	Lifeline Role	v:PriceCheckingVisitor	Y
P4 (XOR)	|o:|ObjectStructure, |init(), |accept() on |m by |o, |visit() on |v by |m, |accept() on |e by |m, |visit() on |v by |e	Lifeline Role, Message Role, Message Role, Message Role, Message Role	None, None, None, visit() by wa, {accept() on w1 by wa, accept() on w2 by wa}, {visit() on v by w1, visit() on v by w2}	N
	|o:|ObjectStructure, |init(), |accept() on |e by |o, |visit() on |v by |e	Lifeline Role, Message Role, Message Role, Message Role	None, None, None, {visit() on v by w1, visit() on v by w2}	N
	|accept() on |m by |c, |visit() on |v by |m, |accept() on |e by |m, |visit() on |v by |e	Message Role, Message Role, Message Role, Message Role	accept() on wa by :VisitorWidgetSolutionChatGPT, visit() on v by wa, {accept() on w1 by wa, accept() on w2 by wa}, {visit() on v by w1, visit() on v by w2}	Y
	|accept() on |e by |c, |visit() on |v by |e	Message Role, Message Role	None, {visit() by w1, visit() by w2}	N

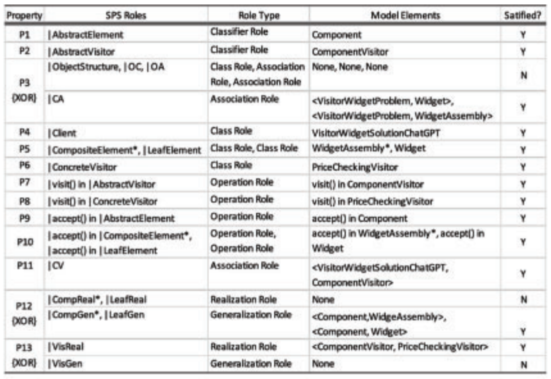
(c) Visitor Problem IPS Applicability

Figure 15: Visitor pattern applicability of widget problem model

After ensuring pattern applicability, the problem model is input to ChatGPT to apply the Visitor pattern. Fig. 16 displays the model refactored by the Visitor pattern application and its conformance to the pattern's solution specification. The model possesses the visitor hierarchy, consisting of the $ComponentVisitor$ interface and the $PriceCheckingVisitor$ class corresponding to the $price()$ operation in the problem model. However, the hierarchy does not include a visitor class for the $simple()$ operation in the problem model because ChatGPT thinks it is a trivial and self-contained behavior that simply prints a message, making the use of the Visitor pattern unnecessary and overly complex. Additionally, the $simple()$ operation naturally aligns with the responsibilities of each class (e.g., $Widget$ or $WidgetAssembly$) and does not require external context or extensibility. ChatGPT believes that keeping the logic within the respective classes adheres to the Single Responsibility Principle, ensuring clarity and avoiding unnecessary abstraction. This reasoning makes sense, as introducing a visitor for such a simple operation would add needless complexity without providing significant benefits. The sequence diagram shows the behavior of computing the price of components using the price visitor. Fig. 16b,c shows that all thirteen SPS properties and all four IPS properties are satisfied. For the SPS conformance in Fig. 16b, P1 is satisfied by the $Component$ class enacting the $|AbstractElement$ role; P2 is satisfied by the $ComponentVisitor$ class playing the $|AbstractVisitor$ role; P3 is satisfied by the second case, in which the client directly accesses widgets without involving the object structure. The first case fails due to the absence of model elements capable of fulfilling the roles $|ObjectStructure$, $|OC$, and $|OA$; P4 is satisfied by the $VisitorWidgetSolutionChatGPT$ class fulfilling the $|Client$ role; P5 is satisfied by the $WidgetAssembly^*$ class playing the $|CompositeElement^*$ role and the $Widget$ class playing the $|LeafElement$ role; P6 is satisfied by the $PriceCheckingVisitor$ class playing the $|ConcreteVisitor$ role; P7 is satisfied by the $visit()$ operation in $CompositeVisitor$ which fulfills the $|visit()$ role in $|AbstractVisitor$; P8 is satisfied by the $visit()$ operation in $PriceCheckingVisitor$ which enacts the $|visit()$ role in $|ConcreteVisitor$; P9 is satisfied by the $accept()$ operation in $Component$ which plays the $|accept()$ operation in $|AbstractElement$; P10 is satisfied by the $accept()$ operation in $WidgetAssembly^*$ playing the $|accept()$ role in $|CompositeElement^*$ and the $accept()$ operation in $Widget$ fulfilling the $|accept()$ role in $|LeafElement$; P11 is satisfied by the $< VisitorWidgetSolutionChatGPT, ComponentVisitor >$ relationship playing the $|CV$ role; P12 is satisfied by the second case where elements are in a generalization hierarchy, with the $< Component, WidgetAssembly >$ relationship playing the $|CompGen^*$ role and the $< Component, Widget >$ relationship fulfilling the $|LeafGen$ role. The first case fails due to the absence of model elements enacting the $|CompReal^*$ and $|LeafReal$ roles; P13 is satisfied by the first case where visitors are in a realization hierarchy, with the $< ComponentVisitor, PriceCheckingVisitor >$ relationship playing the $|VisReal$ role. The second case fails due to the absence of generalization relationships that can play the $|VisGen$ role.

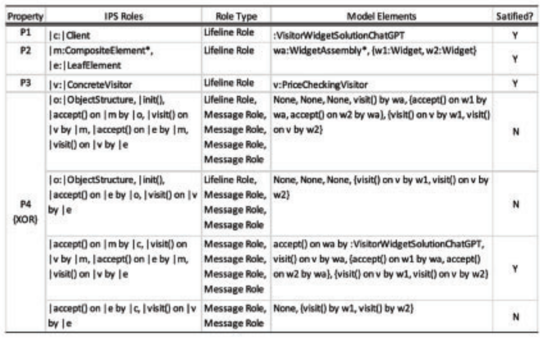

VisitorWidgetSolutionChatGPT
(from visitorwidgetsolutionchatgpt)
+main(args: String[*]): void
ComponentVisitor
(from visitorwidgetsolutionchatgpt)
PriceCheckingVisitor
(from visitorwidgetsolutionchatgpt)
-maxPrice: double
«constructor»+PriceCheckingVisitor(maxPrice: double)
+visit(widget: Widget): void
+visit(widgetAssembly: WidgetAssembly): void
Component
(from visitorwidgetsolutionchatgpt)
#name: String
«constructor»+Component(name: String)
+accept(visitor: ComponentVisitor): void
+getPrice(): double
Widget
(from visitorwidgetsolutionchatgpt)
#price: double
«constructor»+Widget(name: String, price: double)
+accept(visitor: ComponentVisitor): void
+getPrice(): double
WidgetAssembly
(from visitorwidgetsolutionchatgpt)
-components: Vector
«constructor»+WidgetAssembly(name: String)
+addComponent(c: Component): void
+removeComponent(c: Component): void
+accept(visitor: ComponentVisitor): void
+getPrice(): double
sd Price
: VisitorWidgetSolutionChatGPT
wa: WidgetAssembly
w1: Widget
w2: Widget
v: PriceCheckingVisitor
1 : accept(visitor: ComponentVisitor):void
2 : visit(widgetAssembly: WidgetAssembly):void
3 : accept(visitor: ComponentVisitor):void
4 : visit(widget: Widget):void
5 : accept(visitor: ComponentVisitor):void
6 : visit(widget: Widget):void

(a) Widget Solution Model



Property	SPS Roles	Role Type	Model Elements	Satified?
P1	|AbstractElement	Classifier Role	Component	Y
P2	|AbstractVisitor	Classifier Role	ComponentVisitor	Y
P3 (XOR)	|ObjectStructure, |OC, |OA	Class Role, Association Role, Association Role	None, None, None	N
	|CA	Association Role	<VisitorWidgetProblem, Widget>, <VisitorWidgetProblem, WidgetAssembly>	Y
P4	|Client	Class Role	VisitorWidgetSolutionChatGPT	Y
P5	|CompositeElement*, |LeafElement	Class Role, Class Role	WidgetAssembly*, Widget	Y
P6	|ConcreteVisitor	Class Role	PriceCheckingVisitor	Y
P7	|visit() in |AbstractVisitor	Operation Role	visit() in ComponentVisitor	Y
P8	|visit() in |ConcreteVisitor	Operation Role	visit() in PriceCheckingVisitor	Y
P9	|accept() in |AbstractElement	Operation Role	accept() in Component	Y
P10	|accept() in |CompositeElement*, |accept() in |LeafElement	Operation Role, Operation Role	accept() in WidgetAssembly*, accept() in Widget	Y
P11	|CV	Association Role	<VisitorWidgetSolutionChatGPT, ComponentVisitor>	Y
P12 (XOR)	|CompReal*, |LeafReal	Realization Role	None	N
	|CompGen*, |LeafGen	Generalization Role	<Component,WidgeAssembly>, <Component, Widget>	Y
P13 (XOR)	|VisReal	Realization Role	<ComponentVisitor, PriceCheckingVisitor>	Y
	|VisGen	Generalization Role	None	N

(b) Visitor Solution SPS Conformance



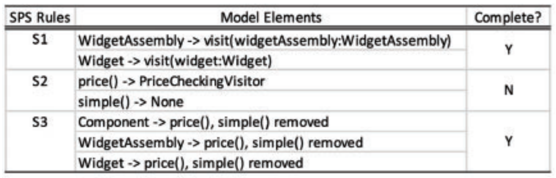
Property	IPS Roles	Role Type	Model Elements	Satified?
P1	|c:|Client	Lifeline Role	:VisitorWidgetSolutionChatGPT	Y
P2	|m:CompositeElement*, |e:|LeafElement	Lifeline Role	wa:WidgetAssembly*, {w1:Widget, w2:Widget}	Y
P3	|v:|ConcreteVisitor	Lifeline Role	v:PriceCheckingVisitor	Y
P4 (XOR)	|o:|ObjectStructure, |init(), |accept() on |m by |o, |visit() on |v by |m, |accept() on |e by |m, |visit() on |v by |e	Lifeline Role, Message Role, Message Role, Message Role, Message Role	None, None, None, visit() by wa, {accept() on w1 by wa, accept() on w2 by wa}, {visit() on v by w1, visit() on v by w2}	N
	|o:|ObjectStructure, |init(), |accept() on |e by |o, |visit() on |v by |e	Lifeline Role, Message Role, Message Role, Message Role	None, None, None, {visit() on v by w1, visit() on v by w2}	N
	|accept() on |m by |c, |visit() on |v by |m, |accept() on |e by |m, |visit() on |v by |e	Message Role, Message Role, Message Role, Message Role	accept() on wa by :VisitorWidgetSolutionChatGPT, visit() on v by wa, {accept() on w1 by wa, accept() on w2 by wa}, {visit() on v by w1, visit() on v by w2)	Y
	|accept() on |e by |c, |visit() on |v by |e	Message Role, Message Role	None, {visit() by w1, visit() by w2}	N

(c) Visitor Solution IPS Conformance

Figure 16: Visitor pattern conformance of widget solution model

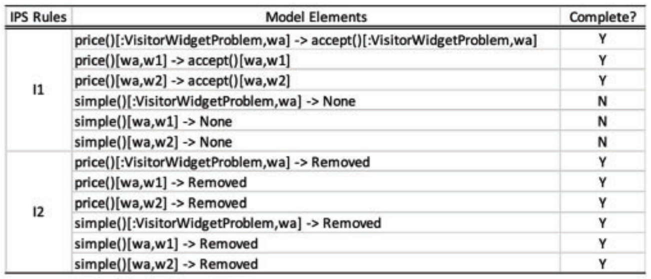
With respect to IPS conformance in Fig. 16c, P1 is satisfied by the $: VisitorWidgetSolutionChatGPT$ lifeline playing the $|c : |Client$ role; P2 is satisfied by the $wa : WidgetAssembly^*$ lifeline fulfilling the $|m : |CompositeElement^*$ role and the $w1 : Widget$ and $w2 : Widget$ lifelines enacting the $|e : |LeafElement$ role; P3 is satisfied by the $v : PriceCheckingVisitor$ lifeline playing the $|v : |ConcreteVisitor$ role; P4 is satisfied by the third case where the double-dispatch mechanism is performed via the client and composite elements. The $accept()$ operation on wa is initiated by $: VisitorWidgetSolutionChatGPT$, fulfilling the $|accept()$ role on $|m$ called by $|c$. The $visit()$ operation on v by wa plays the $|visit()$ role on $|v$ called by $|m$. Similarly, the $accept()$ operations on $w1$ and $w2$, called by wa, enact the $|accept()$ role on $|e$ called by $|m$. Lastly, the $visit()$ operations on v called by $w1$ and $w2$ fulfill the $|visit()$ role on $|v$ called by $|e$. The first case fails due to the absence of model elements that can fulfill the $|o : |ObjectStructure$ role, the $|init()$ role, and the $|accept()$ role on $|m$ called by $|o$. The second case fails due to the absence of model elements that can enact the $|o : |ObjectStructure$ role, the $|init()$ role, and the $|accept()$ role on $|e$ called by $|o$. The fourth case fails due to the absence of model elements that can fulfill the $|accept()$ role on $|e$ called by $|c$. The SPS and IPS conformance leads to 100% ($\frac{13+4}{13+4} \times 100$) conformance to the pattern's solution specification.

Fig. 17 illustrates the evaluation of pattern completeness for the widget application. Fig. 17a indicates that only two out of the three SPS rules are fully enforced, with S2 not being fully enforced for the $simple()$ operation. This results in the absence of the corresponding visitor in the visitor hierarchy. Fig. 17b shows that only one of the two IPS rules is fully enforced, with I1 not fully enforced due to the missing visitor for the $simple()$ operation in S2. This results in 60% ($\frac{2+1}{3+2} \times 100$) completeness of the pattern realization.



SPS Rules	Model Elements	Complete?
S1	WidgetAssembly -> visit(widgetAssembly:WidgetAssembly)	Y
	Widget -> visit(widget:Widget)	
S2	price() -> PriceCheckingVisitor	N
	simple() -> None	
S3	Component -> price(), simple() removed	Y
	WidgetAssembly -> price(), simple() removed	
	Widget -> price(), simple() removed	

(a) SPS Rules



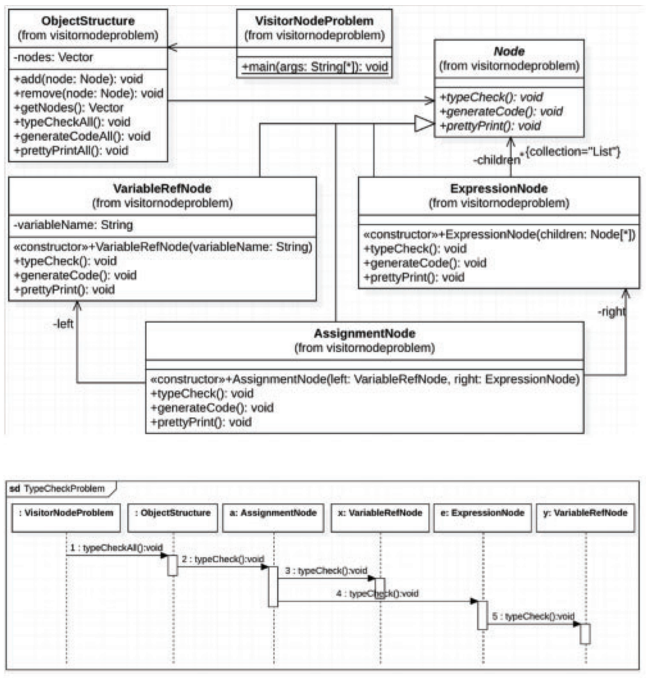
IPS Rules	Model Elements	Complete?
I1	price()[:VisitorWidgetProblem,wa] -> accept()[:VisitorWidgetProblem,wa]	Y
	price()[wa,w1] -> accept()[wa,w1]	Y
	price()[wa,w2] -> accept()[wa,w2]	Y
	simple()[:VisitorWidgetProblem,wa] -> None	N
	simple()[wa,w1] -> None	N
	simple()[wa,w2] -> None	N
I2	price()[:VisitorWidgetProblem,wa] -> Removed	Y
	price()[wa,w1] -> Removed	Y
	price()[wa,w2] -> Removed	Y
	simple()[:VisitorWidgetProblem,wa] -> Removed	Y
	simple()[wa,w1] -> Removed	Y
	simple()[wa,w2] -> Removed	Y

(b) IPS Rules

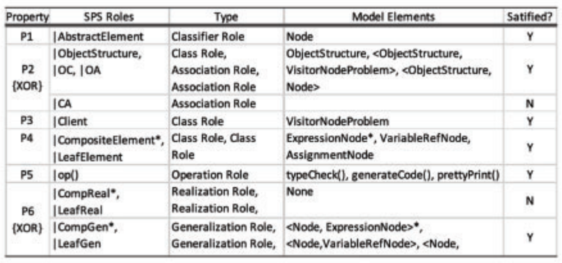
Figure 17: Visitor pattern completeness of widget solution model

5.2 Node Application

The node application is concerned with constructing and manipulating a hierarchical structure of nodes in the syntax of a programming language. Fig. 18 shows the node problem model and its applicability to the Visitor pattern. The class diagram in (a) defines specific node classes such as $AssignmentNode$, $ExpressionNode$, and $VariableRefNode$ which inherit from the base $Node$ class, suggesting a design that accommodates a tree-like structure typical in abstract syntax trees used in compilers or interpreters. $AssignmentNode$ represents assignment statements, linking a reference variable captured by $VariableRefNode$ with an expression denoted by $ExpressionNode$. Node classes include methods for type checking, code generation, and pretty printing which are used in language processing. The sequence diagram specifies the type checking behavior which verifies the type of involved variables and their compatibility. Similar sequence diagrams can be defined for code generation and pretty printing. Fig. 18b shows that all six SPS properties are satisfied, leading to a successful establishment of SPS applicability. P1 is satisfied by the $Node$ class fulfilling the $|AbstractElement$ role; P2 is satisfied by the first case where an object structure exists. However, the second case fails due to the absence of model elements that can play the $|OA$ role; P3 is satisfied by the $VisitorNodeProblem$ class enacting the $|Client$ role; P4 is satisfied by the $ExpressionNode^*$ class fulfilling the $|CompositeElement^*$ role, with the $VariableRefNode$ and $AssignmentNode$ classes playing the $|LeafElement$ role; P5 is satisfied by the $typeCheck()$, $generateCode()$, and $prettyPrint()$ operations, each of which enacts the $|op()$ role; P6 is satisfied by the second case, capturing a generalization hierarchy among node elements, while the first case fails due to the absence of model elements that can fulfill the $|CompReal^*$ and $|LeafReal$ roles.

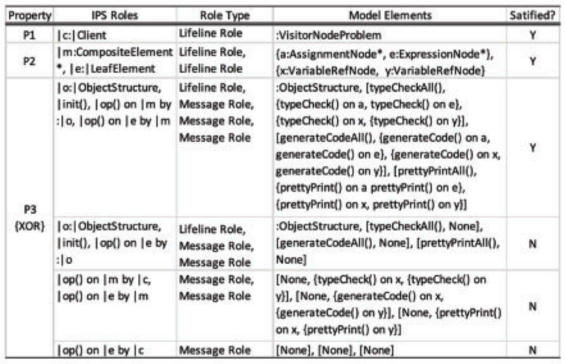


(a) Node Problem Model



Property	SPS Roles	Type	Model Elements	Satified?
P1	|AbstractElement	Classifier Role	Node	Y
P2 {XOR}	|ObjectStructure, |OC, |OA	Class Role, Association Role, Association Role	ObjectStructure, <ObjectStructure, VisitorNodeProblem>, <ObjectStructure, Node>	Y
	|CA	Association Role		N
P3	|Client	Class Role	VisitorNodeProblem	Y
P4	|CompositeElement*, |LeafElement	Class Role, Class Role	ExpressionNode*, VariableRefNode, AssignmentNode	Y
P5	|op()	Operation Role	typeCheck(), generateCode(), prettyPrint()	Y
P6 {XOR}	|CompReal*, |LeafReal	Realization Role, Realization Role,	None	N
	|CompGen*, |LeafGen	Generalization Role, Generalization Role,	<Node, ExpressionNode>*, <Node,VariableRefNode>, <Node,	Y

(b) Visitor Problem SPS Applicability



Property	IPS Roles	Role Type	Model Elements	Satified?
P1	|c:|Client	Lifeline Role	:VisitorNodeProblem	Y
P2	|m:CompositeElement*, |e:|LeafElement	Lifeline Role, Lifeline Role	{a:AssignmentNode*, e:ExpressionNode*}, {x:VariableRefNode, y:VariableRefNode}	Y
P3 {XOR}	|o:|ObjectStructure, |init(), |op() on |m by :|o, |op() on |e by |m	Lifeline Role, Message Role, Message Role, Message Role	:ObjectStructure, [typeCheckAll(), {typeCheck() on a, typeCheck() on e}, {typeCheck() on x, {typeCheck() on y}], [generateCodeAll(), {generateCode() on a, generateCode() on e}, {generateCode() on x, generateCode() on y}], [prettyPrintAll(), {prettyPrint() on a prettyPrint() on e}, {prettyPrint() on x, prettyPrint() on y}]	Y
	|o:|ObjectStructure, |init(), |op() on |e by :|o	Lifeline Role, Message Role, Message Role	:ObjectStructure, [typeCheckAll(), None], [generateCodeAll(), None], [prettyPrintAll(), None]	N
	|op() on |m by |c, |op() on |e by |m	Message Role, Message Role	[None, {typeCheck() on x, {typeCheck() on y}], [None, {generateCode() on x, {generateCode() on y}], [None, {prettyPrint() on x, {prettyPrint() on y}]	N
	|op() on |e by |c	Message Role	[None], [None], [None]	N

(c) Visitor Problem IPS Applicability

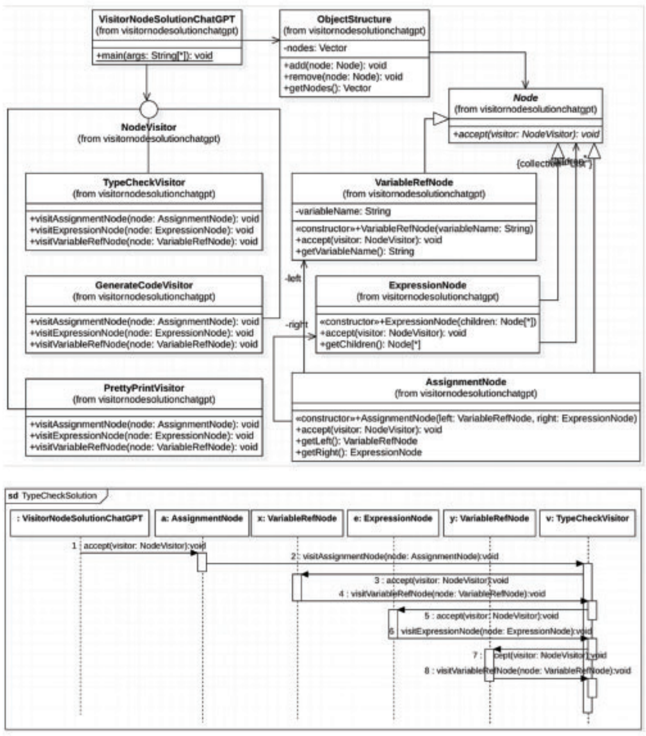
Figure 18: Visitor pattern applicability of node problem model

Fig. 18c shows that all three IPS properties are satisfied, leading to successful IPS applicability. P1 is satisfied by the $: VisitorNodeProblem$ lifeline playing the $|c : |Client$ role; P2 is satisfied by the $a : Assignment^*$ and $e : ExpressionNode^*$ lifelines enacting the $|m : |CompositeElement^*$ role and the $x : VariableRefNode$ and $y : VariableRefNode$ lifelines fulfilling the $|e : |LeafElement$ role; P3 is fulfilled by the first case, which involves the client delegating the $typeCheck()$ message through the object structure and composite components (assignment and expression nodes). This corresponds to the first case described in P2 of Fig. 15b. However, the second case fails due to the absence of model elements that can play the $|op()$ role on $|e$ called by $|c$. Similarly, the third case fails due to the absence of model elements that can enact the $|op()$ role on $|m$ called by $|c$. The fourth case fails due to the absence of model elements that can fulfill the $|op()$ role on $|e$ called by $|c$. The evaluation of both the SPS applicability and IPS applicability yields a 100% applicability rate for the Visitor pattern, calculated as $\frac{6+3}{6+3} \times 100$.

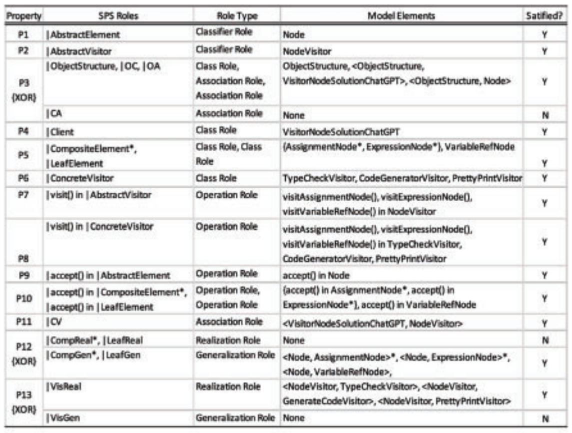
Fig. 19 displays the solution model with the application of the Visitor pattern. In the class diagram, the Visitor pattern introduces a visitor hierarchy that includes $NodeVisitor$, $TypeCheckingVisitor$, $GenerateCodeVisitor$, and $PrettyPrintVisitor$. Each visitor class has a set of operations carrying out the semantics of the visitor on different node types. The sequence diagram illustrates the type-checking behavior, specifying how the $accept()$ and $visit()$ messages are dispatched recursively across node objects by the visitor. Fig. 19b indicates full SPS conformance with all thirteen properties satisfied. P1 is satisfied by the $Node$ class enacting the $|AbstractElement$ role; P2 is satisfied by the $NodeVisitor$ class playing the $|AbstractVisitor$ role; P3 is satisfied by the first case, where the client accesses nodes via the object structure. The second case fails due to the absence of model elements that can fulfill theCA role; P4 is satisified by the $VisitorNodeSolutionChatGPT$ class fulfilling the $|Client$ role; P5 is satisfied by the $AssignmentNode^*$ and $ExpressionNode^*$ classes playing the $|CompositeElement^*$ role, and the $VariableRefNode$ class playing the $|LeafElement$ role; P6 is satisfied by the $TypeCheckingVisitor$, $CodeGeneratorVisitor$, and $PrettyPrintVisitor$ classes, each playing the $|ConcreteVisitor$ role; P7 is satisfied by the operations $visitAssignmentNode()$, $visitExpressionNode()$, and $visitVariableRefNode()$ in $NodeVisitor$, fulfilling the $|visit()$ role within $|AbstractVisitor$; P8 is satisfied by the operations $visitAssignmentNode()$, $visitExpressionNode()$, and $visitVariableRefNode()$ in $TypeCheckingVisitor$, $CodeGeneratorVisitor$, and $PrettyPrintVisitor$, which enact the $|visit()$ role within $|ConcreteVisitor$; P9 is satisfied by the $accept()$ operation in the $Node$ class, which plays the $|accept()$ role in $|AbstractElement$; P10 is satisfied by the $accept()$ operation in $AssignmentNode^*$ and $ExpressionNode^*$, playing the $|accept()$ role in $|CompositeElement^*$, and the $accept()$ operation in $VariableRefNode$, fulfilling the $|accept()$ role in $|LeafElement$; P11 is satisfied by the $< VisitorNodeSolutionChatGPT, NodeVisitor >$ relationship, playing the $|CV$ role; P12 is satisfied by the second case where elements are in a generalization hierarchy, with the $< Node, AssignmentNode >^*$ and $< Node, ExpressionNode >^*$ relationships playing the $|CompGen^*$ role and the $< Node, VariableRefNode >$ relationship fulfilling the $|LeafGen$ role. The first case fails due to the absence of model elements enacting the $|CompReal^*$ and $|LeafReal$ roles; P13 is satisfied by the first case where visitors are in a realization hierarchy, with the $< NodeVisitor, TypeCheckingVisitor >$, $< NodeVisitor, VariableRefVisitor >$, and $< NodeVisitor, ExpressionNodeVisitor >$ relationships playing the $|VisReal$ role. The second case fails due to the absence of generalization relationships that can play the $|VisGen$ role.

Fig. 19c reveals that only three out of four properties are fulfilled, resulting in 75% IPS conformance. P1 is satisfied by the $: VisitorNodeSolutionChatGPT$ lifeline playing the $|c : |Client$ role; P2 is satisfied by the $a : AssignmentNode^*$ and $e : ExpressionNode^*$ lifelines fulfilling the $|m : |CompositeElement^*$ role, and the $x : VariableRefNode$ and $y : VariableRefNode$ lifelines enacting the $|e : |LeafElement$ role; P3 is satisfied by the $: TypeCheckingVisitor$, $: CodeGeneratorVisitor$, and $: PrettyPrintVisitor$ lifelines playing the $|v : |ConcreteVisitor$ role; P4 is not satisfied, as none of the required conditions are met. The first case fails due to the absence of model elements capable of fulfilling the $|o : |ObjectStructure$ role, the $|init()$ role, and the $|accept()$ role on $|m$ called by $|o$. The second case fails due to the absence of model elements that can play $|o : |ObjectStructure$ role, the $|init()$ role, and the $|accept()$ role on $|e$ called by $|o$. The third case fails due to the absence of model elements that can fulfill the $|accept()$ role on $|e$ called by $|m$. The third case fails due to the absence of model elements that can enact the $|accept()$ role on $|e$ called by $|c$. The non-conformance of P4 arises from the visitor-driven recursion depicted in the sequence diagram. Typically, such recursion is managed by the object structure or the client, as outlined in P4's four cases, which offers better traversal control, particularly in complex structures. Overall, the evaluation of SPS and IPS conformance results in 94.12% ($\frac{13+3}{13+4} \times 100$) conformance to the pattern solution specification.

Fig. 20 demonstrates that the transformation rules for both SPS and IPS are fully enforced. Fig. 20a shows that for each of the three common operations in the node hierarchy—$typeCheck()$, $generateCode()$, $prettyPrint()$—a corresponding visitor class is created, ensuring the creation of all necessary visitor classes. Fig. 20b confirms that the double dispatch mechanism involving $accept()$ and $visit()$ is incorporated for each visitor, leading to a complete realization of the pattern behavior. This results in 100% ($\frac{3+2}{3+2} \times 100$) completeness of the pattern realization.

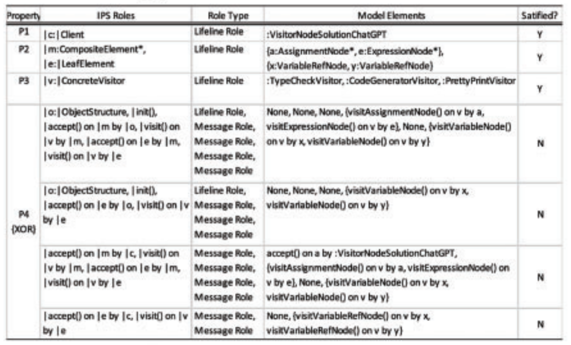

VisitorNodeSolutionChatGPT
(from visitornodesolutionchatgpt)
+main(args: String[*]): void
ObjectStructure
(from visitornodesolutionchatgpt)
-nodes: Vector
+add(node: Node): void
+remove(node: Node): void
+getNodes(): Vector
Node
(from visitornodesolutionchatgpt)
+accept(visitor: NodeVisitor): void
NodeVisitor
(from visitornodesolutionchatgpt)
TypeCheckVisitor
GenerateCodeVisitor
PrettyPrintVisitor
+visitAssignmentNode(node: AssignmentNode): void
+visitExpressionNode(node: ExpressionNode): void
+visitVariableRefNode(node: VariableRefNode): void
VariableRefNode
-variableName: String
«constructor»+VariableRefNode(variableName: String)
+accept(visitor: NodeVisitor): void
+getVariableName(): String
ExpressionNode
«constructor»+ExpressionNode(children: Node[*])
+accept(visitor: NodeVisitor): void
+getChildren(): Node[*]
AssignmentNode
«constructor»+AssignmentNode(left: VariableRefNode, right: ExpressionNode)
+accept(visitor: NodeVisitor): void
+getLeft(): VariableRefNode
+getRight(): ExpressionNode
-left
-right
sd TypeCheckSolution
: VisitorNodeSolutionChatGPT
a: AssignmentNode
x: VariableRefNode
e: ExpressionNode
y: VariableRefNode
v: TypeCheckVisitor
1 : accept(visitor: NodeVisitor):void
2 : visitAssignmentNode(node: AssignmentNode):void
3 : accept(visitor: NodeVisitor):void
4 : visitVariableRefNode(node: VariableRefNode):void
5 : accept(visitor: NodeVisitor):void
6 : visitExpressionNode(node: ExpressionNode):void
7 : accept(visitor: NodeVisitor):void
8 : visitVariableRefNode(node: VariableRefNode):void

(a) Node Solution Model



Property	SPS Roles	Role Type	Model Elements	Satified?
P1	|AbstractElement	Classifier Role	Node	Y
P2	|AbstractVisitor	Classifier Role	NodeVisitor	Y
P3 (XOR)	|ObjectStructure, |OC, |OA	Class Role, Association Role, Association Role	ObjectStructure, <ObjectStructure, VisitorNodeSolutionChatGPT>, <ObjectStructure, Node>	Y
	|CA	Association Role	None	N
P4	|Client	Class Role	VisitorNodeSolutionChatGPT	Y
P5	|CompositeElement*, |LeafElement	Class Role, Class Role	(AssignmentNode*, ExpressionNode*), VariableRefNode	Y
P6	|ConcreteVisitor	Class Role	TypeCheckVisitor, CodeGeneratorVisitor, PrettyPrintVisitor	Y
P7	|visit() in |AbstractVisitor	Operation Role	visitAssignmentNode(), visitExpressionNode(), visitVariableRefNode() in NodeVisitor	Y
P8	|visit() in |ConcreteVisitor	Operation Role	visitAssignmentNode(), visitExpressionNode(), visitVariableRefNode() in TypeCheckVisitor, CodeGeneratorVisitor, PrettyPrintVisitor	Y
P9	|accept() in |AbstractElement	Operation Role	accept() in Node	Y
P10	|accept() in |CompositeElement*, |accept() in |LeafElement	Operation Role, Operation Role	(accept() in AssignmentNode*, accept() in ExpressionNode*), accept() in VariableRefNode	Y
P11	|CV	Association Role	<VisitorNodeSolutionChatGPT, NodeVisitor>	Y
P12 (XOR)	|CompReal*, |LeafReal	Realization Role	None	N
	|CompGen*, |LeafGen	Generalization Role	<Node, AssignmentNode>*, <Node, ExpressionNode>*, <Node, VariableRefNode>,	Y
P13 (XOR)	|VisReal	Realization Role	<NodeVisitor, TypeCheckVisitor>, <NodeVisitor, GenerateCodeVisitor>, <NodeVisitor, PrettyPrintVisitor>	Y
	|VisGen	Generalization Role	None	N

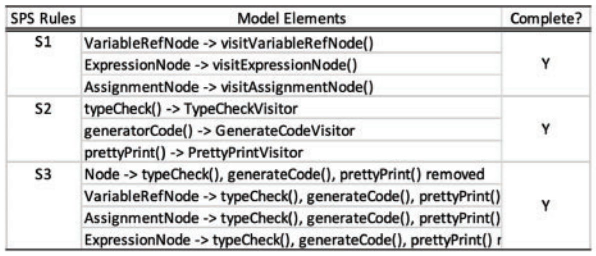
(b) Visitor Solution SPS Conformance



Property	IPS Roles	Role Type	Model Elements	Satified?
P1	|c:|Client	Lifeline Role	:VisitorNodeSolutionChatGPT	Y
P2	|m:CompositeElement*, |e:|LeafElement	Lifeline Role	{a:AssignmentNode*, e:ExpressionNode*}, (x:VariableRefNode, y:VariableRefNode)	Y
P3	|v:|ConcreteVisitor	Lifeline Role	:TypeCheckVisitor, :CodeGeneratorVisitor, :PrettyPrintVisitor	Y
P4 (XOR)	|o:|ObjectStructure, |init(), |accept() on |m by |o, |visit() on |v by |m, |accept() on |e by |m, |visit() on |v by |e	Lifeline Role, Message Role, Message Role, Message Role, Message Role	None, None, None, {visitAssignmentNode() on v by a, visitExpressionNode() on v by e}, None, {visitVariableNode() on v by x, visitVariableNode() on v by y}	N
	|o:|ObjectStructure, |init(), |accept() on |e by |o, |visit() on |v by |e	Lifeline Role, Message Role, Message Role, Message Role	None, None, None, {visitVariableNode() on v by x, visitVariableNode() on v by y}	N
	|accept() on |m by |c, |visit() on |v by |m, |accept() on |e by |m, |visit() on |v by |e	Message Role, Message Role, Message Role, Message Role	accept() on a by :VisitorNodeSolutionChatGPT, {visitAssignmentNode() on v by a, visitExpressionNode() on v by e}, None, {visitVariableNode() on v by x, visitVariableNode() on v by y}	N
	|accept() on |e by |c, |visit() on |v by |e	Message Role, Message Role	None, {visitVariableRefNode() on v by x, visitVariableRefNode() on v by y}	N

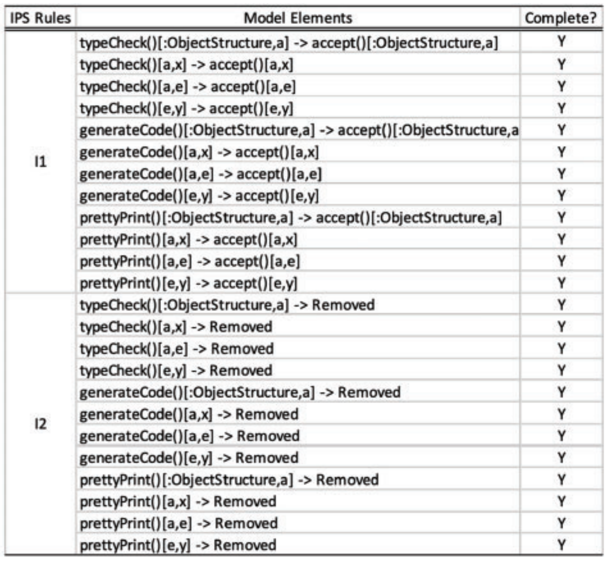
(c) Visitor Solution IPS Conformance

Figure 19: Visitor pattern conformance of node solution model



SPS Rules	Model Elements	Complete?
S1	VariableRefNode -> visitVariableRefNode()	Y
	ExpressionNode -> visitExpressionNode()	
	AssignmentNode -> visitAssignmentNode()	
S2	typeCheck() -> TypeCheckVisitor	Y
	generatorCode() -> GenerateCodeVisitor	
	prettyPrint() -> PrettyPrintVisitor	
S3	Node -> typeCheck(), generateCode(), prettyPrint() removed	Y
	VariableRefNode -> typeCheck(), generateCode(), prettyPrint()	
	AssignmentNode -> typeCheck(), generateCode(), prettyPrint()	
	ExpressionNode -> typeCheck(), generateCode(), prettyPrint() r	

(a) SPS Rules



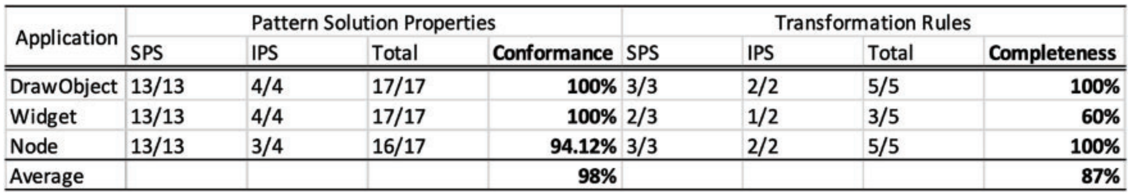
IPS Rules	Model Elements	Complete?
I1	typeCheck()[:ObjectStructure,a] -> accept()[:ObjectStructure,a]	Y
	typeCheck()[a,x] -> accept()[a,x]	Y
	typeCheck()[a,e] -> accept()[a,e]	Y
	typeCheck()[e,y] -> accept()[e,y]	Y
	generateCode()[:ObjectStructure,a] -> accept()[:ObjectStructure,a	Y
	generateCode()[a,x] -> accept()[a,x]	Y
	generateCode()[a,e] -> accept()[a,e]	Y
	generateCode()[e,y] -> accept()[e,y]	Y
	prettyPrint()[:ObjectStructure,a] -> accept()[:ObjectStructure,a]	Y
	prettyPrint()[a,x] -> accept()[a,x]	Y
	prettyPrint()[a,e] -> accept()[a,e]	Y
	prettyPrint()[e,y] -> accept()[e,y]	Y
I2	typeCheck()[:ObjectStructure,a] -> Removed	Y
	typeCheck()[a,x] -> Removed	Y
	typeCheck()[a,e] -> Removed	Y
	typeCheck()[e,y] -> Removed	Y
	generateCode()[:ObjectStructure,a] -> Removed	Y
	generateCode()[a,x] -> Removed	Y
	generateCode()[a,e] -> Removed	Y
	generateCode()[e,y] -> Removed	Y
	prettyPrint()[:ObjectStructure,a] -> Removed	Y
	prettyPrint()[a,x] -> Removed	Y
	prettyPrint()[a,e] -> Removed	Y
	prettyPrint()[e,y] -> Removed	Y

(b) IPS Rules

Figure 20: Visitor pattern completeness of node solution model

5.3 Discussion

In this subsection, we discuss the findings from the three case studies—the drawing application in Section 4, and the widget and node applications in Section 5. Fig. 21 presents the quantitative results of ChatGPT's capability in applying the Visitor pattern to these applications. The table indicates that ChatGPT achieves an average of 98% pattern conformance and 87% pattern completeness. In terms of pattern conformance, the sole instance of non-conformance occurred in the node application where the recursive behavior of $accept()$ and $visit()$ was managed by the visitor instead of the object structure or the client, as delineated in the Visitor solution IPS. Although this behavior is atypical, it might be seen as a variant. Nevertheless, it was marked as non-conformance since the pattern specification does not encompass this variation. Regarding pattern completeness, the only oversight was the absence of a visitor for the $simple()$ operation in the widget application, which could be attributed to a minor oversight by ChatGPT. In contrast, in the node application where there are three common operations in the node hierarchy, ChatGPT successfully generated three visitor classes, one for each operation. Besides the two non-conformance cases, the quantitative outcomes reveal ChatGPT's proficiency in design pattern application, which conclusively demonstrates the effectiveness of the proposed approach. Note that the case studies are not intended to provide statistical insight into ChatGPT, but to evaluate the approach's effectiveness.



Application	Pattern Solution Properties				Transformation Rules			
	SPS	IPS	Total	**Conformance**	SPS	IPS	Total	**Completeness**
DrawObject	13/13	4/4	17/17	**100%**	3/3	2/2	5/5	**100%**
Widget	13/13	4/4	17/17	**100%**	2/3	1/2	3/5	**60%**
Node	13/13	3/4	16/17	**94.12%**	3/3	2/2	5/5	**100%**
Average				**98%**				**87%**

Figure 21: The results of three case studies

The study's findings reveal that modern LLMs can effectively understand and apply complex software design patterns. However, the instances of non-conformance provide particularly interesting insights. For example, ChatGPT's decision to remove the $Shape$ class in the drawing application highlights how LLMs might make unexpected design decisions that, while functionally sound, may deviate from pattern specifications. Furthermore, ChatGPT's choice not to create a visitor for a $simple()$ operation demonstrates a sophisticated judgment about when the application of a pattern might introduce unnecessary complexity. These observations suggest that LLMs are capable of more than mechanically applying design patterns; they exhibit nuanced decision-making capabilities. However, this can sometimes lead to deviations from established pattern specifications. This delicate balance between adhering to patterns and making practical implementation decisions provides valuable insights for both the automated application of design patterns and the evolution of design pattern applications in practice.

This study introduces a novel quantitative framework for evaluating LLMs' capabilities in design pattern implementation, addressing a gap in current LLM'a capabili5y research in software engineering. Its key innovation lies in the systematic use of Role-Based Metamodeling Language (RBML) to create measurable criteria for pattern application, moving beyond subjective assessments to provide concrete metrics for conformance and completeness of design pattern implementation. For academia, this framework offers a methodology for comparing different LLMs to automated design pattern implementation, enabling more rigorous empirical studies in assessing LLMs' capability in design pattern tasks. On the practical side, software developers and organizations can use these metrics to make informed decisions about incorporating LLMs into their development workflows, particularly for design pattern-related tasks. The framework's ability to quantify pattern applicability (through problem specification), conformance (through solution specification), and completeness (through transformation rules) provides a comprehensive evaluation tool that bridges theoretical understanding with practical implementation, benefiting both researchers studying AI-assisted software engineering and practitioners seeking to optimize their development processes.

6 Conclusion

This paper has presented a quantitative approach to evaluating an LLM's ability to implement a design pattern within software applications. The approach utilizes RBML to formalize design patterns rigorously, which then serves as the basis for verifying pattern applicability, conformance, and completeness. Initially, an application without the pattern implementation is assessed for the applicability of the designated pattern. If deemed applicable, the problem model of the application is input into the LLM for pattern application. Subsequently, the LLM's output, a solution model with the pattern applied, is evaluated against the pattern's solution specification to determine the levels of pattern conformance and completeness. Through three case studies, ChatGPT was found to exhibit an average of 98% pattern conformance and 87% pattern completeness, thus demonstrating the efficacy of the proposed approach. Future work will extend this approach to additional LLMs, such as Gemini and CoPilot, for a comparative analysis, aiming for an unbiased comparison. Such expansion would help determine whether the observed performance metrics are model-specific or represent a general capability level of current LLMs in design pattern tasks. By examining variations in pattern implementation across models, researchers could better understand the relationship between an LLM's ability to apply design patterns. This cross-model analysis would be valuable for both developing specialized LLMs for design pattern tasks and refining prompting strategies for design pattern application.

Acknowledgement: The author expresses gratitude to all editors and anonymous reviewers for their insightful comments and suggestions, which have significantly enhanced the quality of this manuscript.

Funding Statement: This study did not receive any external funding.

Availability of Data and Materials: The data and materials supporting the findings of this study are openly available at https://github.com/hanbyul1/Design-Pattern-Refactoring (accessed on 31 January 2025).

Ethics Approval: Not applicable.

Conflicts of Interest: The author declares no conflicts of interest to report regarding the present study.

References

1. Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object-oriented software. Boston, MA, USA: Addison-Wesley; 1995.
2. France R, Ghosh S, Song E, Kim D. A metamodeling approach to pattern-based model refactoring. IEEE Softw, Special Issue on Model Driven Development. 2003;20(5):52–8. doi:10.1109/MS.2003.1231152.
3. OpenAI. ChatGPT. 2024 [cited 2025 Jan 31]. Available from: https://chat.openai.com/.
4. Google. Gemini. 2024 [cited 2025 Jan 31]. Available from: https://gemini.google.com/app.
5. GitHub. Github Copilot. 2021 [cited 2025 Jan 31]. Available from: https://copilot.github.com/.
6. Ozkaya I. Application of large language models to software engineering tasks: opportunities, risks, and implications. IEEE Softw. 2023;40(3):4–8. doi:10.1109/MS.2023.3248401.
7. Ahmad A, Waseem M, Liang P, Fahmideh M, Aktar MS, Mikkonen T. Towards Human-Bot collaborative software architecting with ChatGPT. In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering; 2023; Oulu, Finland. p. 279–85.

8. Zhang Q, Zhang T, Zhai J, Fang C, Yu B, Sun W, et al. A critical review of large language model on software engineering: an example from ChatGPT and automated program repair. arXiv:2310.08879. 2024.
9. Kirinuki H, Tanno H. ChatGPT and human synergy in black-box testing: a comparative analysis. arXiv:2401.13924. 2024.
10. Kim DK, Chen J, Ming H, Lu L. Assessment of ChatGPT's proficiency in software development. In: Proceedings of the 21st International Conference on Software Engineering Research & Practice; 2023; Las Vegas, NV, USA.
11. Eden AH, Jil J, Yehuday A. Precise specification and automatic application of design patterns. In: Proceedings of the IEEE International Conference on Automated Software Engineering; 1997; Incline Village, NV, USA. p. 143–52.
12. Lano K, Bicarregui J, Goldsack S. Formalising design patterns. In: Proceedings of the 1st BCS-FACS Northern Formal Methods Workshop, Electronic Workshops in Computer Science; 1996; Ilkley, UK.
13. Mikkonen T. Formalizing design patterns. In: Proceedings of the 20th International Conference on Software Engineering; 1998; Kyoto, Japan. p. 115–24.
14. Taibi T, Ngo DCL. Formal specification of design patterns—a balanced approach. J Object Technol. 2003;2(4):127–40. doi:10.5381/jot.2003.2.4.a4.
15. Guennec AL, Sunye G, Jezequel J. Precise modeling of design patterns. In: Proceedings of the 3rd International Conference on the Unified Modeling Language (UML); 2000; York, UK. p. 482–96.
16. Lauder A, Kent S. Precise visual specification of design patterns. In: Proceedings of the 12th European Conference on Object-Oriented Programming; 1998; Brussels, Belgium. p. 114–36.
17. Mapelsden D, Hosking J, Grundy J. Design pattern modelling and instantiation using DPML. In: Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS); 2002; Sydney, Australia: ACS. p. 3–11.
18. Kim D, France R, Ghosh S, Song E. Using role-based modeling language (RBML) as precise characterizations of model families. In: Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS); 2002; Greenbelt, MD, USA. p. 107–16.
19. White J, Hays S, Fu Q, Spencer-Smith J, Schmidt DC. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. arXiv:2303.07839. 2023.
20. Dakhel AM, Majdinasab V, Nikanjam A, Khomh F, Desmarais MC, Jiang ZMJ. GitHub Copilot AI pair programmer: asset or liability? J Syst Softw. 2023;203(1):111734. doi:10.1016/j.jss.2023.111734.
21. Solohubov I, Moroz A, Tiahunova MY, Kyrychek HH, Skrupsky S. Accelerating software development with AI: exploring the impact of ChatGPT and GitHub Copilot. In: Proceedings of the 11th Workshop on Cloud Technologies in Education; 2023; Kryvyi Rih, Ukraine. p. 76–86.
22. Nascimento N, Alencar P, Cowan D. Comparing software developers with ChatGPT: an empirical investigation. arXiv:2305.11837. 2023.
23. Surameery NMS, Shakor MY. Use chat GPT to solve programming bugs. Int J Inf Technol Comput Eng. 2023;3(1):17–22. doi:10.55529/ijitc.
24. Asare O, Nagappan M, Asokan N. Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code? Empir Softw Eng. 2023;28(6):129. doi:10.1007/s10664-023-10380-1.
25. Alshahwan N, Harman M, Harper I, Marginean A, Sengupta S, Wang E. Assured offline LLM-based software engineering. In: Proceedings of the ACM/IEEE 2nd International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering; 2024; Lisbon, Portugal. p. 7–12.
26. Marques N, Silva RR, Bernardino J. Using ChatGPT in software requirements engineering: a comprehensive review. Future Internet. 2024;16(6):180. doi:10.3390/fi16060180.
27. Rajbhoj A, Somase A, Kulkarni P, Kulkarni V. Accelerating software development using generative AI: ChatGPT case study. In: Proceedings of the 17th Innovations in Software Engineering Conference; 2024; Bangalore, India. p. 1–11.
28. Bera P, Wautelet Y, Poels G. On the use of ChatGPT to support agile software development. In: Proceedings of the 2nd International Workshop on Agile Methods for Information Systems Engineering; 2023; Zaragoza, Spain. p. 1–9.

29. Felizardo KR, Lima MS, Deizepe A, Conte TU, Steinmacher I. ChatGPT application in systematic literature reviews in software engineering: an evaluation of its accuracy to support the selection activity. In: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement; 2024; Barcelona, Spain. p. 25–36.
30. Özpolat Z, Yıldırım Ö., Karabatak M. Artificial intelligence-based tools in software development processes: application of ChatGPT. Eur J Tech. 2023;13(2):229–40. doi:10.36222/ejt.1330631.
31. Champa AI, Rabbi MF, Nachuma C, Zibran MF. ChatGPT in action: analyzing its use in software development. In: Proceedings of the 21st International Conference on Mining Software Repositories; 2024; Lisbon, Portugal. p. 182–6.
32. Xiao T, Treude C, Hata H, Matsumoto K. DevGPT: studying developer-ChatGPT conversations. In: Proceedings of the International Conference on Mining Software Repositories. MSR 2024; 2024; Lisbon, Portugal. p. 227–30.
33. Hassan AE, Oliva GA, Lin D, Chen B, Ming Z. Rethinking software engineering in the foundation model era: from task-driven AI Copilots to goal-driven AI pair programmers. arXiv:2404.10225. 2024.
34. Pudari R, Ernst NA. From Copilot to pilot: towards AI supported software development. arXiv:2303.04142. 2023.
35. Waseem M, Das T, Ahmad A, Liang P, Fahmideh M, Mikkonen T. ChatGPT as a software development Bot: a project-based study. In: Proceedings of International Conference on Evaluation of Novel Approaches to Software Engineering; 2024; Angers, France. p. 406–13.
36. Fan A, Gokkaya B, Harman M, Lyubarskiy M, Sengupta S, Yoo S, et al. Large language models for software engineering: survey and open problems. In: Proceedings of IEEE/ACM International Conference on Software Engineering: Future of Software Engineering; 2023; Melbourne, Australia. p. 31–53.
37. Nguyen-Duc A, Cabrero-Daniel B, Przybylek A, Arora C, Khanna D, Herda T, et al. Generative artificial intelligence for software engineering—a research agenda. arXiv:2310.18648. 2023.
38. Rahmaniar W. ChatGPT for software development: opportunities and challenges. IT Professional. 2024;26(3):80–6. doi:10.1109/MITP.2024.3379831.
39. Wang Y, Chen Y, Li Z, Tang Z, Guo R, Wang X, et al. Towards efficient and reliable LLM serving: a real-world workload study. arXiv:2401.17644. 2024.
40. Sridhara G, Ranjani HG, Mazumdar S. ChatGPT: a study on its utility for ubiquitous software engineering tasks. arXiv:2305.16837. 2023.
41. Kim DK. Source code. [cited 2025 Jan 31]. Available from: https://github.com/hanbyul1/Design-Pattern-Refactoring.