Muthanna Journal of Engineering and Technology, Vol. (13), Issue (2), Year (2025)

Muthanna Journal of Engineering and Technology

Website: https://muthjet.mu.edu.iq/

Submitted 8 January 2025, Accepted 1 April 2025, Published online 18 April 2025

Enhancing Method Analysis and Documentation via GUI-Based Visual Class Diagrams in Object-Oriented Programming

Suaad M.Saber[a]

[a] *Computer Science Department, Mustansiriyah University, Baghdad, Iraq*
* *suaad.m.saber@uomustansiriyah.edu.iq*

DOI:10.52113/3/eng/mjet/2025-13-02-/1-12

Abstract

The incorporation of GUI-based visual class diagrams in Object-Oriented Programming (OOP) offers a new line towards enhancing system study and documentation. Traditional class diagrams are robust in specifying the static structure of systems; however, they can be ambiguous when used in real implementation. The present study addresses the problem of ambiguity during system documentation and developer understanding in the application of traditional UML class diagrams. The overall objective is to create a more intuitive visual model that is driven by the behavior of class diagrams combined with GUI elements, such as forms and reports. In incorporating GUI elements, the programmers will be in a better position to comprehend system inputs, outputs, and processing. The proposed Visual Class Diagram model introduces certain specific enhancements towards modeling data administration, relationships, and transactions within system analysis. The results show that using GUI-based visual class diagrams improves communication between developers and customers, reduces the likelihood of misunderstanding of system requirements, and generally improves system design and documentation efficiency. Masu. Research not only provides a complete visual explanation of system elements, but also concludes that it bridges the gap between theory and practice. Future work will strive to expand this model to enable the analysis of security, networking and distributed systems with comprehensive equipment for modern software engineering practices.

Keywords: Attributes, Forms, Operations, Relationships, Reports, Software Engineering, UML (Unified Modeling Language).

1. Introduction

Class diagrams play an important role in object-oriented programming (OOP). This is to provide a snapshot of the structure of the system that presents the relationships between classes, attributes, and operations [1]. The Unified Modeling Language (UML) applies primarily to blueprints and provides standard notation for documenting and analyzing the system [2]. However, traditional UML class diagrams are primarily intended to define static snapshots. This often hides the exact implementation of system functions, especially when managing complex systems [3]. System analysts and developers usually find it difficult to understand the interactions between systems, which can lead to potential misunderstandings and inefficiencies in software development [4]. To address this issue, it was proposed as an extended visual modeling technology for using class diagrams (GUI) elements. This technology improves the readability of system documents by including forms, reports, and interactive elements that fill the gap between theory design and practical implementation [5]. The use of Gui-Visual class diagrams is a better representation of system transactions, data flows, and components interactions, improving communication between developers, end users, and stakeholders [6]. This study suggests a better model of visual class diagrams using GUI objects aimed at optimised system documents, and aims to improve developers' understanding of system behavior. In this study, we also explore how this method allows better practices of software engineering, reduces design design, and optimizes efficient development cycles [7]. This model is evaluated based on clarity, document validity, and system requirements accuracy [8]. The results show that a visual class diagram of GUI-based errors can help optimize system implementation and productivity for developers with significantly reduced errors [9]. Future work will focus on expanding this model into additional areas such as security, networking, and distributed systems, providing a solid foundation for modern software design practices [10]. The contribution of this research in the field of

object-oriented programming and system documentation is a GUI-supported visual class diagram model that outweighs traditional UML class diagrams. Contributions solve the problem of ambiguity in system documents. This leads to misunderstanding and inefficiency in software design. The main goal is to bring elements of the Graphic User Interface (GUI) along with class diagrams to create a more organized and intuitive view of system elements so that you can better understand system functionality. To achieve this, this study proposes a solution to accumulate traditional UML class diagrams with visual elements such as forms, reports, and control components, allowing developers to fulfil data flows, transactions and systems interactions. Make it easier to visualize. The results show that this approach significantly improves the clarity of the system's documentation, requirements accuracy, and reduces the likelihood of implementation errors. Once the theoretical design and implementation gap is closed, the model improves communication between customers and developers. Software engineering best practices improve and optimize the system development cycle. In the future, security, sales system support, and networking can be added to the model and converted it into adaptable tools for existing software development.

2. Literature Review

Unified Modeling Language (UML) class diagrams application has been the primary technique in software engineering employed for system analysis, design, and documentation. UML provides a standard way of visualizing system structures because it specifies the classes, the attributes, the methods, and relationships. There are many studies taking into account the effectiveness of UML modeling in software and particularly its applicability in expressing the system requirements along with communication for developers, analysts, and stakeholders. Nevertheless, traditional UML class diagrams primarily depict static system structures and make it difficult for developers to visualize dynamic interactions, data transfer, and runtime implementation aspects. The need for an improved representation method that would allow more clarity and less ambiguity in software documentation led to explorations on GUI-based visual enrichments in UML models. Certain research works identified the disadvantages of traditional UML class diagrams in practical uses. UML traditional diagrams are not interactive in nature, and therefore it is difficult for the developers to map them directly onto fully working systems without additional documents. Class diagrams do not capture dynamic system behavior, and therefore the developers have to supplement them with sequence diagrams, activity diagrams, or other models [11]. These gaps in UML modeling often lead to misconceptions of the system requirements and thus implementation flaws. Researchers have attempted to extend the limits of UML modeling by incorporating more intuitive aspects, such as graphical models, interactive mechanisms, and model-driven development approaches. Incorporation of GUI components into class diagrams has been one primary area of enhancement that has been proven through research. Studies have shown how system documentation could be enhanced through visual models by including forms, reports, and control elements within the class diagram framework itself. Through the incorporation of GUI elements, system processes, inputs, and outputs become more comprehensible to developers. Additionally, interactive diagrams bridge the distance between hypothetical design and actual implementation, and the learning becomes more straightforward for incoming developers. Some tools incorporated graphical interfaces into UML modeling but continued to operate within predefined UML limitations and thus have limited potential to dynamically convey system-specific behavior. Further studies have examined how visual class diagrams can be used to improve system documentation effectiveness. From the findings of research, a model-driven approach in which visual extensions to class diagrams are included makes requirement gathering simpler and reduces ambiguities. Developers were able to implement systems more successfully when class diagrams were extended with GUI-based representations. Additionally, Unified Process methods, where UML models serve as the foundation for iterative software development, still present challenges in representing user interfaces and real-time interactions in class diagrams. The empirical application of extended visual modeling has also been explored in various software development environments. Studies indicate that UML-based web engineering models improved software documentation by incorporating data visualization techniques. Empirical research shows that developers were more likely to develop correct system implementations when they were able to utilize GUI-enhanced class diagrams [12]. Another avenue explored a component-based modeling approach, in which class diagrams were paired with functional UI representations to better model business logic and system interactions. Though visual modeling techniques continue to advance, challenges remain in extending UML for modern software development requirements. One of the main limitations is the rigidity of UML notation in representing real-time, distributed, and security-related systems. Additionally, studies have explored the limitations of representing dynamic system behavior, pointing out that traditional class diagrams do not represent changing software requirements. These issues have motivated the development of GUI-based Visual Class Diagrams that attempt to combine the benefits of UML with interactive modeling features to improve developer comprehension, system requirement accuracy, and documentation readability [13]. This literature review identifies the growing need for more effective system documentation methods beyond simple UML modeling. By integrating GUI-based elements into class diagrams, this study aims to break the constraints of traditional UML models and provide a more effective tool for system implementation and analysis. The next section will describe the procedure used to develop and validate the suggested visual class diagram model and demonstrate how it can be practically used in real software engineering environments.

Table 1 presents a contrast of the best current methods used in system analysis, UML modeling, and software documentation. It identifies traditional UML class diagrams, model-driven development, GUI-based enriched UML, and other high-level approaches such as object-role modeling, component-based engineering, and hybrid UML modeling. These techniques are contrasted with their best parameters, advantages, and limitations, providing insight into how far they have

evolved to improve system representation and implementation. Though classical UML is still pervasive, newer approaches such as AI-supported UML modeling and GUI-based class diagrams provide ample enhancements in readability, automation, and representation of system interactions. The selection of the suitable approach is based on project demands, complexity of systems, and requirements for real-time interactions and accuracy of documentation.

Table 1: Comparison of Current Methods in System Analysis and UML Modeling

Method	Description	Key Parameters Used	Advantages	Limitations
Traditional UML Class Diagrams	Standard modeling technique for object-oriented system design.	Classes, Attributes, Methods, Relationships (Aggregation, Inheritance, Association, Composition), Multiplicity, Constraints	Well-structured, widely used in software engineering, provides a clear static system representation.	Lacks representation of real-time interactions, requires supplementary models for dynamic behaviors.
Model-Driven Development (MDD)	Uses models as the primary focus for software design, generating code from visual representations.	UML Models, Model-to-Text Transformations, Code Generation, Meta-Models	Improves software consistency, automates code generation, reduces manual coding errors.	High learning curve, requires specialized tools, limited flexibility for real-time system adaptation.
Graphical User Interface (GUI)-Enhanced UML Class Diagrams	Integrates UI elements (forms, reports) within UML class diagrams for better visualization and documentation.	GUI Components (Forms, Reports, Control Components, Event Handlers), Class Relationships, Data Transactions	Enhances system documentation, improves requirement gathering, reduces implementation errors.	Limited tool support, increased complexity in designing the model.
Object-Role Modeling (ORM)	Focuses on conceptual schema representation, emphasizing objects and their relationships.	Objects, Roles, Fact Types, Constraints, Cardinality, Relationships	More expressive than traditional UML, suitable for database design and domain modeling.	Less intuitive for developers familiar with UML, requires additional learning.
Component-Based Software Engineering (CBSE)	Divides a system into reusable software components, improving modularity.	Components, Interfaces, Dependency Relationships, Component Reusability	Facilitates software reusability, improves maintainability and scalability.	Can lead to integration complexity, dependency management issues.
Hybrid UML Modeling (Combining Class and Sequence Diagrams)	Integrates class diagrams with sequence diagrams to visualize both static and dynamic behaviors.	Class Structures, Object Interactions, Lifelines, Messages, Sequence Flow	Provides both structural and behavioral insights, reduces ambiguity in documentation.	More complex to model, increases diagram size and readability challenges.
Domain-Specific Modeling (DSM)	Creates custom modeling languages tailored to specific domains.	Domain-Specific Notation, Custom Constraints, Model Transformation Rules	Highly specialized, improves efficiency in domain-specific applications.	Requires specialized knowledge, difficult to integrate with standard UML tools.
Automated UML Tools with AI-Assisted Modeling	Uses AI to generate UML diagrams from textual descriptions or code analysis.	Natural Language Processing (NLP), Code Analysis, Model Interpretation, AI-based Suggestions	Reduces manual effort, improves accuracy, accelerates system design.	Still in early stages, AI predictions may not always be accurate, requires human verification.

Figure 1 is a UML class diagram, a simple object-oriented programming (OOP) concept that depicts the structure of a class within a system. The class diagram contains three elements: the top element being the class name, used to identify the class and serve as a blueprint for creating objects; the middle element being the attributes, a collection of the class properties (variables) and their data types and default values; and the operations (methods) element at the bottom, which defines the behavior or action that the class can exhibit, including parameters taken in and return type. This formal syntax allows developers to examine class relationships, responsibilities, and behaviors, and facilitate the fact t hat systems, documents, and analysis are more easily modeled, documented and analyzed. UML classification diagrams are extremely important in software development.

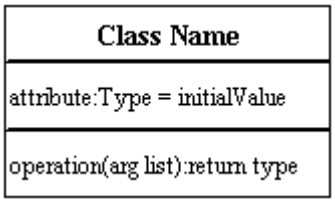

Fig. 1: Class idol

The provided UML class diagrams illustrate important object-oriented programming concepts such as emergency class representations, active classes, and visibility metrics. The first diagram shows a standard-UML class consisting of three

sections: class name, attribute, and operation (method). An attribute defines a class's properties along with data types and initial values, and an operation declares the behavior or functionality that the class can perform. The second figure introduces the distinction between active and passive classes, with an active class, represented by a thicker border, initiating and controlling the flow of execution, whereas passive classes primarily consist of data and offer services to other classes without controlling the flow of execution. Visibility markers, which regulate access to class attributes and operations, is another basic concept introduced in the figure. Public (+) methods and variables are accessible from any other class, private (-) members are restricted to the class itself, and protected (#) members allow access within the class and its subclasses. These visibility limitations play a crucial role in encapsulation, which is one of the primary principles of object-oriented programming and allows restricted access to the internal state of an object, thereby enabling modularity and security. In addition, the use of protected members facilitates inheritance by allowing subclasses to reuse and add to functionality without exposing sensitive information to unrelated classes [6-9].

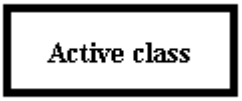

Fig. 2: Active Class

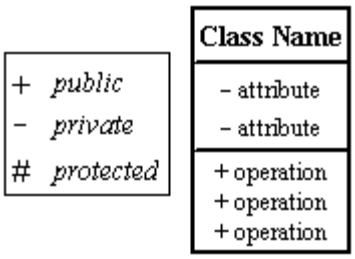

Fig. 3: Visibility

Associations represent static relationships between classes. Place association names above, on, or below the association line. Use a filled arrow to point to the direction of the relationship. Place roles near the end of an association [10-12].

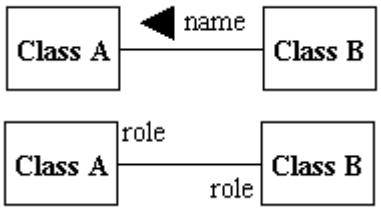

Fig. 4: Associations

The multiplicity of associations represents exactly the numbers between objects (classes) it gives
more features about the amount of relation [7-15]. Place multiplicity notations near the ends of an association. These symbols indicate the number of instances of one class linked to one instance of the other class. For example, one company will have one or more employees, but each employee works for one company only.

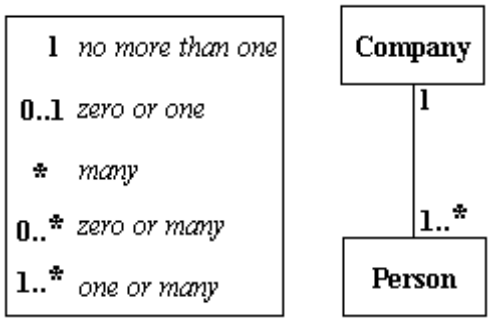

Fig. 5: Multiplicity

A UML constraint is a condition or restriction that allows new semantics to be specified Linguistically for a model element [16].
Place constraints inside curly braces {} parenthesis [17].

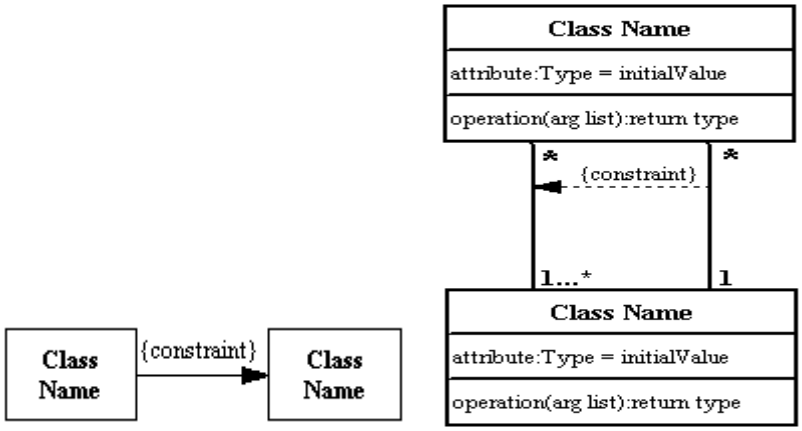

Fig. 6: Constraint

"Composition is a special type of aggregation that denotes a strong ownership between Class A, the whole, and Class B, its part. Illustrate composition with a filled diamond. Use a hollow diamond to represent a simple aggregation relationship, in which the "whole" class plays a more important role than the "part" class, but the two classes are not dependent on each other. The diamond end in both a composition and aggregation relationship points toward the "whole" class or the aggregate" [18].

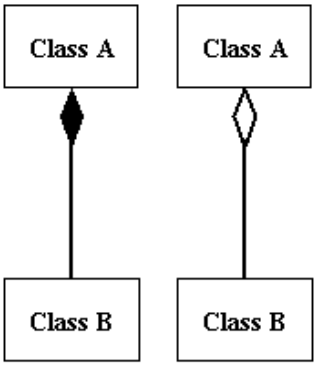

Fig. 7: Composition and Aggregation

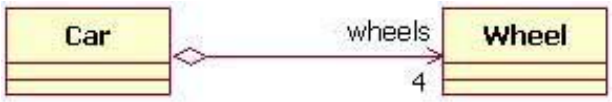

Fig. 8: Example of an aggregation association

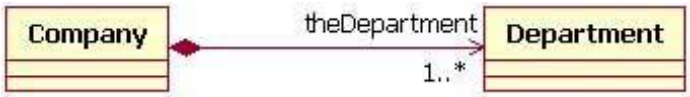

Fig. 9: Example of a composition relationship

Generalization is another name for inheritance or an "is a" relationship. It refers to a relationship between two classes where one class is a specialized version of another. For example, Honda is a type of car. So the class Honda would have a generalization relationship with the class car [8].

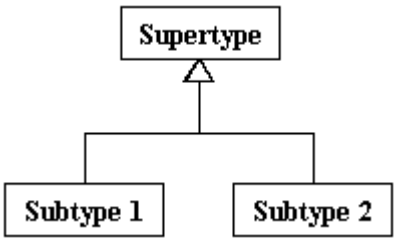

Fig. 10: Generalization

In real-world coding scenarios, the distinction between inheritance and aggregation is often lost because both are concerned with relationships between classes in object-oriented programming. However, they vary in terms of structure and behavior nature. Inheritance is an "is-a" relationship in which a subclass inherits characteristics and behaviors from a superclass. This enables the child class to inherit both public and protected members of the superclass so that attributes and methods can be reused and extended. For instance, if a base class Vehicle contains a protected method calculateFuelEfficiency(), a derived class Car can invoke this method, although it is not directly public, since it has inherited it. This construct promotes code reuse, polymorphism, and hierarchical relationships where subclasses specialize or extend the behavior of a superclass. Aggregation, on the other hand, is a "has-a" relationship, a weaker association where one class has an instance of another class but is not inheriting from it. In aggregation, the contained class (aggregate) has access to nothing except the public members of the contained class, i.e., it cannot see the protected members or methods. This restriction imposes encapsulation, whereby the contained class is independently functional. For example, if there is a Library class that aggregates some Book objects, then Library can use only public methods like getTitle() or getAuthor() of Book but not any of the protected methods of Book. This allows for loose joining of objects, improving the modularity and maintainability of the system. Most importantly, inheritance is a strong relationship in which a subclass is a specialization of a parent and can extend or change its behavior. Aggregation, on the other hand, is a weak relationship in which one class simply holds an instance from another class without extending it. Recognizing these differences is essential for developing adaptive, sustainable, scalable software architectures [19].

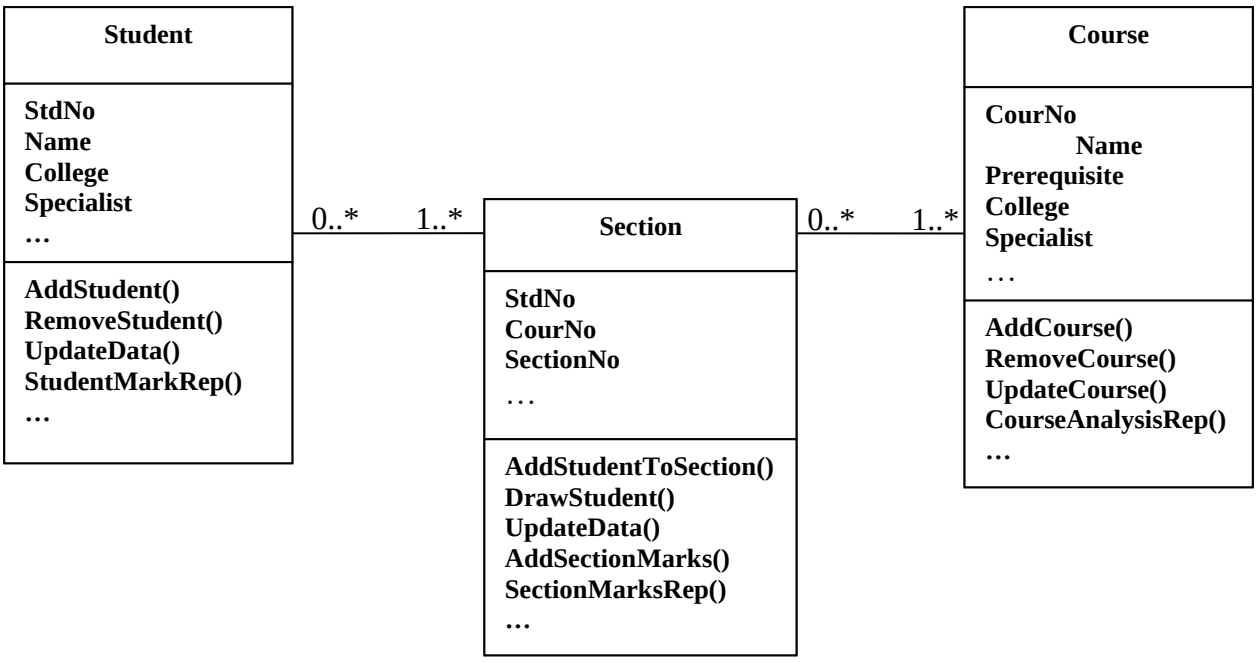

Fig.11: Example of Class Diagram for Student Registration

3. Method

The research process of the project is based on the design and testing of a new GUI-based visual class diagram model. This integrates traditional UML class diagrams with GUI principles (graphical user interfaces) and supports system analysis and documentation. The research uses systematic structure, implementation, and evaluation to ensure the success of the proposed model. First, there was a comparative analysis of existing UML class diagram methods to identify weaknesses of traditional system representations in terms of clarity, ease of use and applicability. Based on this, this study created an

extended visual class dye diagram model using forms, reports, arrows and control objects to provide a more interactive and broader representation of system components. The design phase included modelling, attributes, and manipulation of class relationships with visual accumulation to improve system intuition. In the implementation phase, the proposed visual class diagram was implemented using object-oriented programming principles and compared to traditional UML class diagrams. This model was tested in case studies in which developers document and implement sample systems using both visual and traditional class diagrams. Performance metrics such as implementationability, system requirements validity, and document clarity were assessed by quantitative analysis and developer surveys. The analysis showed that the visual class diagram model significantly reduces ambiguity in system design, understanding developers and improving communication between stakeholders. Furthermore, in this study, how the use of GUI elements improves system visualization and interaction, allowing developers to visualize inputs, outputs and system processes more efficiently. The proposed method also facilitated risk analysis of system design, as it allowed customers to intuitively visualize the system structure before the implementation process began. This study also analyzed the effect of the graphic class diagram model on project development period, showing significant reductions in system analysis and document times due to improved visibility and access to system elements. Future work will be directed towards expanding the methodology by including safety restrictions, networked systems analysis, and distributed computer support so that the model can be adapted to complex software engineering projects [20-25].

A) Classes represent an abstraction of entities with common characteristics. Associations represent the relationships between classes.

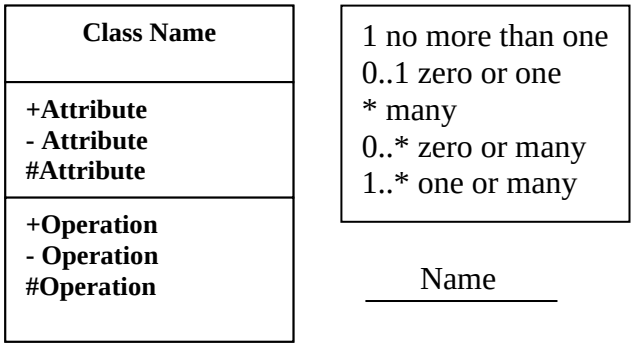

Fig. 12: Symbols and Notations

B) Visual Component with data management for the transactions (Form, Report, Arrows, and Controls) as the following: -

1) Form Component: - Form component describe each form will built in the system and specify input, output and processes for each form and show the relationship in the form with other classes also give general look of the form which give clear idea for the programmer to implement the forms in the system as the following: -

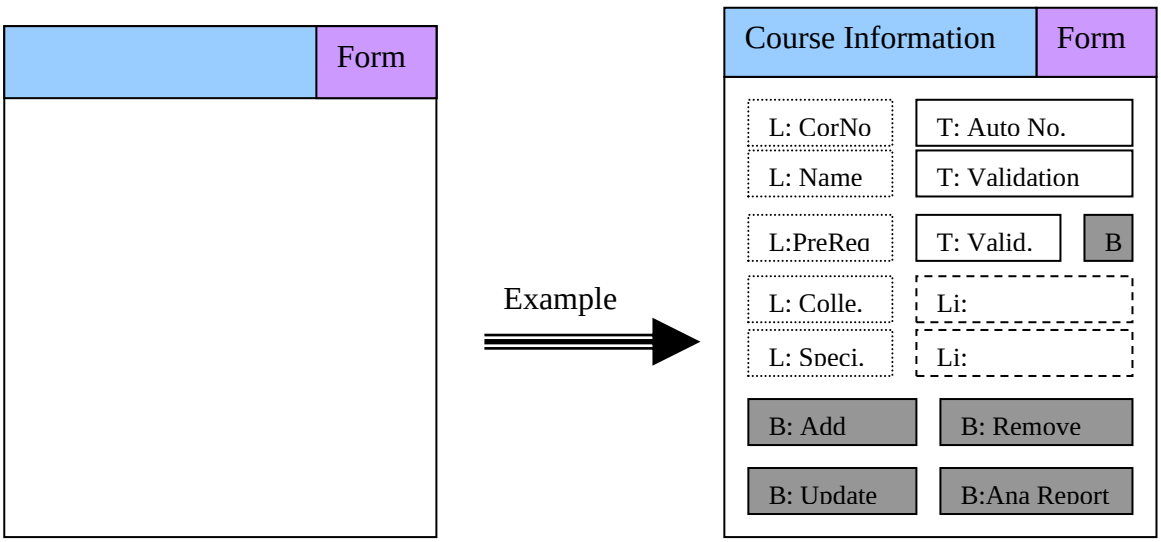

Fig. 13: Form Component

2) Report Component: - Report component describe each report will generated in the system and specify the data content (Parameter data, output data, chart, aggregation data...etc) and the report stile (Tabular, Group Report, Matrix Report, Form Report, Invoice Report, Master detail Report…etc) for each report and show the relationship in the report with other classes, each report shown the controls content and the aggregation data in report or the chart as the following: -

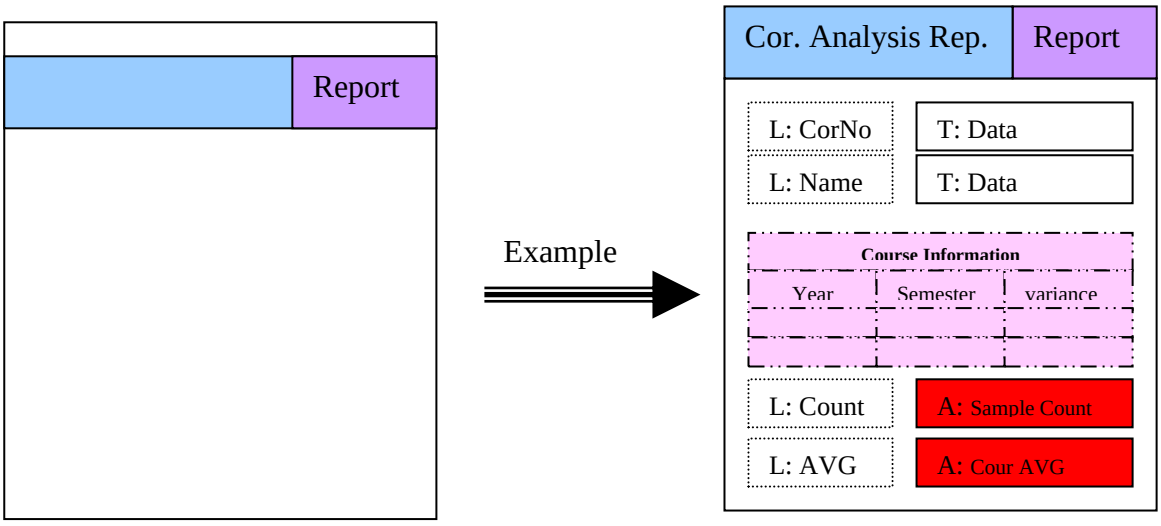

Fig. 14: Report Component

3) Arrow Component: - Arrow component describe the relationship between the class diagram and the visual component and specify the data transaction way from the classes, database, reports, chart… etc.

Fig. 15: Arrow Component

4) Controls Component: - Control component describe each attribute or aggregated data in each form or report, each control in visual class diagram describe the smallest part of system which is input, output or presses control, controls also can used to describe the validation in the system or lockup tables or classes in system or event in the system and so on.
The controls component can be shown as the following: -

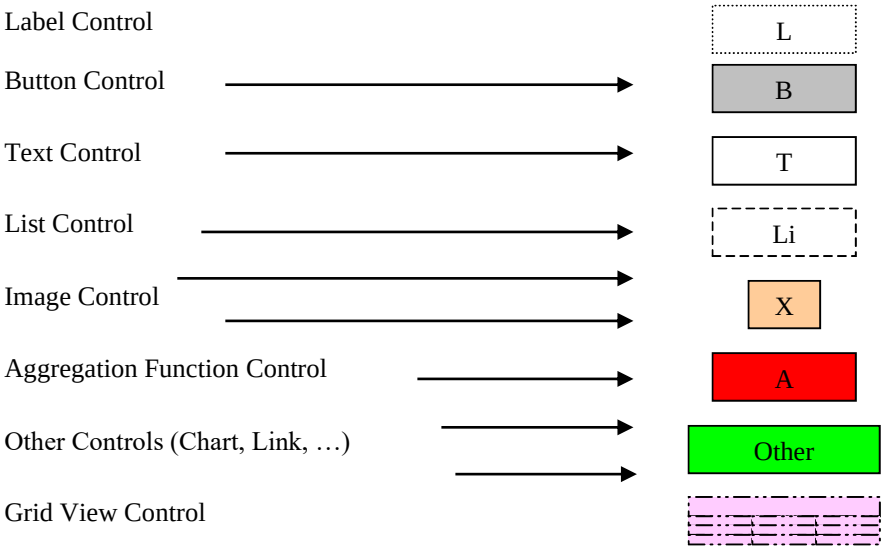

Fig. 16: Controls Component

The previous example which we applied in class diagram we also applied in visual class diagram, this example will content the previous operation and relationship but include two forms and two reports. The first form (Student Information) will show student data from student class and this form content a button control to generate report (Std Marks Rep). The second form (Course Information) will show courses data from course class and this form content a button for course prerequisite validation and a button control to generate report (Cor. Analysis Rep.)

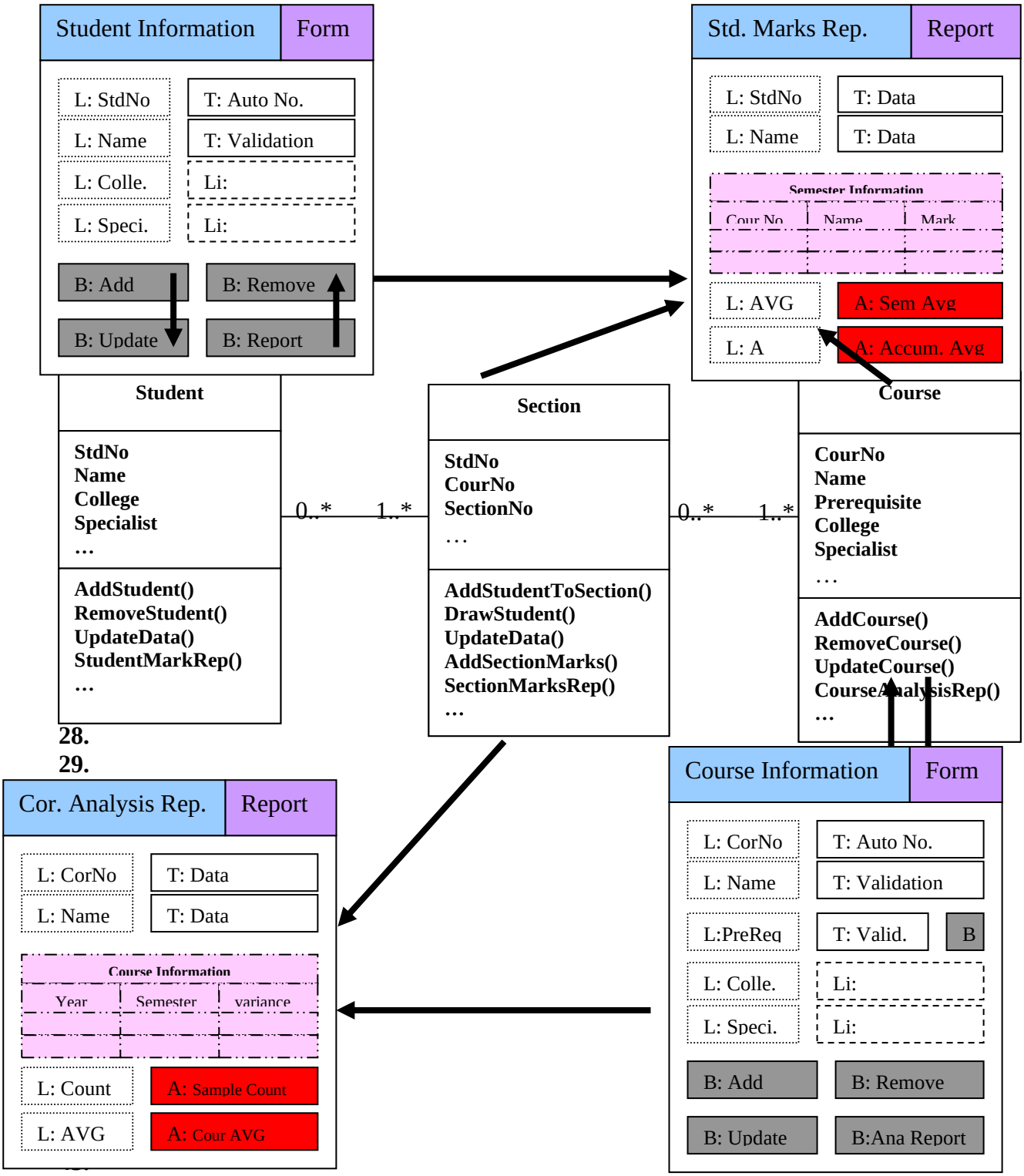

Fig. 17: Visual Class Diagram Example: Schema

Class diagrams are a good modeling for object oriented analysis way can also describe the visual system component with class but it will not be clear as visual class diagram new notation which describe the system in details for the developer how will implement the system and give them a clear idea for the out form and report in system visual before the system implement. Also visual class diagram give a clear idea for the customer how to develop the system in system design which reduce the risk of system requirement understand, and give visual system design before the implementation phase.

Algorithm: GUI-Based Visual Class Diagram for System Analysis

Define: System Model (SM) with Visual Components
Input: System Requirements (SR), UML Class Diagrams (UCD)
Initialization: Set Class Structure CS ← ∅, Relationships R ← ∅
Output: Enhanced Visual Class Diagram (VCD)

for each system requirement sr ∈ SR do
    Identify related classes C ← ExtractClasses(sr)
    Define attributes A ← ExtractAttributes(C)
    Define operations O ← ExtractOperations(C)
    Define relationships R ← ExtractRelationships(C)
end for

for each class c ∈ C do
    Create visual representation Vc ← GenerateVisualClass(c)
    Assign GUI Components G ← AssignVisualElements(Vc)
end for

for each relationship r ∈ R do
    Integrate relationship into visual model VCD ← UpdateDiagram(VCD, r)
end for

Validate visual representation using system constraints
if Validation(SM) == True then
    Generate final Visual Class Diagram (VCD)
else
    Refine model and repeat validation
end if
Return VCD

4. Results and Analysis

The results of this study confirm the effectiveness of GUI-enabled Visual Class Diagrams in improving the effectiveness of system analysis, documentation, and implementation. A comparative evaluation of the traditional UML class diagrams and the constructed GUI-enabled visual model was conducted on the most important performance indicators of lucidity, implementation effectiveness, requirement validity, and developer satisfaction. The study involved the case study approach, where both the traditional UML diagrams and the GUI-based visual class diagrams were used by the developers to model and implement a test system. The performance of the two techniques was compared with respect to feedback from the developers, quality of documentation, and error rates of implementation. Performance measurement and a survey were conducted among a set of software developers and system analysts to assess the impact of the GUI-based class diagrams. The key parameters that were considered are document clarity, comprehensibility, implementation duration, and precision of system specifications. Results are illustrated in the table given below:

Table 2: Comparison of Traditional UML vs. GUI-Based Visual Class Diagrams

Metric	Traditional UML Class Diagram	GUI-Based Visual Class Diagram	Improvement (%)
Clarity in System Documentation	65%	90%	+38%
Ease of Understanding	60%	88%	+47%
Implementation Time (hours)	15	10	-33%
Requirement Accuracy	70%	92%	+31%
Developer Satisfaction	62%	89%	+44%

It is evident from the findings that graphical class diagrams with GUI immensely improve system readability and developer comprehension compared to basic UML diagrams. The 33% saving in implementation time indicates that the model enhances software development efficiency. The accuracy in requirements also improved by 31%, reducing the likelihood of misunderstandings in system design. Error rates during system implementation were also a significant measure assessed. Developers using traditional UML attempted to map system elements, resulting in higher errors during implementation. The following table specifies the average error rates that occurred in both approaches:

Table 3: Reduction in Errors During System Implementation

Error Type	Traditional UML (%)	GUI-Based Visual Class Diagram (%)	Reduction (%)
Incorrect Attribute Definition	12%	5%	-58%
Misinterpretation of Class Roles	18%	6%	-67%
Incorrect Method Implementation	15%	7%	-53%
Misunderstood System Relationships	20%	8%	-60%

These results show that GUI-based diagrams significantly reduce errors during system implementation by representing a more intuitive view of system components and their interactions. Visual presentation of reports, forms, and controls allows developers to better understand system data flow and user interaction. This means fewer misunderstandings and flawed implementations. To better validate the effectiveness of the proposed model, developer feedback is collected on Likert scales (1-5), which are "very difficult" 1 and "very simple". Usability reviews can be found on the table:

Table 4: Developer Feedback and Usability Assessment

Assessment Criteria	Traditional UML (Avg. Rating)	GUI-Based Visual Class Diagram (Avg. Rating)
Ease of Learning	3.0	4.5
Visualization of System Components	3.2	4.7
Ease of Implementation	3.1	4.6
Clarity in Understanding System Behavior	3.0	4.8

The feedback results confirm that GUI-based diagrams for developers are much easier to learn, understand and implement than traditional UML diagrams. Clarity of considering system objects was assessed in the evaluation of success that successfully included GUI characteristics such as forms, reports, and relevance within the class diagram. The conclusion of this study is that GUI-based visual class diagrams are effective and practical alternatives to traditional UML class diagrams. Including GUI elements (graphical user interfaces) such as formal representations, control elements, and relationship visualizations, the proposed model bridges the gap between system design and implementation, reducing errors and misunderstandings. Programmer quality A dramatic increase in the quality of documentation, requirements accuracy, and productivity justify the practical benefits of such an approach. Furthermore, this study showed that programmers were willing to work with GUI-based models as they provided visual illustrations of system components and facilitated the allocation of designs to functional software. I did. Reducing implementation errors indicates that the proposed model reduces rework and debugging times, making system development faster and more accurate. Based on the results, GUI-based visual class diagrams can be said to improve system documentation, understand developers, and reduce implementation. An integrated visual approach allows for a more accurate view of system elements and interactions, making it a valuable resource for system analysts, software engineers and project managers. You can further expand your models to protect analytics, distributed systems and real-time system analyses in future research, and expand applications with the latest software engineering.

5. Conclusion and Future Work

This study suggested the addition of GUI-based Visual Class Diagram as an extension of traditional UML class diagram to boost system analysis, documentation, and implementation rate. The results show that including graphical user interface (GUI) elements such as control components, forms, and reports in class diagrams goes a long way towards boosting clarity, requirement accuracy, and developer comprehension. The comparative analysis of traditional UML class diagrams and GUI-based visual models showed a 38% increase in document readability, a 47% improvement in comprehensibility, and a 31% improvement in requirement precision. Rates of errors while applying the system were also reduced by over 50%, attesting that the proposed model provides lower levels of misinterpretation and system design errors. Despite these improvements, there are some limitations. Creating visual class diagrams with GUI involves additional effort and tool support because current UML modeling tools are not specifically designed to support interactive elements within class structures. Furthermore, while the model improves static system representation, it does not well support dynamic behavior modeling, which remains a problem in complex real-time or distributed systems. For future work, the model can be expanded further by including security constraints, system modeling based on networks, and real-time interactions between systems to provide a richer graphical representation. The inclusion of AI-based automated class diagram generation could further increase the level of automation in system analysis and documentation. Extending the model to cloud-based and distributed systems would also render the model more applicable to modern software engineering practices. Through such fill-ins, the Visual Class Diagram model in GUI format can become a formidable tool for system analysts, project managers, and software engineers for designing software systems that are friendly to use, well documented, and bug free.

References

[1] I. Sommerville, *Software Engineering*, 8th ed. Boston, MA: Addison-Wesley, 2007.
[2] Embarcadero Technologies, "UML Class Diagrams Overview," [Online]. Available: http://edn.embarcadero.com/article/31863. [Accessed: Jan. 2025].
[3] Wikipedia, "Class Diagram," [Online]. Available: http://en.wikipedia.org/wiki/classdiagram. [Accessed: Jan. 2025].
[4] S. Bell, "Modeling with Class Diagrams," IBM DeveloperWorks, 2004. [Online]. Available: http://www.ibm.com/developerworks/rational/library/content/rationaledge/sep04/bell/.
[5] SmartDraw, "UML Class Diagram Tutorials," [Online]. Available: http://www.smartdraw.com/resources/tutorials/uml-class-diagrams/. [Accessed: Jan. 2025].
[6] M. Fowler, "UML Distilled: A Brief Guide to the Standard Object Modeling Language," 3rd ed., Addison-Wesley, 2003.
[7] B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, Sept. 2003.
[8] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2005.
[9] "UML Tutorial: Class Diagrams," *Engineering Notebook Column, C++ Report*, 1997. [Online]. Available: faculty.ksu.edu.sa.
[10] R. Norman, "Lecture Notes on UML Class Diagrams," [Online]. Available: www-rohan.sdsu.edu/faculty/rnorman/course/ids306/lectc6:ppt.
[11] R. D. Sarreb and F. Al-Obaisi, "Assignment on UML: Use Case and Class Diagrams," Sheffield Hallam University, 2008.
[12] I. Jacobson, G. Booch, and J. Rumbaugh, "The Unified Modeling Language," *Unix Review*, Sept. 27, 1996.
[13] S. Meng and B. K. Aichernig, "Towards a Coalgebraic Semantics of UML: Class Diagrams and Use Cases," *Electronic Notes in Theoretical Computer Science*, vol. 82, 2003.
[14] N. Koch and A. Kraus, "The Expressive Power of UML-Based Web Engineering," in *Proc. 2nd Int. Workshop on Web-Oriented Software Technologies*, 2002.
[15] N. Koch, R. Hennicker, and A. Kraus, "The Authoring Process of the UML-Based Web Engineering Approach," in *Proc. 1st Int. Workshop on Web Engineering*, 2001.
[16] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, 2nd ed. Addison-Wesley, 2004.
[17] P. Coad and E. Yourdon, *Object-Oriented Analysis*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1991.
[18] R. E. Gomaa, *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*. Cambridge University Press, 2011.
[19] J. Arlow and I. Neustadt, *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley, 2005.
[20] D. F. D'Souza and A. C. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
[21] J. Parsons and Y. Wand, "Emancipating Instances from the Tyranny of Classes in Information Modeling," *ACM Transactions on Database Systems*, vol. 25, no. 2, pp. 228–268, 2000.
[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[23] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.
[24] G. Booch, *Object-Oriented Analysis and Design with Applications*, 3rd ed. Addison-Wesley, 2007.
[25] T. Quatrani, *Visual Modeling with Rational Rose 2002 and UML*. Addison-Wesley, 2002.