Sohar University Journal of Engineering and Information Technology Innovations (SUJEITI), ISSN: 3078-7513
Volume 1, Issue 1, pp 26-36, January 2025, DOI: xxxxxxxxxxxxxxx
Received: 16/7/2024 Revised: 1/9/2024 Accepted: 30/12/2024

A Standardized Textual Language for UML Modeling: Review and Suggestions

Khaled Abuhmaidan
Faculty of Computing and IT, Sohar University, Oman, khmaidan@su.edu.om.

Abstract

Using Unified Modeling Language (UML) notations and tools to represent structure, behavior, architecture, and business processes is crucial when specifying newly developing systems. UML has become the de facto standard in software engineering and development. Numerous tools support UML modeling, helping project managers, system analysts, business analysts, and architects create UML diagrams through both text-based instructions and visual drawing tools. These tools often have their own unique notations for text-based instructions.
In this paper, we review a selection of popular UML textual notations and introduce a new formalized textual language called Standard Textual Unified Modeling Language (STUML). STUML is designed to standardize the way UML notations are expressed in text-based tools. It can be used directly or mapped to the notations of existing textual and graphical tools. This new language is intended for various types of diagrams, including class and use case diagrams, and is crafted to be easy to write, easy to read, and usable without requiring special editors. STUML has the potential to become the standard for textual UML notations.

Keywords: UML modeling, Text-Based Notation, Class Diagram, Use case diagram, Graphical notations, Textual UML, UML.

1. Introduction

Modeling is a crucial part of the software development process because it describes and explains the software structure and its operations (Fan et al., 2023). Modeling is defined as the means used to capture ideas, define relationships, and analyze requirements by applying well-defined notations, namely modeling languages. Modeling languages are used to create display, store, and exchange models (K. Goher, & A. Al-Ashaab, 2021). UML (Unified Modeling Language) is a general-purpose modeling language and is considered the standard in the software engineering field (M. Mazanec & O. Macek, 2012; Sommerville, 1989). UML modeling is utilized across all stages of the software engineering process, from requirements and design to the maintenance phase (M. Ozkaya & F. Erata, 2020).

UML (UML, 2017) is the most widely used modeling approach in software engineering. Moreover, UML has become the dominant notation for software engineering modeling and is unlikely to be replaced soon (M. Fowler, 2018). It includes various types of diagrams used to model the structure and behavior of systems. UML Diagrams are classified into two main divisions: Structural (e.g., class and package diagrams) and Behavioral (e.g., use case and activity diagrams). Among the set of UML diagrams, the class diagram, use case diagram, and sequence diagram are the most frequently used (M. R. Chaudron, 2012).

Due to UML's popularity as a modeling language, numerous graphical tools exist to draw and represent these diagrams. Examples of commonly used tools include StarUML, IBM Rational Rose Enterprise Edition, SmartDraw, GenMyModel, and Microsoft Visio. On the other hand, there are also many text-based UML tools that rely on textual notations to express UML diagrams, such as yUML (yUML, 2017), MetaUML, and UMLGraph (UMLGraph, 2017).

Three different types of modeling tools are used today: those using only graphical notations, those using text-based notations, and those using a combination of both (H. Grönninger et al., 2014). When reviewing papers presenting graphical tools, a prevalent assumption is that graphical tools are superior simply because they are graphical. However, this assumption is questioned by some researchers (Addazi and F. Ciccozzi, 2021). Defining a textual language is generally easier than defining a graphical notation, and because of its interoperability, transparency, extensibility, and ease of testing, textual notations are preferred by many developers (M. Winikoff, 2005). For instance, Essalmi, Fathi, and Leila Jemni Ben Ayed (F. Essalmi & L. J. B. Ayed, 2006) used Extended Backus-Naur Form (EBNF) to represent grammar using class diagrams. Ibrahim, Noraini, et al. (N. Ibrahim et al., 2011) provided a formal definition of use case diagrams to check consistency between use case and activity diagrams. Michael Winikoff (M. Winikoff, 2005) extended UML by proposing textual notations to model agent systems, termed AUML (Agent UML). Hans Gronniger, et al. (H. Grönninger et al., 2014), reviewed the advantages and disadvantages of textually based notations. Earl Grey (EG) (M. Mazanec and O. Macek, 2012) introduced a modeling language but did not formalize it, comparing existing modeling languages based on defined features suitable for general-purpose textual languages. Frédéric Jouault and Jérôme Delatour (F. Jouault and J. Delatour, 2014) proposed tUML, a textual UML language to fix incomplete UML diagrams and identify errors and inconsistencies, facilitating model validation and verification. An executable textual language (txtUML) was proposed in (G. Dévai et al., 2014), enabling execution and debugging at the model level using a Java runtime environment.

Despite the popularity of graphical tools, many developers prefer textual notations due to several attractive properties. Advantages of textual notations from a developer's perspective can be summarized as follows:

- Content: Textual notations require less space to display the same content compared to graphical descriptions. While graphical notations provide an overview of the diagram, developers often seek detailed information easily found in textual notations. Textual notations are also simpler to print compared to graphical models.
- Speed of Drawing: Graphical tools may be simpler and faster for inexperienced users, but for experienced users, textual notations can expedite the creation of graphical models, especially with features like autocomplete and instance checking.
- Formatting: Text-based notations delegate formatting and layout processes to algorithms, relieving developers of time-consuming tasks.
- Independence: Text-based notations are platform-independent and do not require specific tools, allowing use with any text editor.

However, due to the diversity of modeling tools, many text-based modeling tools use different notational languages to represent standard UML diagrams (Petrausch, V et al.,2016). Developers using a particular tool must learn its specific notation language, which can hinder interoperability between tools. Furthermore, many notational languages are limited to specific UML diagram domains, such as class diagrams or sequence diagrams, without a comprehensive textual UML notation (S. Seifermann & H. Groenda, 2016).

In this paper, we introduce STUML (Standard Textual UML), a text-based modeling language defining notations for any modeling tool wishing to express UML diagrams textually using a text-based editor and automatically draw specified UML diagrams. In developing this notation, we aimed to apply design principles from (M. R. Chaudron et al., 2012; K. Goher et al., 2021) to ensure ease of readability, writing, and compatibility with various tools.

This paper is organized as follows: Section 2 provides background on text-based notations and modeling, along with examples of existing text-based tools. Section 3 describes the proposed notational language STUML. Section 4 includes the formal representation of the proposed notation. Finally, conclusions are presented in Section 5.

2. Textual Based Modeling

As mentioned before, there are many existing text-based UML tools. In this section, we review many of these tools and give examples of the set of the most promising tools among them.

2.1. Text-Based UML tools

Textual UML tools utilize a set of notations to describe the semantics of UML diagrams and automatically draw the graphical UML diagram from that description. These tools are more convenient for developers and easier to integrate with other systems such as version control applications. Table 1 lists a set of textual tools that are either already available online, downloadable, or usable as plugins with other tools such as Eclipse. Furthermore, we specify the diagrams targeted by each textual tool and their properties. As seen in Table 1, there is no standard textual language that helps increase interoperability between these notations and tools. Furthermore, the diversity and heterogeneity of these notations are high. Some are comprehensive but hard to read, while others are simple and easy to write, yet not suitable for large models.

2.2. Textual Modeling Languages

2.2.1. yUML

yUML (yUML, 2017) is a free online tool for drawing class, activity, and use case UML diagrams without the need for any graphical or drawing tools. Many tools help integrate yUML with wikis, emails, and blogs. It is used for lightweight UML design; however, it is not appropriate for all projects.
Figure 1 shows an example of drawing a class diagram using the notational language of yUML.

The online editor allows you to enter your code, which is automatically rendered to produce the required graphical notations. yUML is fast and suitable for small projects, but the code can be hard to read due to the many symbols. Additionally, yUML cannot be used to model all UML diagrams.

Table 1: Review of the most popular UML textual tools

Ref.	Tool	Diagrams	Properties and shortcomings
(Mazanec, M., et al., 2012)	EG	• Class Diagram • State Diagram	• Eclipse Plugin. • Grammar. • Readable and simple. • A limited number of diagrams.
(Winikoff, M., 2005)	AUML	• Agent UML	• UML Extension. • Simple. • Textual notations for AUML protocols.
(PlantUML, 2017)	PlantUML	• Sequence diagram • Usecase diagram • Class diagram • Activity diagram • Component diagram • State diagram • Object diagram • Deployment diagram • Timing diagram	• Online editor. • Create diagrams from java code. • Generate png images for the created diagrams. • Supports tools to generate LaTeX and SQL. • Grammar. • It becomes hard to read for large models. • Sometimes confusing and verbose.
(Create UML, 2017)	yUML	• Class diagram • Activity diagram • Use cases diagram	• Fast and easy. • It can be embedded in wikis and blog posts. • It can be formatted as pdf, jpg, and other formats. • Confusing and hard for large models.
(Nomnoml, 2017)	Nomnoml	• Class Diagram • Use case Diagram	• Saved on the client side. • Download diagrams as images. • Notations to change font and style .
(UMLGraph, 2017)	UMLGraph	• Class diagram. • Sequence diagrams	• Java code style. • The notations are restricted to the class and sequence diagrams.
(TextUM, 2017)	TextUML	• Structural Notations. • Behavioural Notations	• Open source IDE. • Compatible with tools such as eclipse.

(Zenuml, 2017)	ZenUML	• Sequence Diagram	• Fast and easy way to draw sequence diagrams. • Limited to sequence diagrams only.
(Jouault, F., & Delatour, J., 2014)	tUML	• Class diagrams • Composite structure diagrams • State diagram	• It is used for validation and verification.
(Dévai, G. et al., 2014)	txtUML	• Class Diagram • State Diagram • Activity Diagram	• Executable UML modelling. • Embedded Language. • Hosted in Java.

```
//Class Diagram
[Customer|-forname:string;surname:string;salary:double|doShiz()]<>-orders*>[Order]
[Order]++-0..*>[LineItem]
[Order]-[note:Aggregate root{bg:wheat}]
```

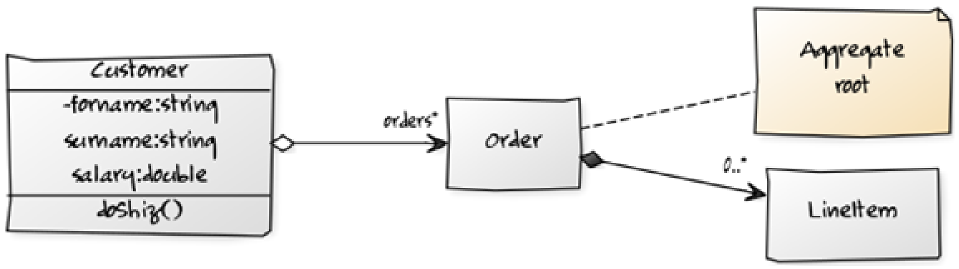

Figure 1: An example of drawing a class diagram using the notational language of yUML

2.2.2. UMLgraph

A tool makes use of language to specify and draw UML diagrams. It gives the ability to model class diagrams and sequence diagrams. It uses Java syntax to specify class diagrams, as depicted in Figure 2, and function-like pic macros to specify sequence diagrams. The problem is that it uses Java syntax for class diagrams with UMLGraphdoclet for drawing and pic macros for sequence diagrams, making the notations restricted to these languages. Additionally, the notations are limited to class and sequence diagrams (UMLGraph, 2017).

```
class Person {
        String Name;
}
class Employee extends Person { }
class Client extends Person { }
```

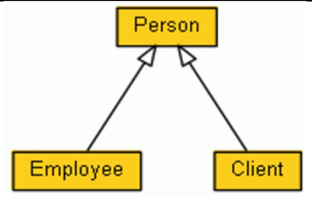

Figure 2. UMLGraph tool

3. Standard Textual UML (STUML)

In this section, the properties for the proposed notations are presented as a unified modeling notation for drawing tools. This notation, as previously mentioned, is called Standard Textual Unified Modeling Language (STUML). STUML will follow the recommendations proposed in (Mazanec, M., & Macek, O., 2012). As mentioned earlier, the importance of defining text-based notations for UML modeling stems from the fact that textual languages are easier to formalize, independent of tool capabilities, can be easily embedded into existing tools, and are preferred by many developers because they are similar to programming languages (H. Grönninger et al., 2014; M. Mazanec & O. Macek, 2012).

In our text-based language, we will not focus on features such as type declaration, variable declaration, or object orientation because the purpose is not to build a programming language. Instead, we will focus on properties such as readability, simplicity, and unambiguity.

- Wide Scope: STUML should be able to provide the necessary structures to describe UML models, whether they are static (Structural Diagrams) or dynamic (Behavioral Diagrams).
- Simplicity: The language should be easy to read and understand, leveraging the advantages of text-based notations, which give developers more detailed insights into the diagram. It should also be easy to write, encouraging developers to use it instead of graphical modeling, which is often more visually appealing (L. Addazi & F. Ciccozzi, 2021). The simplicity of the language comes from eliminating complicated symbols to represent relationships, such as those used in yUML and MetaUML. Instead, simple, well-known expressions such as 'implements' and 'extends' are used to express class inheritance and interface implementation.
- In STUML, we aimed to use a structure similar to the syntax used in many popular programming languages, such as C#, C++, and Java. This similarity will make the language easy to read and write, as developers will feel familiar with it and will not face difficulty using it. It will also be easy to learn.
- Unambiguous: The language should be unambiguous to avoid misinterpretation and to ease the development of tools that will adopt STUML as a textual modeling language. The notations will not produce more than one parse tree for the same expressions, ensuring clarity and precision.

4. Notations

Context-Free Grammar (CFG) is one way to describe the possible strings that might be produced by a formal language. STUML will be described by CFG so that the language can be used by any tool that employs text-based notation to draw UML diagrams (Aho, A. V et al, 2007).

A context-free grammar is a 4-tuple G=(V, Σ, R, S), where:

- **Terminals (Σ)**: Σ is a finite set of terminals that represent the basic set of symbols from which the strings are constructed.
- **Non-terminals (V)**: V is a finite set of non-terminals. This set helps in defining the language.
- **Start symbol (S)**: One non-terminal is specified as the start symbol.
- **The productions of the grammar (R)**: R is a finite set of production rules. Each production consists of a non-terminal, an arrow (→), and a body of zero or more terminals or non-terminals. Production rules can be mathematically formalized as a pair (α,β) where $\alpha \in V$ and $\beta \in (V \cup \Sigma)^*$ (F. Essalmi & L. J. B. Ayed, 2006).

4.1. Notational Conventions

To distinguish between terminals and non-terminals and to ease the reading of the production rules, the following conventions will be used throughout the rest of the paper:

- Anything *italic* and starting with an uppercase letter is a non-terminal.
- The letter S is the start of the grammar.
- Anything **bold** is a terminal.
- Digits and letters are terminals.
- Punctuation symbols such as parentheses, commas, and so on are terminals.
- The symbol **|** means "or."

4.2. Class Diagram

The class diagram is a fundamental diagram of UML and one of the most commonly used in software development. It describes the classes and interfaces existing in the system. Class diagrams not only show the properties, features, variables, and operations of classes and interfaces but also depict the relationships between entities, such as generalization and association.

The generalization relationship indicates a connection between a superclass and a subclass, where the subclass inherits the properties of the superclass. We can say that the subclass is a specialization of the superclass. Another important relationship is the association, which identifies a link between objects (D. Pilone & N. Pitman, 2005).
The class diagram has many components, including:

- Classes
- Abstract classes
- Interfaces
- Properties defined using names, types, and values
- Operations (methods)
- Generalization relationships
- Association relationships (including aggregation and composition)
- Visibility

4.3. Use Case Diagram

A use case diagram is used to expose the relationships between actors and use cases in a system. It is widely used to model the functional requirements of the system and is considered an effective way to communicate with stakeholders (G. Zhang et al., 2013; Nagy, B et al., 2023; Abuhmaidan, K et al., 2024).
A use case diagram consists of a set of objects: actors, use cases, and relationships. An actor can be a person, organization, thing, or external system that interacts with the system and can initiate a use case (D. Pilone & N. Pitman, 2005). In use case diagrams, actors are usually represented as stick figures, while use cases are depicted using ovals. Use cases represent the functionality of the system, encapsulating a set of actions that the system performs. Relationships in use case diagrams can exist between actors, between actors and use cases, and between use cases themselves (F. Essalmi & L. J. B. Ayed, 2006).

Several types of relationships can be used in use case diagrams:

- **Association** between an actor and a use case.
- **Association** between one use case and another use case:
    - **Extend**
    - **Include**
- **Generalization** between actors.
- **Generalization** between use cases.

4.4. Grammar

This subsection defines a context-free grammar (CFG) for a textual language used to model UML diagrams. The grammar specifies the syntax rules for constructing valid UML diagrams, including class diagrams and use case diagrams.

Figure 3 shows a high-level diagram illustrating the key components and their relationships within the grammar. This visual representation clarifies how different parts of the grammar interact. The diagram progresses from the start symbol ‘S’ to ‘Diagram’, which then branches into ‘classDiagram’ or ‘usecaseDiagram’. The ‘classDiagram’ further breaks down into ‘ClassModel’ and then into ‘Objects’, which include ‘Class’, ‘Interface’, and ‘Relation’. The ‘usecaseDiagram’ breaks down into ‘UseCaseModel’ and further into blocks.

As stated earlier, the grammar is presented in terms of four sets: terminals (Σ), non-terminals (V), production rules (R), and a start symbol (S). By applying the previously listed conventions, we present the grammar by listing the necessary productions. Any digits, signs such as < and -, and boldface strings such as start are terminals.

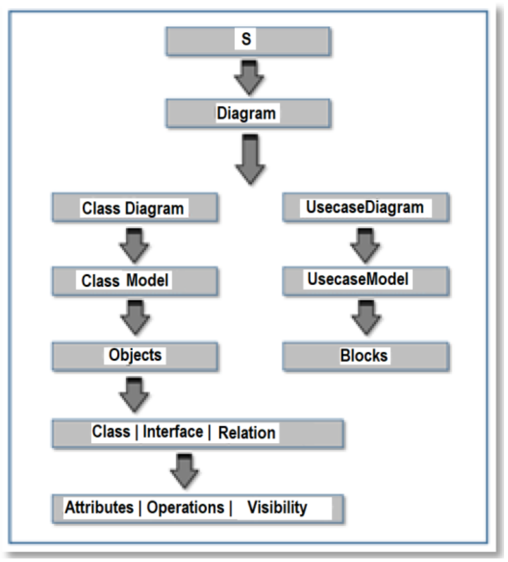

Figure 3: Aims to visually represent the hierarchical structure and relationships within the grammar.

An italicized name is a non-terminal (F. Essalmi & L. J. B. Ayed, 2006). Our grammar G = (V, Σ, R, S) consists of the four basic sets mentioned above and detailed below:

V = { Diagram, classDiagram, usecaseDiagram, Objects, Object, Class, Interface, Relation, Modifier, id, CAttributes, COperations, CAtts, Att, Visibility, AttDef, Type, Ops, Op, OpType, OpParams, params, intAttributes, intOperations, intAtts, intOps, IntOp, Binary, Composition, Many, BinaryRelation, RelName, CompType, side, Role, Arity, ArityType, Number, sides, Digit, Letter, characters, character, blocks, Entities, Entity, Actors, UseCases, URelationships, ActorRel, UseRel, ActUseRel }

Σ = {start, end, ClassModel, class, {, }, Attributes, -, private, protected, public, Integer, Real, Boolean, String, Operations, (,), ;, interface, }, void, abstract, [,], extends, implements, relates, Composite, Aggregate, .., *, connects, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A…Z, a…z, UseCaseModel, block, actor, usecase, isa, include, extend, uses}

R = {

S → **start** Diagram **end**
Diagram → classDiagram | usecaseDiagram

classDiagram → **ClassModel** Objects | ε
Objects → Objects Object | ε
Objects → Class | Interface | Relation
Class → Modifier **class [** id **] {**
CAttributes
COperations
}
CAttributes → **Attributes -** CAtts - | ε
CAtts → CAtts Att | Att
Att → Visibility AttDef ;
Visibility → **private** | **protected** | **public**
AttDef → Type **:** id
Type → **Integer** | **Real** | **Boolean** | **String** | id | ε
COperations → **Operations -** Ops | ε
Ops → Ops Op | ε

Op → Visibility Modifier OpType id **(**OpParams**);**
OpParams → params | ε
params → params **,** AttDef | AttDef
Interface → **interface [**id**]{**
    intAttributes
    intOperations
}
intAttributes → **Attributes -** intAtts **-** | ε
intAtts → intAtts AttDef | AttDef **;**
intOperations → Operations **–** intOps **-** | ε
intOps → intOps IntOp | intOp
intOp → opType id **(**OpParams**)**
OpType →Integer | Real | Boolean | String | void |*id* | *ε*
Modifier → **abstract** | ε
Relation → Binary | Composition | Many
Binary → BinaryRelation RelName **[**id**][**id**]**
BinaryRelation → **extends** | **implements** | **relates**
RelName → **[**id**]** | ε
Composition → CompType side side
CompType → **Composite** | **Aggregate**
Side → <id Role Arity>
Role → **,**id | ε
Arity → **;** ArityType | ε
ArityType →* | Number|Number..Number|Number..*|*..Number|*..*
Many → **connects** sides
sides → sides side | side side
Number → Number Digit | Digit
id → Letter characters
characters → characters character | ε
character → Letter | Digit
Digit → **0|1|2|3|4|5|6|7|8|9**
Letter → **A|B|…|Z|a|..|z | blank**
Extending the Grammar for use case diagram
UseCaseDiagram → **UseCaseModel** blocks | ε
blocks → **block [**id**]** Entities blocks | Entities
Entities → Entities Entity | Entity
Entity → Actors | UseCases | URelationships
Actors → **actor [**id**]**
UseCases → **usecase [**id**]**
URelatioships → ActorRel | UseRel | ActUseRel
ActorRel→ **[**id**] isa [**id**]**
UseRel → **(**id**)isa(**id**) | (**id**)include(**id**) | (**id**)extend(**id**)**
ActUseRel →**[**id**]uses(**id**)**
}

S = {*S*}

4.4.1 Examples of Using the Grammar

Example 1: Class Diagram

The following example defines a class ‘Person’ with attributes ‘name’ and ‘age’, and operations ‘setName’ and ‘getName’. To illustrate how to use the grammar to define a simple class diagram.

```
start classDiagram end

classDiagram → ClassModel Objects
ClassModel → classDiagram | ε

Objects → Class | Interface | Relation

Class → Modifier class [ id ] {
                CAttributes
                COperations
}

CAttributes → Attributes - CAtts - | ε
CAtts → CAtts Att | Att
Att → Visibility AttDef ;
Visibility → private | protected | public
AttDef → Type : id
Type → Integer | Real | Boolean | String | id | ε
COperations → Operations -  Ops | ε
Ops → Ops Op | ε
Op → Visibility Modifier OpType id (OpParams);
OpParams → params | ε
params → params , AttDef | AttDef

class [Person] {
  Attributes -
    private String : name;
    private Integer : age;
  Operations -
    public void : setName(String name);
    public String : getName();
}
```

Example 2: Use Case Diagram

The following example defines a use case diagram with a ‘User’ actor who interacts with ‘Login’ and ‘Register’ use cases. It illustrates how to use the grammar to define a simple use case diagram

```
start usecaseDiagram end

usecaseDiagram → UseCaseModel blocks | ε

blocks → block [id] Entities blocks | Entities
Entities → Entities Entity | Entity
Entity → Actors | UseCases | URelationships
Actors → actor [id]

UseCases → usecase [id]
URelatioships → ActorRel | UseRel | ActUseRel

ActorRel → [id] isa [id]
UseRel → (id)isa(id) | (id)include(id) | (id)extend(id)

ActUseRel → [id]uses(id)
UseCaseModel → usecaseDiagram | ε

block [System] {
  actor [User]
  usecase [Login]
  usecase [Register]
  [User] uses (Login)
  [User] uses (Register)
}
```

5. Conclusions

Text-based modeling is a powerful approach that many users prefer for its appealing features. There are already several tools available that use different notational languages for UML modeling. However, these tools are not compatible with each other, as each one uses its own unique notation. In this paper, we reviewed some of the most popular existing textual tools, highlighting their strengths and weaknesses. We also introduced a new textual language, Standard Textual Unified Modeling Language (STUML), designed to serve as a standard for UML text-based modeling tools. STUML is easy to write, read, and learn, making it an attractive option. In this paper, we focused on applying STUML to class diagrams and use case diagrams. In the future, we plan to expand STUML's grammar to cover a broader range of UML modeling diagrams.

Acknowledgements

The authors express their gratitude to their colleagues at the Faculty of Computing and IT at Sohar University for their valuable guidance.

Author contribution: Single author manuscript.
Conflict of interest: The author declares no conflict of interest.

References

[1]. Abuhmaidan, K. H., Al-Share, M. A., Abualkishik, A. M., & Kayed, A. (2024). Enhancing data protection in digital communication: A novel method of combining steganography and encryption. *KSII Transactions on Internet and Information Systems*, *18*(6), 1619–1637. https://doi.org/10.3837/tiis.2024.06.001

[2]. Addazi, L., & Ciccozzi, F. (2021). Blended graphical and textual modelling for UML profiles: A proof-of-concept implementation and experiment. *Journal of Systems and Software, 175*, 110912. https://doi.org/10.1016/j.jss.2021.110912
[3]. Aho, A. V., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, techniques, and tools* (2nd ed.). Addison-Wesley.
[4]. Chaudron, M. R., Heijstek, W., & Nugroho, A. (2012). How effective is UML modeling? *Software and System Modeling, 11*(4), 571–580. https://doi.org/10.1007/s10270-010-0172-7
[5]. Create UML diagrams online in seconds, no special tools needed. (2017). yUML. http://yuml.me/
[6]. Dévai, G., Kovács, G. F., & An, Á. (2014). Textual, executable, translatable UML. In *Proceedings of OCL@MoDELS 2014*.
[7]. Essalmi, F., & Ayed, L. J. B. (2006). Graphical UML view from extended Backus-Naur form grammars. In *Proceedings of the Sixth International Conference on Advanced Learning Technologies*.
[8]. Fan, B., Gokkaya, M., Harman, M., Lyubarskiy, S., Sengupta, S., Yoo, S., & Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. *arXiv Preprint*. https://arxiv.org/abs/2310.03533
[9]. Fowler, M. (2018). *UML distilled: A brief guide to the standard object modeling language* (3rd ed.). Addison-Wesley Professional.
[10]. Goher, K., Shehab, E., & Al-Ashaab, A. (2021). Model-based definition and enterprise: State-of-the-art and future trends. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture, 235*(14), 2288–2299. https://doi.org/10.1177/0954405421992574
[11]. Grönninger, H., Krahn, H., Rumpe, B., Schindler, M., & Völkel, S. (2014). Text-based modeling. *arXiv Preprint*. https://arxiv.org/abs/1409.6623
[12]. Ibrahim, N., Ibrahim, R., Saringat, M. Z., Mansor, D., & Herawan, T. (2011). Consistency rules between UML use case and activity diagrams using logical approach. *International Journal of Software Engineering and Its Applications, 5*(3), 119–134.
[13]. Jouault, F., & Delatour, J. (2014). Towards fixing sketchy UML models by leveraging textual notations: Application to real-time embedded systems. In *Proceedings of OCL@MoDELS 2014* (pp. 73–82).
[14]. Mazanec, M., & Macek, O. (2012). On general-purpose textual modeling languages. In *Proceedings of DATESO*.
[15]. Nagy, B., Abuhmaidan, K., & Aldwairi, M. (2023). Logical conditions in programming languages: Review, discussion, and generalization. *Annales Mathematicae et Informaticae, 57*, 65–77.
[16]. nomnoml: A tool for drawing UML diagrams. (2017). nomnoml. http://www.nomnoml.com
[17]. Ozkaya, M., & Erata, F. (2020). A survey on the practical use of UML for different software architecture viewpoints. *Information and Software Technology, 121*, 106275. https://doi.org/10.1016/j.infsof.2020.106275
[18]. Petrausch, V., Seifermann, S., & Müller, K. (2016). Guidelines for accessible textual UML modeling notations. In *Proceedings of the International Conference on Computers Helping People with Special Needs*.
[19]. Pilone, D., & Pitman, N. (2005). *UML 2.0 in a nutshell*. O'Reilly Media.
[20]. PlantUML - Open-source tool that uses simple textual descriptions to draw UML diagrams. (2017). PlantUML. http://plantuml.com
[21]. Seifermann, S., & Groenda, H. (2016). Survey on textual notations for the Unified Modeling Language. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*.
[22]. Sommerville, I. (1989). *Software engineering*. Addison-Wesley Longman Publishing Co.
[23]. TextUML: TextUML Toolkit is an open-source IDE for UML. (2017). TextUML. http://abstratt.github.io/textuml/readme.html
[24]. UMLGraph - Declarative Drawing of UML Diagrams. (2017). *UMLGraph*. http://www.umlgraph.org/
[25]. Unified Modeling Language (UML). (2017). UML. http://www.uml.org
[26]. Winikoff, M. (2005). Towards making Agent UML practical: A textual notation and a tool. In *Proceedings of the Fifth International Conference on Quality Software (QSIC 2005)*.
[27]. Zenuml. (2017). Zenuml. https://www.zenuml.com
[28]. Zhang, G., Yue, T., Ali, S., & Wu, J. (2013). An extensible use case modeling approach for cyber-physical systems (CPSs). In *Proceedings of Demos/Posters/Student Research@MoDELS*.