systems

MDPI

Article

Text-to-Model Transformation: Natural Language-Based Model Generation Framework

Aditya Akundi [1,*], Joshua Ontiveros [2] and Sergio Luna [3]

[1] Industrial and Manufacturing Engineering Department, College of Engineering and Applied Sciences, University of Wisconsin Milwaukee, Milwaukee, WI 53211, USA
[2] College of Engineering, University of Texas Rio Grande Valley, Edinburg, TX 78539, USA; joshua.ontiveros01@utrgv.edu
[3] Industrial, Manufacturing and Systems Engineering Department, College of Engineering, University of Texas El Paso, El Paso, TX 79936, USA; salunafong@utep.edu
* Correspondence: akundivy@uwm.edu

Abstract: System modeling language (SysML) diagrams generated manually by system modelers can sometimes be prone to errors, which are time-consuming and introduce subjectivity. Natural language processing (NLP) techniques and tools to create SysML diagrams can aid in improving software and systems design processes. Though NLP effectively extracts and analyzes raw text data, such as text-based requirement documents, to assist in design specification, natural language, inherent complexity, and variability pose challenges in accurately interpreting the data. In this paper, we explore the integration of NLP with SysML to automate the generation of system models from input textual requirements. We propose a model generation framework leveraging Python and the spaCy NLP library to process text input and generate class/block definition diagrams using PlantUML for visual representation. The intent of this framework is to aid in reducing the manual effort in creating SysML v1.6 diagrams—class/block definition diagrams in this case. We evaluate the effectiveness of the framework using precision and recall measures. The contribution of this paper to the systems modeling domain is two-fold. First, a review and analysis of natural language processing techniques for the automated generation of SysML diagrams are provided. Second, a framework to automatically extract textual relationships tailored for generating a class diagram/block diagram that contains the classes/blocks, their relationships, methods, and attributes is presented.

Keywords: MBSE; machine learning; SysML; class diagram; block definition; systems architecture; system model; systems design

check for updates

Citation: Akundi, A.; Ontiveros, J.; Luna, S. Text-to-Model Transformation: Natural Language- Based Model Generation Framework. *Systems* **2024**, *12*, 369. https://doi.org/10.3390/systems12090369

Received: 12 June 2024
Revised: 3 September 2024
Accepted: 6 September 2024
Published: 14 September 2024

1. Introduction

Natural language processing (NLP) assists in extracting and analyzing text data gathered from documents to help specify assembling, classifying, and recording requirements. Although NLP has many advantages in machine learning and software requirement specification, it can be highly ambiguous and complex when analyzing and extracting requirements from a text. Various tools help reduce errors and issues when extracting information from text documents. These techniques enable the development of algorithms for machines to process and understand human language, which benefits information retrieval, translation, data analysis, and extraction using NLP techniques. Significant advancements in NLP have been made in areas such as word segmentation, part-of-speech tagging, and syntactic analysis [1].

Unified modeling language (UML) and system modeling language (SysML) are visual modeling languages used to design, specify, and document system artifacts. UML is widely used to create models of software systems. In contrast, SysML v1.6, an extension of UML, is used for systems engineering applications to support the specification, requirements, analysis, design verification, and validation of systems and systems of systems [2]. It is critical to identify that SysML v1.6 and UML are similar but do not share an identical graphical modeling language, each having unique implementation and modeling requirements.

Typically, systems engineers manually synthesize engineering documentation, such as requirement documents, to generate system architectures for a streamlined systems development process, widely considered robust. However, the manual process of developing SysMLv1.6 diagrams can vary in consistency, leading to various interpretations and representations by a systems modeler. Furthermore, creating a model, such as identifying elements and choosing how they are connected using SysML, also highly depends on a modeler's experience.

Integrating natural language processing (NLP) with SysML can enhance engineering systems' design experience. A few attempts include aspects of NLP for requirements' elicitation, tracing, and classification, with a substantial need to preprocess and structure input data [3]. Three approaches to implementing NLP in creating system models are explored, namely rule-based, machine learning-based, and hybrid [4,5]. A rule-based approach implies a set of predefined rules that analyze and transform information from text-based requirements to create a system model. The machine learning approach involves training a model on large datasets of natural language text using algorithms to recognize elements and generate diagram components from a structured text representation [6,7]. A hybrid approach combines rule-based and machine learning techniques; it can use a rule-based approach to identify relationships and entities from NL text and use a machine learning model to generate a diagram. The integration of these approaches needs careful attention, considering the risk of automation bias. Several attempts have been observed to introduce automated processes aiming to standardize and expedite the creation of architectural diagrams from text-based requirements. However, it is essential to acknowledge the potential for automation bias in such scenarios. This paper proposes a text-to-model transformation (T2M) framework that allows a modeler to be flexible in mitigating these errors where selecting elements from an input text is subjective to the modeler's experience. The proposed framework does not create new knowledge but transforms information from one form to another. The framework accommodates a range of input texts, from formal statements of stakeholder needs or requirements to potentially less structured, free descriptions of the intended system.

Please note that from here onward, the term SysML in this paper refers to SysMLv1.6. Further, A class diagram in UML and a block definition diagram in SysML similarly represent a system's structural aspects. A class is a primary block representing a system's attributes (properties) and operations (methods). At the same time, a block is a core element in a block definition diagram representing properties (attributes) and operations (methods). Considering their notation, both utilize rectangles to define the system elements. Names, attributes, and operations are categorized within these rectangles, with lines and arrows depicting relationships. Though UML class diagrams and SysML block definition diagrams are defined in different contexts, their foundation stems from their similarities, considering SysML is an extension of the system engineering domain from UML. The text-to-model framework applied to the case studies illustrated in this paper—generates a class diagram. The similarities between the UML Class Diagram and the SysML block definition diagram led to the assumption of the framework's applicability to generate the SysML block definition diagram.

This paper first briefly reviews (Section 2) the NLP techniques using rule-based, machine learning-based, and hybrid approaches in generating baseline models that could act as a starting point for a system design. Subsequently, Section 3 introduces a rule-based text-to-model transformation framework to create a class/block definition diagram from text-based input, followed by case studies used to implement the framework and a discussion of the metrics for measuring the approach's effectiveness in Section 4. The challenges, roadblocks, and the path forward are discussed in the concluding sections.

2. A Review of NLP-Based Systems Model Generation

2.1. Rule-Based Approach

The rule-based approach generates system diagrams based on rules that define how architectural elements should be represented and formed into diagrams. It helps to structure and generate diagrams consistently and avoid misunderstandings, which creates accurate diagrams and saves time manually generating them.

Rule-based NL processing techniques using spell checking, segmentation, tokenization, chunking, and POS tagging to extract required information from text have been used to identify the elements of a use case diagram, such as actors (identifying subjects and pronouns of a text), relationships, and use cases (identifying verbs in text) [8]. A similar approach involves using a text-to-model framework to improve productivity while creating SysML models. The process includes techniques such as pre-NLP text cleaning, structural analysis, and named-entity recognition (NER) to identify actors and their responsible actions for a set of machine-readable natural language-based policy documents. NER refers to labeling words of a text with their grammatical category, such as nouns, verbs, etc. This aids in ensuring all the sets of actions required are first captured using NLP and then translated and compiled into a proper SysML model [9]. Another approach to assist in developing system models is to use a set of heuristic rules based on the frequency of unique verbs and actions in a text to help in identifying objects, attributes, relationships, and actors for developing a use case diagram [10,11]. Chen and Zheng used the semantic representation of text through an intermediate graphic language called the recursive object model (ROMA) to generate use case and class diagrams [12]. This approach depends on the capability of the intermediate ROMA system used, which is typically used to capture the semantics of the natural language used.

Meziane et al. developed a set of rules for attributes, class, and relationship naming conventions using 45 class diagrams taken from academic textbooks for a syntactic analysis based on the frequency of how often the textbook used a specific rule in identifying classes and relationships. The association identified in most cases is reported to be composed of a single verb in the third person singular or a verb followed by a preposition. The rules developed aimed to understand and disambiguate the names given to classes, relationships, attributes, and operations in a UML class diagram [13]. Arumugam and Uma used a similar rule-based approach, where a given text input was split into sentences for tagging and marking the parts of speech of each word. The text input was simplified into constructs by using a normalizer for the ease of mapping words to object-oriented constituents [14]. Biase et al. proposed a high-level model through a semi-automatic approach containing rules that improved the initial models by creating transitions and annotating them with triggers, conditions, and actions. This enabled fragments of SysML state machine diagrams to be generated from a set of text requirements [15]. For illustrative purposes, we provide an excerpt of a rule-based technique identified by Salih and Sahraoui for generating class diagrams [16] in Table 1. We represent a set of rules, each separated by a semicolon, that helps identify components of a class diagram from a natural language text document.

2.2. Machine Learning-Based Approach

The machine learning approach trains a model to study relationships and patterns between NL text and architectural elements. It automates the process of translating a NL text of requirements that helps generate SysML diagrams, which can sometimes be a tedious task to do manually. Using machine learning helps reduce time and errors in creating diagrams. Machine learning algorithms can study the extent of meaningful understanding in categorizing and identifying patterns to generate diagrams, depending on the data analyzed. Limited articles exist on machine learning in developing automated system architectures using NLP.

Narawita and Vidanage developed a web-based UML generator that extracted use cases and identified actors and attributes using a combination of NLP preprocessing techniques and Extensible Markup Language (XML) rules to generate class diagrams. Once identified, a Weka module helped rate the use cases and extract associations [7]. The Weka module is a popular Java-based machine learning library that provides users with access to visualization tools and algorithms for data analysis and modeling [17]. Kochbati et al. proposed a machine learning model in which pre-processed text was transformed into numerical vectors to compute semantic similarity among the words in a text input and identify clusters to generate use case models [18].

Table 1. A rule-based technique to generate class diagrams.

Components of a Class Diagram	Classification Rules
Classes	NN + NNP + VBP; NN + NNP; NNP + NN + VBZ; NN + VBZ, base form + NN; NNP + NNS, NNP + NN, NNS, NNP + NNS; NNP, NNP + VBZ past tense
Methods	NN + NN + NNP; NNP + NNP + NN + NN; non-3rd person singular present VBP + NN; third person singular present VBZ + NN + NN; non-third person singular present VBP + CC + NN + any words; IN + JJ + NN
Attributes	JJ + NN
Relationships	VBP + NN; VBP + NNP; VBP + VBG

Note: singular noun (NN), plural noun (NNP), verb (VBP), plural noun (NNS), cardinal number (CC), preposition (IN), adjective (JJ), verb third person singular present (VBZ), verb present participle (VBG).

Chami et al. proposed a text-to-model framework that labeled the uploaded raw text data to identify the actors, use cases, blocks, and associations. Approximately 100 sentences were labeled and fed into an open-source library for advanced NLP to train and customize a machine learning model. This framework was applied to identify actors textually, use cases, and associations from a text input [19]. Qie et al. proposed a deep learning technique that first extracted semantic relationships from input texts to identify relationships such as composition, aggregation, and generalization for developing a block definition diagram. The semantic analysis involved entity recognition and entity relation extraction that gathered domain-specific words and used word embedding to implement a deep convolutional neural network (CNN). Once the relationships were identified, an API in Rhapsody was used to create models dynamically [1]. The proposed techniques highlight the potential of ML models to streamline and aid the automatic translation of natural language text to system models and primarily, use case diagrams.

2.3. Hybrid Approach

The hybrid approach combines rule-based and machine learning approaches to help generate UML diagrams from textual requirements and model training. It can be used in parts to process textual data and locate patterns to create diagrams, combining the strengths of manual and automated approaches. Generating diagrams with this approach can benefit the overall design quality; it can accommodate using both automatic and manual methods to resolve complex problems that are impractical to solve with one method. Hybrid approaches use NL processing techniques such as tokenizing, POS tagging, sentence splitting, word chunking, and other tasks to extract text information, eventually identifying components such as elements, actors, and relationships to make up SysML diagrams. Machine learning algorithms are trained to identify relationships between the processing data and categorize applicable components of SysML diagrams. The combination of both rule-based and machine learning approaches allows for a more robust and accurate system.

Narawita and Vidanage saved time and increased the precision in generating use case and class diagrams using machine learning and NLP techniques. A text input of requirements was first analyzed using an NLP module for tokenizing and POS tagging to classify potential actors and classes using POS tag value nouns. An XML rule removed unwanted words from a list of nouns, followed by word chunking to find verbs, nouns to represent actions, and two consecutive nouns where the second noun was a number to define attributes in a use case diagram. A Weka machine learning model evaluated and rated the actions and relationships to determine their validity [7]. Riesener and Dölle proposed a structured model-based system engineering (MBSE) requirement table for processing unstructured text using the spaCy NLP module. The text was tokenized and POS is tagged to group noun phrases to identify subjects and objects within a sentence by co-referencing pronouns to nouns. NER identified entities added to the requirement table for training machine learning models to detect requirement properties based on context [20]. Zhong et al. proposed removing relationships and critical phrases from raw text input to identify blocks and relationships based on documents such as specifications, manuals, technical reports, and maintenance reports to generate SysML diagrams, specifically structure and requirement diagrams. The steps involved the manual selection of corpus text documents, extracting key nouns, extracting relationships, generating a list of phrases and relations, generating SysML model elements, and the manual iterative selecting of the profiles and blocks to be plotted [4,21].

2.4. Challenges

Natural language is inherently complex, and developing a set of rules to aid in the automation of generating a system architecture would require rules that address language variability among different actors of a team while generating text-based requirements.

This demands the use of contexts and synonyms when considering the development of a set of rules. However, from a domain-specific outlook, as one develops a set of rules, the number of rules to be created can increase exponentially, and maintaining consistency and scale will become increasingly difficult. We also acknowledge that rules, once defined, are rigid, meaning that with changes in technology and terms used across domains, a standard set of terms and keywords are not applicable across domains and the change to new terms and what they mean in a sentence structure needs to be continually updated with significant manual input. Another challenge could be the propagation of errors if the parts of speech that define a rule must be tagged appropriately, leading to incorrect set assumptions on constituent SyML diagram elements from text.

When using machine learning models, large amounts of data are required. Considering the nature of SysML diagrams and their context-specific models, collecting data to train the models could prove difficult. While machine learning models can analyze substantial amounts of data, one issue is that these methods demand extensive datasets to tune the model for optimal outcomes to ensure the output is interpretable in the context of SysML. Training machine learning models with large datasets to account for variability in natural language is a challenge, and the key is to ensure that a cohesive set of data, such as manually annotated diagrams, is used for training. Table 2 dissects these challenges, identifying and categorizing them according to the type of system diagram.

Table 2. Challenges identified for generating individual SysML diagrams using NLP.

Diagram Type	Associated Challenges
Use Case Diagrams	• The lack of tools for grouping English language text into different bits of meaningful information [8]; • Requirement specification issues due to continually emerging technical jargon and frequently observed inconsistencies in large requirement textual documents [22]; • The lack of NLP plugins that support complex part-of-speech relationships, such as compound nouns between use cases [23]; • The lack of a standardized format in which domain requirements are specified [7,12,24]; • The lack of the ability for the user to modify a diagram once automatically generated as an image [7,12].

Table 2. *Cont.*

Diagram Type	Associated Challenges
Activity and Sequence Diagrams	• The ambiguity of natural language could generate different versions of the activity diagram based on how a model interprets the natural language text [13]; • The difficulty of NLP tools to understand variable terminology across domains that may not accurately represent the required transition and the order of actions among different activities [10].
Class Diagram	• The ambiguity and imprecision of natural language may lead to not extracting the relevant information and relationships between concepts [25]; • Tools such as CM-Builder can generate UML class diagrams. However, they cannot identify operations for candidate classes, and despite being proficient in analyzing NL requirements, their efficiency in generating UML models from analyzed requirements is in question [25]; • Using grammatical knowledge patterns by machine learning algorithms while training could lead to assumptions like that core classes are always connected, leading to incomplete diagrams and the inability to identify the multiplicity of relationships [10,11].
Package Diagram	• A poor writing style and document structure, such as implicit headings, can lead to difficulty in automatically creating a package diagram [9]. An example includes the accuracy of text-to-model results, which was challenging to evaluate, as differences in writing style between documents could affect the accuracy [9].
Block Definition Diagram	• Lexical-level features cannot capture compound semantics as they only describe word similarities, making it difficult for a model to understand NL text, leading to a failure to capture entity relationships [1,9].
Requirement Diagram	• Factors such as writing style and domain expertise can vary the input text's quality, directly affecting the generated diagram's quality. Developing accurate diagrams requires obtaining a sufficient corpus of text documents, which can be challenging in specific domains or topics [21].
Internal Block Diagram	• Like block definition diagrams, writing style, domain expertise, and document complexity affect quality and accuracy [9,21].

3. Automated System Model Generation Framework

3.1. Motivation

Several challenges have been reported in transforming natural language descriptions into automated system models. These include the inherent complexity of natural language, the difficulty in keeping rule-based systems updated and working, the strictness of rules, and the amount of data needed to make machine learning models more effective. Based on the review, approaches have been implemented to overcome problems by utilizing tokenization, spell-checking, parsing, and named-entity recognition to refine the accuracy of rule-based methods. Additionally, machine learning techniques are implemented with semantic analysis, deep learning, and NLP methods to help with the ambiguity and variability of natural languages.

The proposed framework takes a step to aid in understanding textual complexities and further clarifies natural language while reducing ambiguity and further providing flexibility to understand linguistic contexts, a challenge observed typically in rule-based techniques when interpreting natural language. An open-source library uses pre-trained machine learning models to organize and understand the text and mitigate the need for linguistic learning from scratch. Furthermore, the information extraction and selection process integrated into the framework includes user-friendly features that allow stakeholders to interact directly with the model, making diagram generation more accessible and adaptable to their needs.

3.2. Model Generation Framework—An Approach to Generate Class/Block Definition Diagram

The proposed framework utilizes Python's programming strengths and spaCy NL libraries' [26] natural language processing features. It specializes in tasks like tokenization, part-of-speech tagging, lemmatization, dependency parsing, and the attributes included with each task to process textual descriptions from .docx files for automatic system model representation. Spyder [27] and PyCharm [28] Community Edition IDEs are used for development along with PlantUML [29], a PyCharm add-in that generates system representations, alleviating manual labor in software design documentation and providing an effective transition from textual analysis to visual modeling. Figure 1 illustrates the proposed framework.

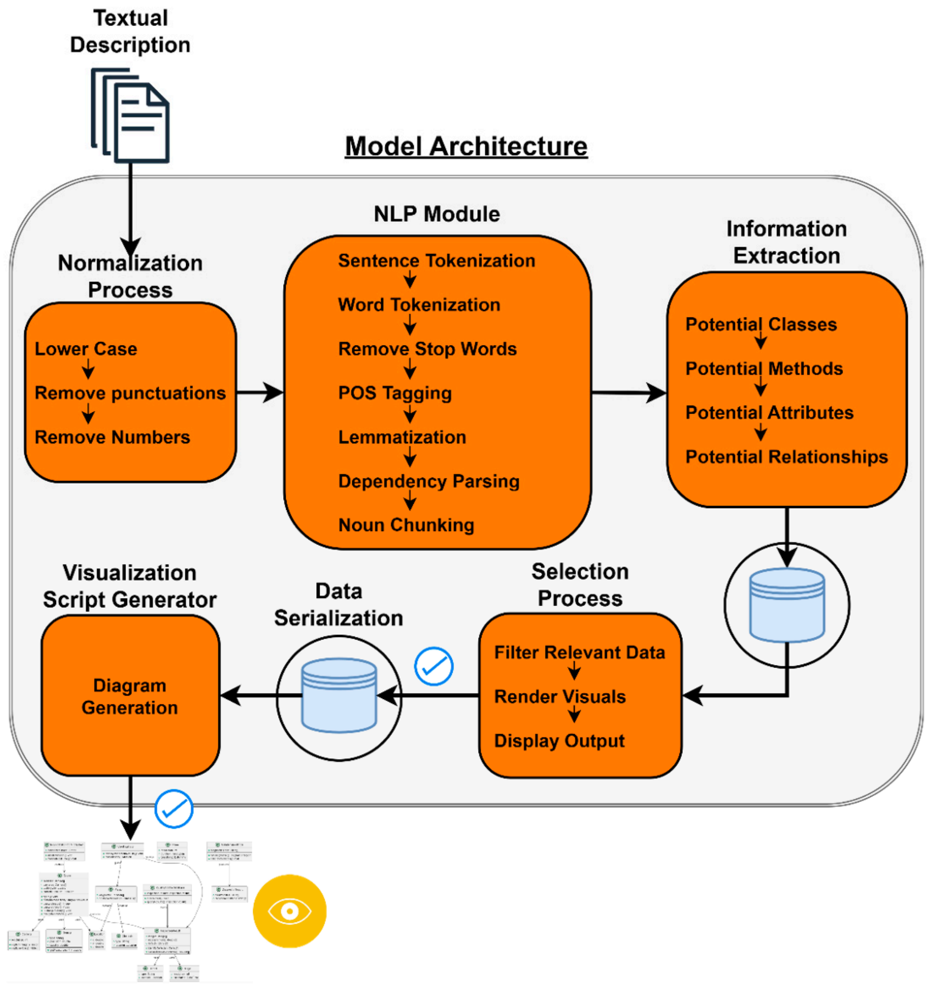


Figure 1. Proposed model generation framework.

1. *Normalization—Step 1*

The first step involves processing textual documents using the Python docx library to work with a Word document. This library can read paragraphs as structured objects from any Word document while maintaining the original format [30]. This step is crucial for normalization as it is used to clean and structure the document for analysis. The 'docx' library is imported to work specifically with Microsoft Word (.docx) files, allowing textual descriptions to be pulled from input text documents and preparing text for paragraph segmentation [31].

Once gathered, the next step involves text cleaning and preprocessing to manipulate and prepare text for further analysis. These steps include transforming all words to lowercase [32] using the regular expression library tool to remove numerical data and pattern-matching techniques [33], using the string library tool to remove punctuation symbols [34], and using the spaCy library tool to identify and remove stop words [35]. Specific numerical representation can be retained when specific numerical values are necessary to show how a system functions.

The aim is to help ensure uniformity and simplification of the text during the normalization process, establishing a solid foundation for the advanced parsing tasks that follow into the NLP module.

2. *NLP Module—Step 2*

After the text is cleaned and prepared, extracting meaningful information is next. The spaCy library was chosen over the Natural Language Toolkit (NLTK) because of its user-friendly quality. It provides an object-oriented approach rather than just serving as a tool [36]. It is well known for its speed and efficiency in processing large amounts of textual data, making it an ideal choice for the system's requirements [37]. The core of information extraction is establishing a pipeline to analyze text for tagging, parsing, lemmatization, and named-entity recognition. These pipelines can be tailored by preference regarding the language, capability features, the type of text it is trained on, and package size [38]. The 'en_core_web_sm' trained model is a small, English, web–text pipeline that uses basic NLP features, used for its optimal balance between the speed execution and accuracy the NLP tasks have to offer [26,38]. These tasks, such as part-of-speech tagging and identifying dependencies from text, are instrumental in identifying potential classes, methods, attributes, and relationships within a text input.

The first step is to examine the text and split it into sentences to determine the sentence boundaries and define word structuring. Each sentence then undergoes word tokenization, where every word is broken down into its building blocks [15]. These two steps are crucial in creating a linguistic and semantic structure for both sentences and word levels. The next step involves analyzing each token using several attributes, namely its part of speech (POS), its role in sentence structure (DEP), its base form of the word (LEMMA), and the token itself within each sentence. This process involves iterating over each token to capture and record these essential details and successfully storing each token in a structured format, making organizing each token's information possible. This analysis provides a clear view of the input text's linguistic structure, aiding in extracting elements for system model representation.

(2a) *Sentence Tokenization*

Sentence tokenization allows for examining each sentence as a distinct unit, which is essential for the further analysis of input text [15]. Iterating over each sentence ensures a more focused and accurate analysis in identifying specific elements and relationships that assist in creating accurate models [39].

(2b) *Word Tokenization*

Sentence tokenization breaks down the text into fundamental building blocks known as tokens [36]. Word tokenization involves segmenting the text into words, punctuation, numbers, etc., found within sentences throughout an input text document. This breakdown is necessary for understanding the grouping of tokens to recognize relationships, a crucial step in extracting meaningful elements necessary to create architectural diagrams. Organizing the text through tokenization lays the foundation for further analysis, paving the way toward insightful model generation.

(2c) *Part-of-Speech (POS) Tagging*

Speech tagging is necessary to identify the grammatical role of each token within the sentence and determine whether it's a noun, verb, adjective, etc. [40]. These POS tag descriptions distinguish lexical and grammatical properties of words [38]. The syntactic structure of these tokens plays a crucial role in determining what words can be considered for potential class names, actions, or methods and identifying relationships and attributes. By distinguishing the syntactic structure, POS tagging can precisely map textual descriptions to determine which tokens can be assigned to the models.

(2d) *Lemmatization*

Lemmatizing transforms words based on their tagged token to their root or dictionary form based on their POS tag [15]. This step normalizes words such as changing "installed" to "install" or "engines" to "engine", ensuring consistency and uniformity in the terminology used across diagram elements. It is performed by iterating over tokens to access each token's base form [41]. It helps refine the extraction process, permitting a more accurate depiction of classes and methods.

(2e) *Dependency Parsing*

Dependency parsing builds a categorized tree of relationships between tokens, illustrating their grammatical structure and clarifying how words interact [41]. This tree analogy helps determine and visualize connections like trailing family ties, which helps identify relationships between entities within the text. It analyzes which tokens function as modifiers or are associated with others to confirm potential attributes or relationships in architectural diagrams. It helps distinguish syntactic roles, including subjects, objects, and modifiers, essential for mapping characteristics and relationships within system diagrams [41].

(2f) *Noun Chunking*

Noun chunking splits sentences into noun phrases tagged with nouns from the POS tagger, typically comprising a head noun and its modifiers [41]. It highlights critical entities and their descriptions, helping extract information and providing a deeper understanding of the text's structure.

3. *Information Extraction—Step 3*

After the text is cleaned and processed, the next step is extracting information. A dependency visualizer is a feature within the spaCy library that connects word relationships within a sentence, known as a dependency tree [26]. It visualizes how tokens are related, using arrows to connect child tokens to their parents, illustrating a sentence's grammatical structure. This visualization helps understand how words work together, which can be used to define attributes and identify relationships between potential classes in the system. Figure 2 illustrates the grammatical structure of sentences, highlighting how tokens relate to each other, which assists in defining attributes and identifying relationships between potential classes. Each word in the sentence is tagged with its respective part of speech. For example, in the phrase "the conveyor belt", "the" is a determiner for "conveyor", and "conveyor" is connected to "belt" as a compound relationship, indicating that two nouns combine to form a single noun entity.

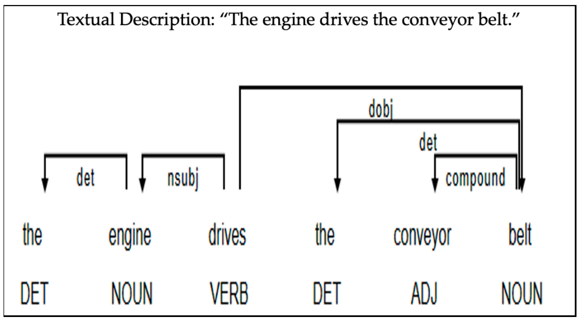


Figure 2. Illustration of dependency visualizer use.

(3a) *Extracting Potential Classes*

The process identifies potential classes by inspecting singular, multiword, and chunking nouns, capturing the combination of simple and complex entities within each sentence [42]. The rules below illustrate how class elements can be extracted from textual descriptions and an example description. We refer to the term "rule" as a predefined criterion that is used to identify specific pieces of information from text.

- ***Class rule 1***: If the POS tag contains [NOUNS] or [PROPN—Proper Noun], singular nouns are extracted for potential classes, capturing the lemmatized noun. These singular nouns form the base of potential class identification.
- ***Class rule 2***: Multiword nouns adjacent to each other, which represent class names such as [NOUN] [NOUN] or [NOUN] [PROPN], are extracted.
- ***Class rule 3***: Noun chunks that contain [NOUNS] and their modifiers are extracted to capture a class description. These chunks include a combination of [NOUNS] with their modifiers, which can be [VERBS], [ADJ—Adjective], or both their combinations. Noun chunking helps capture the full description of class attributes from simple to complex forms.
- ***Class rule 4***: If the DEP tag, i.e., the syntactic dependency, the relation between tokens contains [NSUBJ—Nominal Subject] and [NSUBJPASS—Nominal Subject Passive], their tokens are extracted for potential classes, capturing the lemmatized token. If modifiers such as [COMPOUNDS] or [ADJ] precede the subjects, the subject and modifier are extracted to create a descriptive entity. These subjects are potential entities since they are assigned to the syntactic heads of nouns or pronouns.

The singular nouns, multiword nouns, and chunking nouns are then added to a list of potential classes, identifying the entities to be considered for the system.

(3b) *Extracting Potential Methods*

Potential methods are identified by analyzing verbs in a text within sentences [40]. The process begins by examining each sentence's structure, looking for verbs that denote actions with nouns or proper nouns as potential objects of these actions. Before forming verb–noun pairs, a set list is created for each sentence to capture unique pairings and avoid word overlapping. This step helps examine the interactions within each sentence to find elements suitable for potential methods. It also filters out any digits and stop words and takes each token's lemmatized form in each sentence. The rules below illustrate how potential methods are identified and extracted, along with a description and example.

- ***Method rule 1***: Begin by creating a new set at the start of each sentence to capture verb–noun pairs to ensure the sentence is analyzed independently. This step helps prevent duplications in potential method identification and pairs actions with their objects.
- ***Method rule 2***: Singular [VERBS] are identified after iterating over each sentence token and added to the potential method set considering their lemmatized word. The [VERB] is also stored for a possible pairing with the following noun in the next step.
- ***Method rule 3***: Verb–noun pairs are extracted by identifying a [NOUN] or [PROPN] that is found after a [VERB] from the pairing list to capture a potential method by

combining an action and their object. This step takes both lemmatized words for the pair.

- ***Method rule 4***: The pairs are then added to a list of potential methods, identifying the actions and their respective objects within the system.
- ***Method rule 5***: To provide deeper insight, this analysis can be extended to capture verb–noun triples that comprise a [VERB], its direct [NOUN], and an additional [NOUN]. This expands the context of understanding an action and extracts additional elements for a system being designed.

(3c) *Extracting Potential Attributes*

Potential attributes are identified by words describing a class entity's characteristics or properties [43]. These attributes are found by locating nouns modified by adjectives or, less commonly, by a compound relationship. This step analyzes child tokens to identify class characteristics by examining the child tokens and the adjectives, modifying them to present an accurate representation of elements in a diagram. Using child tokens in the analysis helps identify a thorough range of attributes, providing enhanced details of class characteristics. The rules below illustrate how potential attributes are identified and extracted, along with a description and an example.

- ***Attribute rule 1***: POS tagging is used to identify [NOUNS] after iterating over each token in a sentence. These nouns are used for the next step to help capture a representative attribute.
- ***Attribute rule 2***: For each [NOUN] captured, children tokens connected to it labeled [ADJ] or [compound] are identified with it. These modifiers provide a descriptive context for the noun.
- ***Attribute rule 3***: The modifiers collected with the noun create a complete attribute phrase. This phrase captures a more sophisticated description of the attribute.
- ***Attribute rule 4***: These attributes are added to a set to ensure each is unique and not duplicated, creating a list of potential attributes.

(3d) *Extracting Potential Relationships*

Relationships are identified by connecting entities associated with one another [40]. The extraction of potential relationships identifies patterns where verbs are paired with nouns, suggesting an association among entities. The process focuses on each verb and examines dependent children to see if it matches with any nouns or proper nouns that are possible subjects related to potential classes. Relationships are recorded between the verb in its lemmatized form and the child token's text when a match is found. This method can be further altered by identifying verb–noun pairs between classes, where the two nouns bridging the pair are considered potential classes from the first extraction step. This method can identify any relationship between classes stated in the text document, illustrating the system's structure. The rules below illustrate how potential relationships are identified and extracted, along with a description and an example.

- ***Relationships rule 1***: The text is scanned for [VERBS] using the POS tagger, identifying actions within sentences for relationship identification.
- ***Relationships rule 2***: For each [VERB] found, its child tokens are examined to see if it is associated with potential subjects. These subjects are considered among previously identified potential classes, such as [NOUNs] or [PROPN].
- ***Relationships rule 3***: The [VERB] is matched with any potential subjects to see if it acts upon any entity recognized as a class from the previous analysis.
- ***Relationships rule 4***: A relationship entry is created if a match is found, indicating that the subject is recognized as a potential class. This entry pairs the [VERB] in its lemma form with the subject's text, capturing the action–entity relationship within the relationship set.
- ***Relationships rule 5***: Relationships can be extracted by examining verb–direct object connections, capturing another action–entity within the text.

- ***Relationships rule 6***: For each [VERB] found, its [dobj—direct object] is identified among the verb's child tokens. If the direct object is found to the verb, they are both paired in their lemma form, creating a relationship entry.
- ***Relationships rule 7***: Another relationship can be formed by identifying a [VERB] and a DEP tag [PREP—Preposition] that follows it. If the [PREP] is present, both [VERB] and [PREP] are paired, creating an association within the text.

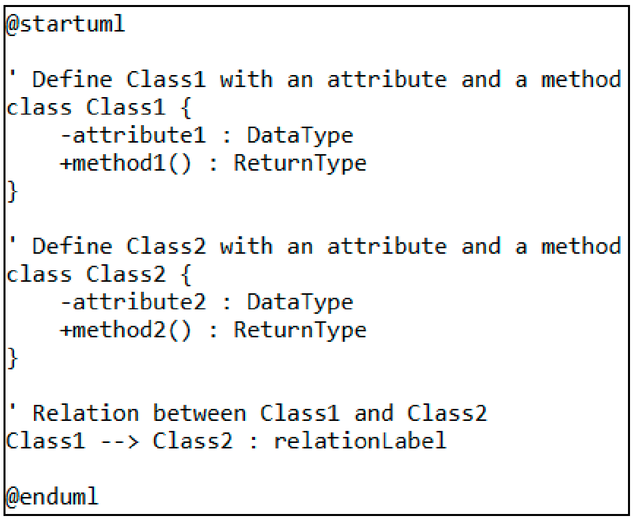
4. *Data Selection and Serialization Process*

After extraction, all potential elements are documented in a JSON file and are now pending user selection. These files are used due to their simplicity and handling of large amounts of data to help verify the extraction of essential elements [44]. This format offers the advantage of re-executing the extraction method with modified rules if any elements were missed or not correctly captured, ensuring the data serialization remains current without storing any previous markers.

An interactive interface allows users to finalize the appropriate classes, methods, and attributes. This process prompts the user to select the proposed elements from the file to form classes with their corresponding characteristics and operations. After creating each class, the process further engages users to refine and specify relationships. This involves selecting the relationships found in the text and combining them with the classes chosen for the system. Once all necessary elements are assigned, they are saved in a JSON file. This file is then utilized to generate PlantUML syntax, successfully transforming the extracted data into a structured representation of a system. PlantUML is a tool that allows users to create diagrams from textual descriptions, particularly UML diagrams.

5. *Visualization Script Generator—Step 4*

The final step involves visualization, where the extracted data generate a system model. It begins with the selection process phase, where potential elements, such as classes, corresponding methods, attributes, and relationships, are chosen. The elements are translated and stored into the PlantUML syntax required to generate system models. Its syntax is user-friendly and straightforward, making creating, sharing, and modifying multiple diagrams easy. PlantUML can also handle and represent JSON formats, as it can turn them into visual diagrams, which helps in the understanding and visualization of data structures. It can be integrated through various IDEs (integrated development environments), such as Spyder or Pycharm, to help modify and create diagrams within the coding environment, making it an easy, productive process [45]. An example of PlantUML syntax is shown in Figure 3, showcasing how a hypothetical Class 1 is connected to Class 2, with both classes having methods and attributes.

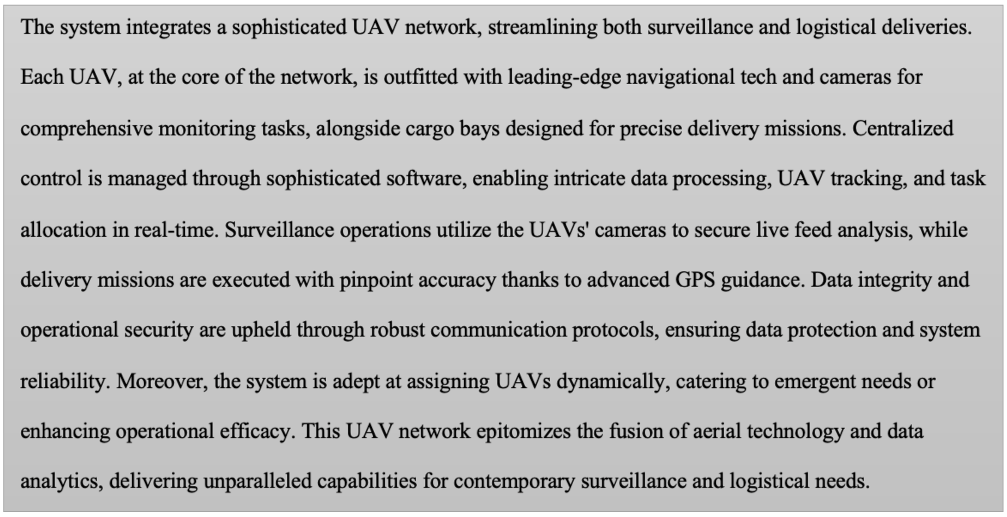


```
@startuml

' Define Class1 with an attribute and a method
class Class1 {
    -attribute1 : DataType
    +method1() : ReturnType
}

' Define Class2 with an attribute and a method
class Class2 {
    -attribute2 : DataType
    +method2() : ReturnType
}

' Relation between Class1 and Class2
Class1 --> Class2 : relationLabel

@enduml
```

Figure 3. Representation of PlantUML syntax.

4. Framework Implementation and Case Studies

We generated a hypothetical requirement specification for a surveillance scenario involving unmanned aerial vehicles (UAVs) to demonstrate the framework's applicability. Potential classes, attributes, methods, and relationships are extracted for the text illustrated in Figure 4. The elements are saved to a JSON file for the next step, enabling the user to select possible aspects for the selection process to generate a class diagram. The user is prompted first to identify an entity in the selection process. The process goes in order of creating a class, its methods, and its attributes, and then the user is asked if another class creation is necessary.



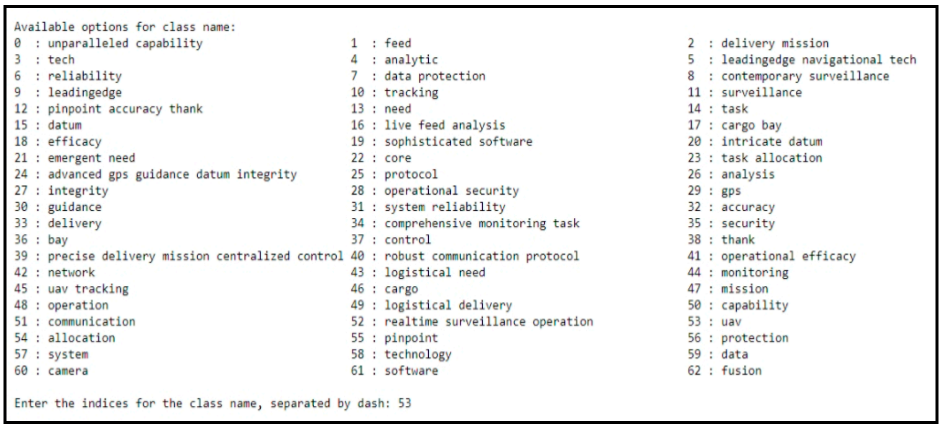
The system integrates a sophisticated UAV network, streamlining both surveillance and logistical deliveries. Each UAV, at the core of the network, is outfitted with leading-edge navigational tech and cameras for comprehensive monitoring tasks, alongside cargo bays designed for precise delivery missions. Centralized control is managed through sophisticated software, enabling intricate data processing, UAV tracking, and task allocation in real-time. Surveillance operations utilize the UAVs' cameras to secure live feed analysis, while delivery missions are executed with pinpoint accuracy thanks to advanced GPS guidance. Data integrity and operational security are upheld through robust communication protocols, ensuring data protection and system reliability. Moreover, the system is adept at assigning UAVs dynamically, catering to emergent needs or enhancing operational efficacy. This UAV network epitomizes the fusion of aerial technology and data analytics, delivering unparalleled capabilities for contemporary surveillance and logistical needs.

Figure 4. A set of requirements for a hypothetical UAV surveillance scenario.

A list of nouns, noun chunks, and multiple nouns tagged from the textual requirements are generated to be classified as potential classes. Next, a list of words is tagged by verbs that denote nouns to characterize an object's action. After choosing a class, the user can pick relevant methods, separated by dashes, to be incorporated. For instance, methods that pertain to the entity UAV based on the textual description can be that it executes a 'pinpoint destination' or 'utilizes a camera'. Next, nouns are tagged in the list of words and their modifiers using children tokens to identify class characteristics representing properties that pertain to entities. After selecting the methods for a particular class, the user can pick as many attributes as appropriate, separated by dashes, to incorporate them into a specific class. After the entity's methods and attributes are chosen, the user can select and associate potential relationships with the previously chosen classes. These lists of relationships are tagged by verbs paired with nouns or direct objects using the dependent children to suggest possible subjects being associated between entities.

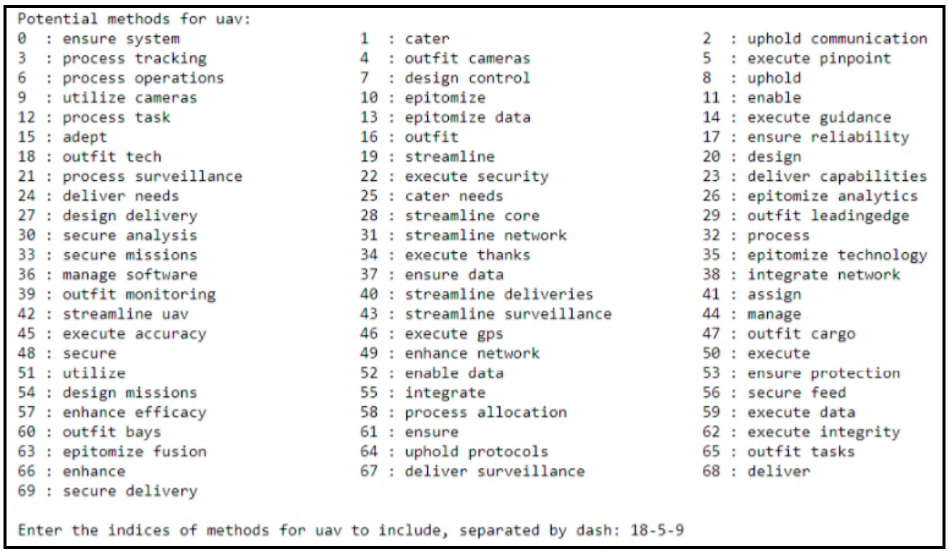
Finally, the selections are saved into a JSON file as an object-oriented class representation. This file saves the selection data for the visualization script generator to generate the selections of PlantUML syntax for model generation. Figures 5 and 6 illustrate the instances of the user interface prompting the user to choose appropriate classes, methods, attributes, and their relationships for the text illustrated in Figure 4.

Once the class representation data is saved, the visualization script generator converts it into appropriate PlantUML syntax and pastes its output for visualizing the class diagram. Figure 7 illustrates the class diagram representation of the UAV requirement specification. Each class shows the named entity located on the top, the methods toward the bottom of the class, and the attributes in between them. The lines with an arrowhead going from one class to another represent the relationship between the two entities, labeled with an association description specifying their connection's nature. This approach in generating a class diagram not only focuses on word-to-word matches but also employs an adaptable approach to account for semantic similarities and recognize phrases with similar meanings from text input.



```
Available options for class name:
0  : unparalleled capability                      1  : feed                                          2  : delivery mission
3  : tech                                         4  : analytic                                      5  : leadingedge navigational tech
6  : reliability                                  7  : data protection                               8  : contemporary surveillance
9  : leadingedge                                  10 : tracking                                      11 : surveillance
12 : pinpoint accuracy thank                      13 : need                                          14 : task
15 : datum                                        16 : live feed analysis                            17 : cargo bay
18 : efficacy                                     19 : sophisticated software                        20 : intricate datum
21 : emergent need                                22 : core                                          23 : task allocation
24 : advanced gps guidance datum integrity        25 : protocol                                      26 : analysis
27 : integrity                                    28 : operational security                          29 : gps
30 : guidance                                     31 : system reliability                            32 : accuracy
33 : delivery                                     34 : comprehensive monitoring task                 35 : security
36 : bay                                          37 : control                                       38 : thank
39 : precise delivery mission centralized control 40 : robust communication protocol                 41 : operational efficacy
42 : network                                      43 : logistical need                               44 : monitoring
45 : uav tracking                                 46 : cargo                                         47 : mission
48 : operation                                    49 : logistical delivery                           50 : capability
51 : communication                                52 : realtime surveillance operation               53 : uav
54 : allocation                                   55 : pinpoint                                      56 : protection
57 : system                                       58 : technology                                    59 : data
60 : camera                                       61 : software                                      62 : fusion

Enter the indices for the class name, separated by dash: 53
```

Figure 5. An instance of a user interface prompt to enable the selection of appropriate classes.



```
Potential methods for uav:
0  : ensure system                    1  : cater                            2  : uphold communication
3  : process tracking                 4  : outfit cameras                   5  : execute pinpoint
6  : process operations               7  : design control                   8  : uphold
9  : utilize cameras                  10 : epitomize                        11 : enable
12 : process task                     13 : epitomize data                   14 : execute guidance
15 : adept                            16 : outfit                           17 : ensure reliability
18 : outfit tech                      19 : streamline                       20 : design
21 : process surveillance             22 : execute security                 23 : deliver capabilities
24 : deliver needs                    25 : cater needs                      26 : epitomize analytics
27 : design delivery                  28 : streamline core                  29 : outfit leadingedge
30 : secure analysis                  31 : streamline network               32 : process
33 : secure missions                  34 : execute thanks                   35 : epitomize technology
36 : manage software                  37 : ensure data                      38 : integrate network
39 : outfit monitoring                40 : streamline deliveries            41 : assign
42 : streamline uav                   43 : streamline surveillance          44 : manage
45 : execute accuracy                 46 : execute gps                      47 : outfit cargo
48 : secure                           49 : enhance network                  50 : execute
51 : utilize                          52 : enable data                      53 : ensure protection
54 : design missions                  55 : integrate                        56 : secure feed
57 : enhance efficacy                 58 : process allocation               59 : execute data
60 : outfit bays                      61 : ensure                           62 : execute integrity
63 : epitomize fusion                 64 : uphold protocols                 65 : outfit tasks
66 : enhance                          67 : deliver surveillance             68 : deliver
69 : secure delivery

Enter the indices of methods for uav to include, separated by dash: 18-5-9
```

Figure 6. An instance of a user interface prompt to enable the selection of appropriate methods.

The effectiveness of the extraction framework in identifying the appropriate elements to create a class diagram and identifying essential elements of a class diagram, such as classes, methods, attributes, and relationships, from an input text is evaluated using precision and recall measures.

Precision and recall are commonly used metrics for measuring a machine learning model's performance. They are vital measures to help understand how well a system identifies and selects appropriate elements [44–47]. Precision helps measure the proportion of positively identified elements predicted, whereas recall helps measure the proportion of actual elements identified.

- *Precision = (Number of correct entities detected)/(Total number of correct and incorrect entities detected)*
- *Recall= (Number of correct entities detected)/(Total numbers of correct entities)*

To illustrate this, Table 3 shows two sample sentences segmented from the UAV requirements text with the actual and extracted classes, methods, attributes, and relationships with the precision and recall calculated. In this case, precision is calculated by counting the number of correctly predicted classes, methods, attributes, and relationships divided by the total number of entities in each class predicted.

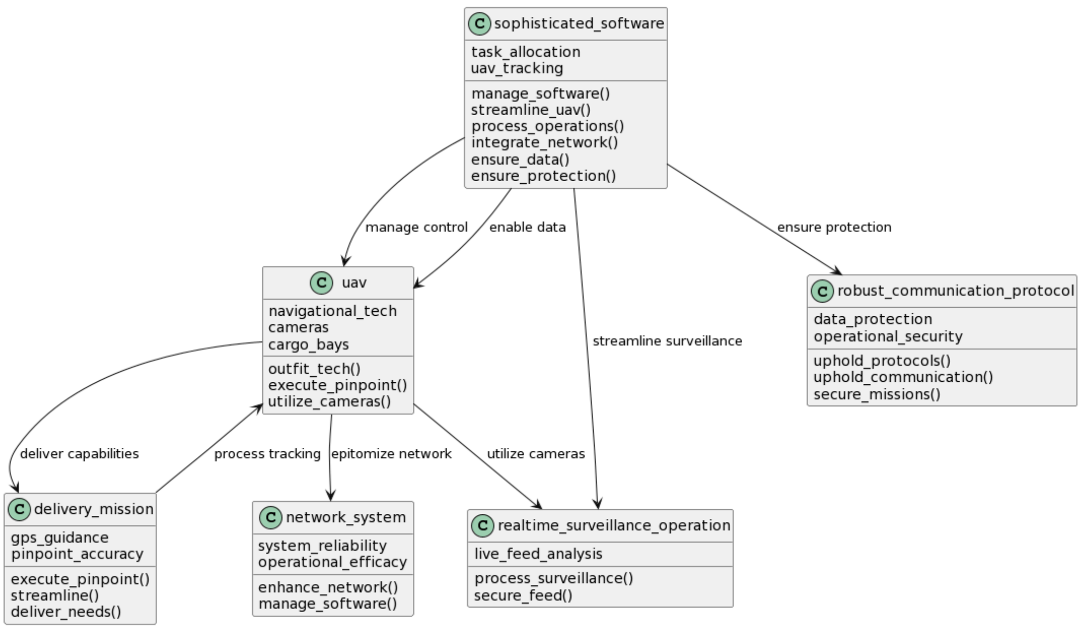


Figure 7. Generated class diagram using the framework for the UAV requirement specification.

Table 3. Precision and recall metrics for sample segmented sentences.

Segmented Sentences from Requirements			
Sentence # 1: The system integrates a sophisticated UAV network, streamlining both surveillance and logistical deliveries. Sentence # 2: Each UAV, at the core of the network, is outfitted with leading-edge navigational tech and cameras for comprehensive monitoring tasks, alongside cargo bays designed for precise delivery missions.			
Classes	**Extracted Classes**	**Precision**	**Recall**
System; UAV; network; surveillance; logistical delivery	'delivery', 'network', 'system', 'sophisticated uav network', 'surveillance', 'logistical delivery'	0.66	1.0
UAV; network; navigational tech; camera; monitoring task; cargo bay; delivery mission	'delivery mission', 'tech', 'uav', 'delivery', 'comprehensive monitoring task', 'leading edge navigational tech', 'core', 'monitoring task', 'leadingedge', 'task', 'network', 'bay', 'monitoring', 'cargo', 'mission', 'camera', 'precise delivery mission', 'cargo bay'	0.38	1.0
Methods	**Extracted Methods**	**Precision**	**Recall**
Integrates network; surveillance; deliveries	'streamline', 'streamline surveillance', 'integrate network', 'integrate', 'streamline deliveries'	0.6	1.0
Tech; cameras; monitoring; task; delivery	'design', 'outfit', 'design delivery', 'design missions', 'outfit cameras', 'outfit tech', 'outfit cargo', 'outfit monitoring', 'outfit bays', 'outfit leadingedge', 'outfit tasks'	0.45	1.0
Attributes	**Extracted Attributes**	**Precision**	**Recall**
UAV network; logistical delivery; surveillance	'logistical deliveries', 'system', 'sophisticated uav network', 'surveillance'	0.75	1.0
Navigational tech; monitoring task; precise delivery; cargo bays; cameras	'monitoring tasks', 'uav', 'delivery', 'core', 'network', 'monitoring', 'cargo', 'navigational tech', 'cargo bays', 'precise delivery missions', 'cameras'	0.45	1.0
Relationships	**Extracted Relationships**	**Precision**	**Recall**
System integrates; integrates network; integrates; streamline	'streamline surveillance', 'streamlining', 'integrates', 'integrate system', 'integrate network'	0.8	1.0
outfitted with; designed for; outfitted	'outfitted with', 'outfit uav', 'design for', 'outfitted'	0.75	1.0

Recall is measured by counting the number of correctly predicted classes, methods, attributes, and relationships, each divided by the number of entities that must be extracted. The observed high recall value across the sentences indicates the framework's effectiveness in predicting the required elements of classes, attributes, methods, and relationships. Relatively low precision values, when compared to the recall, indicate the framework's ability to capture a broader list of potential elements. One must consider the tradeoff between precision and recall, where a high precision value reduces recall and vice versa. In Table 3, low precision values indicate that the framework extracts entities more than required from the input text, increasing false positives, and high recall values suggest that it predicted elements accurately.

Case Studies

The proposed framework is tested against two case studies which consider different sizes of input text in terms of word count and writing style. Precision and recall measures are calculated for each text input [47].

Text 1: "A university consists of a number of departments. Each department offers several courses. A number of modules make up each course. Students enroll in a particular course and take modules towards the completion of that course. Each module is taught by a lecturer from the appropriate department, and each lecturer tutors a group of students." [47].

Text 2: "Participants at the summer school are either students or teachers. Each student registers for the NEMO Summer School providing, amongst others, their level of study (Bachelor, Master or PhD) and their field of study. Additionally each student provides her/his first name, last name, their country of provenience and e-mail address. Students attend courses during the summer school. Courses can be a lecture, a fundamentals exercise or application exercises. [The fundamental exercise is considered as one unit as it covers one topic, although it takes place in several sessions.] Each course has a title, is being given by one or more lecturers and takes places in a room. Every room has a name, a seating capacity, and technical equipment. Lectures and application exercises take place in a lecture hall, while fundamental exercises are conducted in PC-labs. Within the fundamentals exercise students are split in groups. Each group has a group number, a room (i.e., PC-lab) and a tutor. Teachers can be either lecturers or tutors. Each teacher has a first name, last name, host institution, and country." [47].

Similar to the previous example, potential classes, attributes, methods, and relationships are extracted from each input text. Figures 8 and 9 illustrate the class diagrams generated.

The metric results shown in Table 4 indicate that for Text 1, the framework identified all correct elements with a 100% recall for classes and relationships compared to the previous work [47]. However, precision is observed to be relatively low, indicating a broader capture of potential classes, attributes, and relationships.

Table 4. Precision and recall measures for selected text inputs.

	Text 1		Text 2	
Extracted Elements	**Precision**	**Recall**	**Precision**	**Recall**
Classes/Entities	0.5	1.0	0.17	1.0
Attributes	0.57	1.0	0.45	0.96
Relationships	0.58	1.0	0.18	0.93

For Text 2, the framework correctly identified more elements than the previous work [48] but still captured a broad list of potential elements as the system's precision values are low.

The rules for extracting potential elements ensure that no possible elements are overlooked, showing that a broad list can inspire more flexible design decisions. Low precision serves as a strategy by providing a more extensive set of elements that cater to various user needs and scenarios when developing a system model.

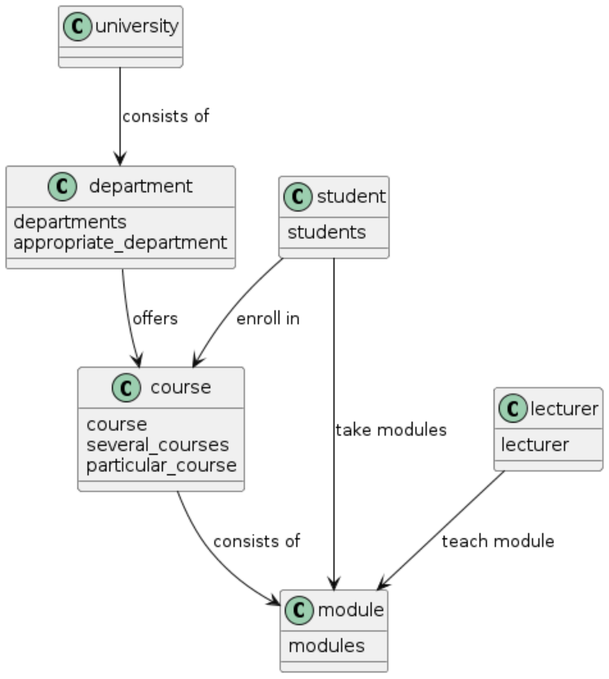


Figure 8. Class diagram generated for Text 1.

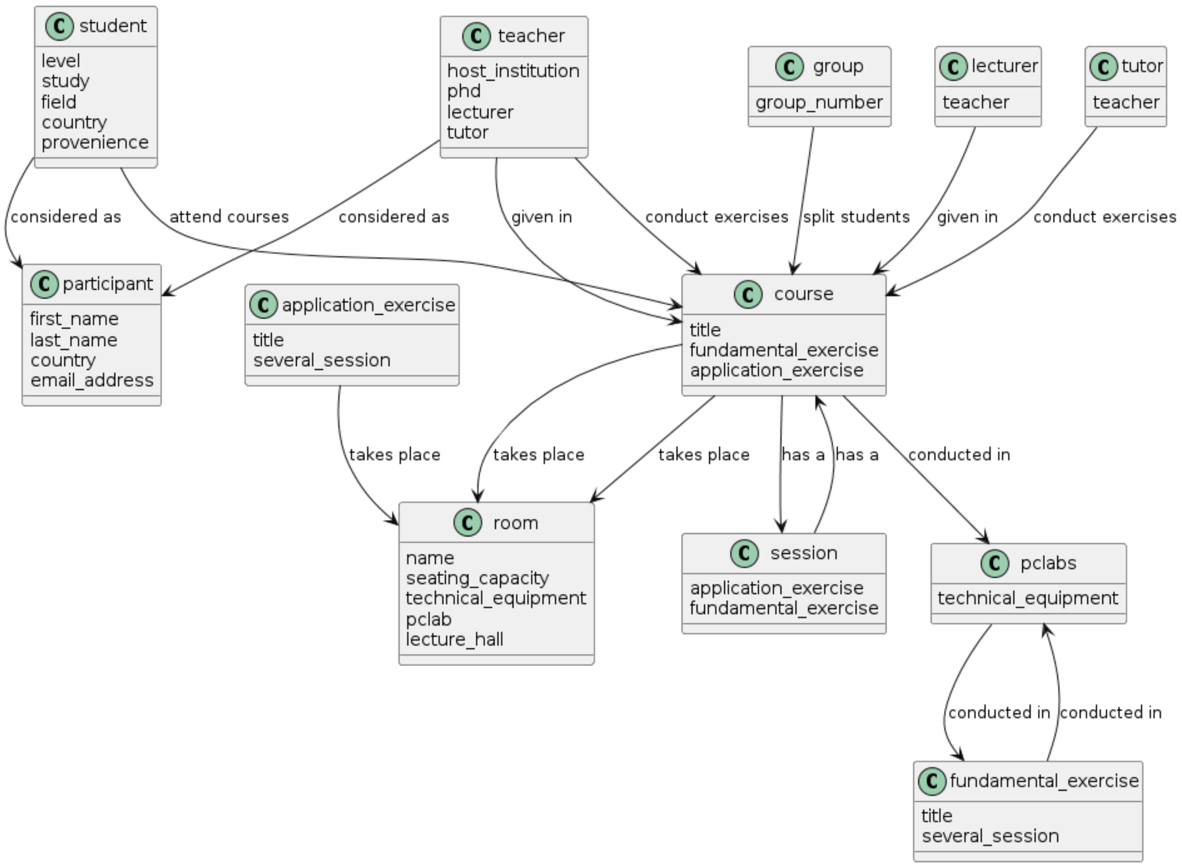


Figure 9. Class diagram generated for Text 2.

5. Discussion

Previous attempts to automatically generate system architectures from natural language text, such as requirements, can be categorized into rule-based, machine learning-based, or hybrid methods. Even though these three approaches are used differently in steps and phases to develop models across the literature, the overall importance of text preprocessing, which is essential in transforming NL text into structured data, is prominent. This paper proposes a rule-based architectural generation framework that uses heuristic rules and predefined patterns to help map natural language text to a SysML element. The rule-based approach requires manual effort to define and update the rules. In contrast, the machine learning-based approach refers to statistical techniques that help glean and learn from data to map natural language text to SysML elements. It requires training data sets based on which the model depends on identifying elements for a SysML diagram from a natural language text. It implies a need for extensive structured data sets to train the models to understand the subtle differences of natural language written by various stakeholders. The roadblocks observed can be categorized into the following:

- *The inflexibility of rules in a rule-based system:* A uniform set of rules developed for a context in a rule-based approach must accurately consider the language variability among different stakeholders while generating natural language-based text. Machine learning tools and techniques can aid in creating and adjusting rules and recognizing contexts in natural language text to address this inflexibility as data are augmented over time.
- *The lack of ability to manually change a diagram once generated by an algorithm:* This refers to the need to develop an interactive ML system where one can provide manual input during the training phase to improve accuracy when generating SysML diagrams from natural text. The proposed framework takes a step in this direction by enabling an interactive user interface for users to create a streamlined model.
- *The lack of ML models to better interpret natural language nuances:* This refers to the lack of advanced ML models that can capture contexts and compound semantics from natural language text in the context of SysML. This stresses the importance of deep learning models such as transformers [48] designed to understand context more effectively, leading to the development of famous large language models like GPTs (generative pretrained transformers). A similar model could be developed for tasks such as SysML diagram generation.

6. Scalability

Many systems design tasks for complex systems contain requirements that can number in the thousands. As the volume of the text input increases, so does the complexity, thus requiring the integration of deep learning models in the proposed framework, which requires significant computational resources. Furthermore, large complex systems have interdependencies that can be challenging to parse and analyze accurately. Future research is needed to scale the framework and accommodate modular components such as text preprocessing, entity recognition, and relationship extraction. Integrating more advanced NLP and machine learning algorithms, such as transformer-based models, could effectively transform text into models. Testing its applicability with incrementally large data sets is suggested to validate the framework further.

7. Conclusions

The contribution of this paper to the systems modeling domain is two-fold. First, a review and analysis of natural language processing techniques for the automated generation of SysML diagrams are provided, broadly categorized into three approaches: rule-based, machine learning-based, and hybrid. Challenges for generating SysML diagram types using natural language processing techniques are also mapped. Second, a framework to extract textual relationships tailored for generating a class diagram/block diagram that contains the classes/blocks, their relationships, methods, and attributes is presented. The process involves utilizing NLP techniques to analyze and interpret text descriptions of a system and then transforming that information into a graphical representation. This transformation does not create new knowledge but changes the form of existing information. The applicability of this approach is presented in two case studies from the previous literature and is validated against precision and recall measures. The framework's performance largely depends on how well the NLP model captures a diverse dataset. Tailoring the model for hardware- or software-dominant systems would involve refining the rules with feedback from domain experts to capture domain-specific terminologies. Finally, to mitigate the bias generated by the rules, future work would include testing the framework on various texts generated by varying levels of domain experts so that the framework can handle valuable natural language descriptions.

Author Contributions: Conceptualization, A.A. and J.O.; methodology, A.A. and J.O.; validation, A.A., J.O. and S.L.; writing—original draft preparation, A.A. and J.O.; writing—review and editing, A.A., J.O. and S.L.; visualization, A.A. and J.O. All authors have read and agreed to the published version of the manuscript.

Funding: The authors would like to acknowledge the funding and support provided by the National Science Foundation (NSF) Award No. 2235999 and the CREST Center for Multidisciplinary Research Excellence in Cyber-Physical Infrastructure Systems (MECIS) under NSF Award No. 2112650. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing does not apply to this article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Qie, Y.; Zhu, W.; Liu, A.; Zhang, Y.; Wang, J.; Li, T.; Li, Y.; Ge, Y.; Wang, Y. A Deep Learning Based Framework for Textual Requirement Analysis and Model Generation. In Proceedings of the 2018 IEEE CSAA Guidance, Navigation and Control Conference (GNCC), Xiamen, China, 10–12 August 2018; pp. 1–6.
2. Theobald, M.; Tatibouet, J. Using fUML Combined with a DSML: An Implementation using Papyrus UML/SysML Modeler. In Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, Prague, Czech Republic, 20–22 February 2019; pp. 248–255.
3. Zhao, L.; Alhoshan, W.; Ferrari, A.; Letsholo, K.J.; Ajagbe, M.A.; Chioasca, E.-V.; Batista-Navarro, R.T. Natural language processing for requirements engineering: A systematic mapping study. *ACM Comput. Surv. (CSUR)* **2021**, *54*, 1–41. [CrossRef]
4. Zhong, S.; Scarinci, A.; Cicirello, A. Natural Language Processing for systems engineering: Automatic generation of Systems Modelling Language diagrams. *Knowl.-Based Syst.* **2023**, *259*, 110071. [CrossRef]
5. Ahmed, S.; Ahmed, A.; Eisty, N.U. Automatic Transformation of Natural to Unified Modeling Language: A Systematic Review. In Proceedings of the 2022 IEEE/ACIS 20th International Conference on Software Engineering Research, Management and Applications (SERA), Las Vegas, NV, USA, 25–27 May 2022; pp. 112–119.
6. Petrotta, M.; Peterson, T. Implementing Augmented Intelligence In Systems Engineering. *INCOSE Int. Symp.* **2019**, *29*, 543. [CrossRef]
7. Narawita, C.R.; Vidanage, K. UML generator—Use case and class diagram generation from text requirements. *Int. J. Adv. ICT Emerg. Reg. (ICTer)* **2017**, *10*, 1–10. [CrossRef]
8. Hamza, Z.A.; Hammad, M. Generating UML Use Case Models from Software Requirements Using Natural Language Processing. In Proceedings of the 2019 8th International Conference on Modeling Simulation and Applied Optimization (ICMSAO), Manama, Bahrain, 15–17 April 2019; pp. 1–6.
9. Chen, M.; Bhada, S.V. Converting natural language policy article into MBSE model. *INCOSE Int. Symp.* **2022**, *32*, 73–81. [CrossRef]
10. Shinde, S.K.; Bhojane, V.; Mahajan, P. NLP based Object Oriented Analysis and Design from Requirement Specification. *Int. J. Comput. Appl.* **2012**, *47*, 30–34. [CrossRef]
11. Abdelnabi, E.A.; Maatuk, A.M.; Abdelaziz, T.M.; Elakeili, S.M. Generating UML Class Diagram using NLP Techniques and Heuristic Rules. In Proceedings of the 2020 20th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA), Monastir, Tunisia, 20–22 December 2020; pp. 277–282.
12. Chen, L.; Zeng, Y. Automatic Generation of UML Diagrams From Product Requirements Described by Natural Language. In Proceedings of the ASME 2009 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, San Diego, CA, USA, 30 August 2009. [CrossRef]

13. Meziane, F.; Athanasakis, N.; Ananiadou, S. Generating natural language specifications from UML class diagrams. *Requir. Eng.* **2008**, *13*, 1–18. [CrossRef]
14. Anandha Mala, G.S.; Uma, G.V. Automatic Construction of Object Oriented Design Models [UML Diagrams] from Natural Language Requirements Specification. In *PRICAI 2006: Trends in Artificial Intelligence, Proceedings of the 9th Pacific Rim International Conference on Artificial Intelligence, Guilin, China, 7–11 August 2006*; Springer: Berlin/Heidelberg, Germany, 2006; Volume 4099, pp. 1155–1159. [CrossRef]
15. de Biase, M.S.; Marrone, S.; Palladino, A. Towards Automatic Model Completion: From Requirements to SysML State Machines. In Proceedings of the 18th European Dependable Computing Conference (EDCC 2022), Zaragoza, Spain, 12–15 September 2022. [CrossRef]
16. Dawood, O.S. Toward requirements and design traceability using natural language processing. *Eur. J. Eng. Technol. Res.* **2018**, *3*, 42–49.
17. Frank, E.; Hall, M.A.; Witten, I.H. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*, 4th ed.; Morgan Kaufmann: Burlington, MA, USA, 2016.
18. Kochbati, T.; Li, S.; Gérard, S.; Mraidha, C. From User Stories to Models: A Machine Learning Empowered Automation. In Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development, Online Streaming, 8–10 February 2021. [CrossRef]
19. Chami, M.; Zoghbi, C.; Bruel, J.M. A First Step towards AI for MBSE: Generating a Part of SysML Models from Text Using AI. In Proceedings of the AI4SE 2019: INCOSE Artificial Intelligence for Systems Engineering, Madrid, Spain, 12–13 November 2019.
20. Riesener, M.; Dölle, C.; Becker, A.; Gorbatcheva, S.; Rebentisch, E.; Schuh, G. Application of natural language processing for systematic requirement management in model-based systems engineering. In Proceedings of the INCOSE International Symposium, Virtual Event, 17–22 July 2021; Volume 31, pp. 806–815. [CrossRef]
21. Buchmann, R.; Eder, J.; Fill, H.G.; Frank, U.; Karagiannis, D.; Laurenzi, E.; Mylopoulos, J.; Plexousakis, D.; Santos, M.Y. Large language models: Expectations for semantics-driven systems engineering. *Data Knowl. Eng.* **2024**, *152*, 102324. [CrossRef]
22. Seresht, S.M.; Ormandjieva, O. Automated assistance for use cases elicitation from user requirements text. In Proceedings of the 11th Workshop on Requirements Engineering (WER 2008), Barcelona, Spain, 12–13 September 2008; Volume 16, pp. 128–139.
23. Elallaoui, M.; Nafil, K.; Touahni, R. Automatic Transformation of User Stories into UML Use Case Diagrams using NLP Techniques. *Procedia Comput. Sci.* **2018**, *130*, 42–49. [CrossRef]
24. Osman, M.S.; Alabwaini, N.Z.; Jaber, T.B.; Alrawashdeh, T. Generate use case from the requirements written in a natural language using machine learning. In Proceedings of the 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), Amman, Jordan, 9–11 April 2019; pp. 748–751.
25. Joshi, S.D.; Deshpande, D. Textual Requirement Analysis for UML Diagram Extraction by using NLP. *Int. J. Comput. Appl.* **2012**, *50*, 42–46. [CrossRef]
26. Fantechi, A.; Gnesi, S.; Livi, S.; Semini, L. A spaCy-based tool for extracting variability from NL re-quirements. In Proceedings of the 25th ACM International Systems and Software Product Line Conference—Volume B, Leicester, UK, 6–11 September 2021; pp. 32–35.
27. Spyder IDE Contributors. Spyder (Version 5.4.1) [Software]. 2023. Available online: https://www.spyder-ide.org/ (accessed on 1 August 2024).
28. JetBrain. PyCharm 2023.2.1 (Community Edition) [Software]. Build #PC-232.9559.58, Built on 22 August 2023. Available online: https://www.jetbrains.com/pycharm/ (accessed on 1 August 2024).
29. PlantUML Integration. PlantUML Integration (Version 7.0.0-IJ2023.2) for PyCharm [Software Plugin]. 2023. Available online: https://plugins.jetbrains.com/plugin/7017-plantuml-integration (accessed on 1 August 2024).
30. Claghorn, R.; Shubayli, H. Requirement Patterns in the Construction Industry. In Proceedings of the INCOSE International Symposium, Virtual Event, 17–22 July 2021; Volume 31, pp. 391–408. [CrossRef]
31. Kulkarni, A.; Shivananda, A. *Natural Language Processing Recipes*; Apress: Berkeley, CA, USA, 2019.
32. Octavially, R.P.; Priyadi, Y.; Widowati, S. Extraction of Activity Diagrams Based on Steps Performed in Use Case Description Using Text Mining (Case Study: SRS Myoffice Application). In Proceedings of the 2022 2nd International Conference on Electronic and Electrical Engineering and Intelligent System (ICE3IS), Yogyakarta, Indonesia, 4–5 November 2022; pp. 225–230.
33. Mande, R.; Yelavarti, K.C.; JayaLakshmi, G. Regular Expression Rule-Based Algorithm for Multiple Documents Key Information Extraction. In Proceedings of the 2018 International Conference on Smart Systems and Inventive Technology (ICSSIT), Tirunelveli, India, 13–14 December 2018; pp. 262–265.
34. Ismukanova, A.N.; Lavrov, D.N.; Keldybekova, L.M.; Mukumova, M.Z. Using the python library when classifying scientific texts. In *European Research: Innovation in Science, Education and Technology*; 2018; pp. 9–13. Available online: https://internationalconference.ru/images/PDF/2018/46/using-the-python-1.pdf (accessed on 1 August 2024).
35. Srinivasa-Desikan, B. *Natural Language Processing and Computational Linguistics: A Practical Guide to Text Analysis with Python, GenSim, SpaCy, and Keras*; Packt Publishing Ltd.: Birmingham, UK, 2018.
36. Jugran, S.; Kumar, A.; Tyagi, B.S.; Anand, V. Extractive automatic text summarization using SpaCy in Python & NLP. In Proceedings of the 2021 International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE), Greater Noida, India, 4–5 March 2021; pp. 582–585.
37. Uysal, A.K.; Gunal, S. The impact of preprocessing on text classification. *Inf. Process. Manag.* **2014**, *50*, 104–112. [CrossRef]

38. Explosion AI. SpaCy: Industrial-Strength Natural Language Processing in Python. 4 April 2024. Available online: https://spacy.io (accessed on 1 August 2024).
39. Vasiliev, Y. *Natural Language Processing with Python and SpaCy: A practical Introduction*; No Starch Press: San Francisco, CA, USA, 2020.
40. Bashir, N.; Bilal, M.; Liaqat, M.; Marjani, M.; Malik, N.; Ali, M. Modeling class diagram using NLP in object-oriented designing. In Proceedings of the 2021 National Computing Colleges Conference (NCCC), Taif, Saudi Arabia, 27–28 March 2021; pp. 1–6.
41. Shuttleworth, D.; Padilla, J. From Narratives to Conceptual Models via Natural Language Processing. In Proceedings of the 2022 Winter Simulation Conference (WSC), Singapore, 11–14 December 2022; pp. 2222–2233. [CrossRef]
42. Herchi, H.; Abdessalem, W.B. From user requirements to UML class diagram. *arXiv* **2012**, arXiv:1211.0713.
43. Almazroi, A.A.; Abualigah, L.; Alqarni, M.A.; Houssein, E.H.; AlHamad, A.Q.M.; Elaziz, M.A. Class diagram generation from text requirements: An application of natural language processing. In *Deep Learning Approaches for Spoken and Natural Language Processing*; Springer: Cham, Switzerland, 2021; pp. 55–79.
44. Arachchi, K.D. AI Based UML Diagrams Generator. Ph.D. Thesis, University of Colombo School of Computing, Colombo, Srilanka, 19 August 2022.
45. PlantUML. Available online: https://plantuml.com/ (accessed on 1 August 2024).
46. Bozyiğit, F. Object Oriented Analysis and Source Code Validation Using Natural Language Processing. Ph.D. Thesis, Dokuz Eylül University, Izmir, Turkey, 2019.
47. Baginski, J. Text analytics for conceptual modelling. Master's Thesis, University of Vienna, Vienna, Austria, 2018.
48. Islam, S.; Elmekki, H.; Elsebai, A.; Bentahar, J.; Drawel, N.; Rjoub, G.; Pedrycz, W. A Comprehensive Survey on Applications of Transformers for Deep Learning Tasks. *Expert Syst. Appl.* **2024**, *241*, 122666. [CrossRef]