Universiteit Leiden

ICT in Business and the Public Sector

From Natural Language to UML Class Models: An Automated Solution Using NLP to Assist Requirements Analysis

Name: Tiantian Tang
Student-no: s2236516

Date: 01/09/2020

1st supervisor: Dr. G.J. Ramackers
2nd supervisor: Prof. dr. S.A. Raaijmakers

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

1st supervisor: Dr. G.J. Ramackers
2nd supervisor: Prof. dr. S.A. Raaijmakers

Acknowledgements

Foremost, I would like to express my deepest appreciation to my first supervisor, Guus Ramackers, for his continuous support and guidance on my thesis. This thesis would never have been completed without his patient and enthusiastic guidance. His guidance helped me in all the time of research and writing of this thesis. Also, I am thankful to my second supervisor Stephan Raaijmakers. He gave me valuable suggestions and feedback to improve my thesis and point out the direction of future work.

I am deeply indebted to my parents and friends. They have provided me with so much understanding and laughter during the most difficult and stressful moments of this research work.

At such a special time, I would like to acknowledge Wiebe Posthuma, Leiden Student Councilor Department and Leiden Institute of Advanced Computer Science (LIACS) for their efforts to help me. Also, I would like to extend my sincere thanks to Dai Clegg and Fahmeena Odetta Moore, who responded to my questionnaire and provided valuable comments on the tool developed in this research.

Abstract

The initial input to the requirements analysis process of many software development projects typically consists of natural language documents produced by business experts and end-users. When dealing with complex software systems, these documents are then often converted into structured Unified Modeling Language (UML) models by UML experts and consultants using software development tools. This process of manually converting natural language into UML models is a time-consuming process. Furthermore, once models are expressed in UML, they become harder to interpret by non-experts, leading to increased complexity when requirements have not been expressed in a complete or consistent manner, and when subsequent changes to requirements occur.

To address some these issues, this research utilizes Natural Language Processing (NLP) toolkits to automate a significant part of the conversion of natural language requirements documents into UML Class metadata. A web-based application to demonstrate the feasibility of the approach has been developed in Python, with back-end processing based on the NLTK and Stanford CoreNLP libraries. When compared to existing tools in this area of research, our application does not restrict the input documents to follow a rigid, pre-defined structure. Furthermore, it requires minimal user intervention, and covers the full range of structural concepts that are part of the definition of UML Class models.

In addition, this research implementation interfaces with other research performed at Leiden University that provides a runnable prototype from the generated UML meta data. This combined facility provides immediate feedback to the user by showing them the (initial) application that results from their textual requirements input. This enables users to identify errors early on in the process, and iterate towards the intended result in a rapid manner.

Table of Content

List of figures

List of tables

1. Introduction

1.1. Background and Problem Statement

The System Development Life Cycle (SDLC) in software development consists of various methods such as waterfall, iterative, rapid application development (RAD), agile, SCRUM, etc. All of these contain requirements analysis during the early phases. Reducing errors in the requirements analysis phase is crucial, because such errors escalate exponentially in the later stages of a system development process (Boehm & Basili, 2001; Westland, 2002; Gupta & Deraman, 2019).

Software requirements are commonly expressed by business experts and end users in natural language (NL) during discussions and textual communication. Subsequently, they will be analyzed and documented, typically by analysis experts and consultants. The Unified Modelling Language (UML) is widely utilized in transforming textual or oral requirements to system design, particularly in terms of its use case models, activity models, and class models (Agarwal & Sinha, 2003).

UML class models provide a structural view of a system in terms of the central information (concepts) of the problem domain, and their interrelationships. As such, these models define the “universe of discourse” which is dynamically modified by the dynamic internal and external interactions of the system (Fowler, 2003; Narawita & Vidanage, 2016).

The process of defining a precise UML class model based on the verbal and textual requirements is currently a manual one, requiring the involvement of UML experts. An automated, or semi-automated solution for generating UML models is highly desirable (Narawita & Vidanage, 2016), see figure 1. As part of their research, the authors show that 75% of survey respondents thought an automated solution for UML model generation is important.

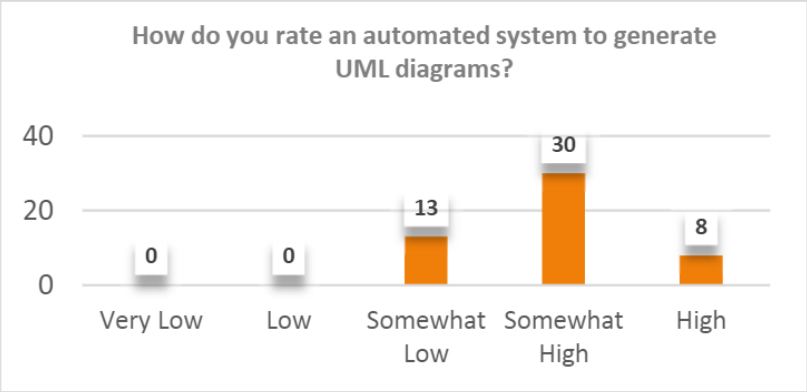


Figure 1. The Importance of An Automated Solution (Narawita & Vidanage, 2016)

The current process of generating UML models from requirements has many drawbacks. Firstly, reading and analyzing a large volume of requirements by human actors (UML experts and consultants) is time-consuming. Secondly, although there are a large number of commercial and open-source tools that help with drawing UML class diagrams, the majority of them utilize a “drag and drop” style user interface. In such a conventional context, not only analyzing the input, but subsequently defining the UML model adds an additional layer of complexity. The defining process also requires multiple rounds of communication and discussion between requirements analysts and stakeholders (e.g. end users and business experts) to enable iterations of the UML models. Therefore, in order to improve these inconveniences, automated support for the requirement analysis process is highly desirable.

In addition, a critical challenge in the requirement analysis process from NL requirements to system design is coping with changes, particularly in situation such as a system with correspondingly complex stakeholder communities (Berzins et al, 2008). For example, business experts may suggest additional requirements after the UML class models has been completed, or end users decide that a part of a UML model from their previous requirement is not what they intended. During this process, changes occur frequently and need to be responded to in a rapid way, otherwise they can propagate into later phases of software development, potentially causing errors or even failure (Kof, 2005). Therefore, the automated support for generating UML class models

during the requirement analysis process should enable the inclusion of all stakeholders in such a manner that enables obtaining rapid feedback on the results. This enables validation of the models produced, and incorporating any required changes in an iterative process.

In summary, there are two requirements for the solution provided in resolving the above issues:

- Automate the analysis process from textual requirements to UML models;
- Involve more stakeholders in the requirements analysis process through the use of the tool.

The details of these requirements are elaborated more specifically in the next section (chapter 1.2).

Natural Language Processing (NLP), a subset of Artificial Intelligence focuses on analyzing unstructured text (Arellano et al., 2015). It has been employed to automate UML class model generation in previous research (Yalla & Sharma, 2015; Osman & Zălhan, 2016). However, the tools implemented were designed to aid requirement analysis experts only. Other stakeholders such as end users and business specialists are not enabled by such tools. Secondly, the vast majority of automated tools developed are not “open source”, and therefore are not open to researchers and developers, and as a result, are no longer being improved. Finally, an important problem with existing solutions in the area of NLP driven requirements analysis is that these tools still *“require consistent human intervention in the process of UML diagrams generation”* (Osman & Zălhan, 2016).

Our research provides a tool that can be accessed by users at different levels, and is available for further research. Reducing human intervention in the requirements analysis process is a central objective of our research, and tool implementation. The requirements analysis process is also a highly challenging task due to the ambiguous and semantic problems that are inherently present in natural language (Narawita & Vidanage, 2016; Deshpande, 2012). As a result, it is unrealistic to obtain completely error-free analysis results. This research thus turns to the purpose of achieving rapid

generation of preliminary UML model results when given textual software requirements.

1.2. Research Objectives

In the context of the problem statement above, we define our research objectives as follows:

Develop a flexible, open-source UML class model generator to automate the requirements analysis software development phase by utilizing NLP techniques.

The generator is a web-based application that is developed by using the open-source Python Web framework “Django”. Several NLP toolkits and techniques are used to develop the UML class model generator. A more detailed requirements based on previous section addressed in this automated solution are as follows:

- Automate the analysis process from textual requirements to UML models
  - Minimize the tool interruption by UML experts;
  - Enable to use fairly natural expressions of software requirements from business specialists and end users. The tool does not force a particular structure on the input text.

- Involve more stakeholders in the requirements analysis process through the use of the tool
  - Easy to use for any level of user, whether end-user, business specialist, software developer, or UML consultant;
  - Enable rapid testing of requirements by generating a prototype implementation (this part of the research utilizes the run-time application framework – developed by Ralph Driessen (2020)), providing a rapid feedback to users about the implications of their requirements in terms of executable prototype software.

The focus of this research is to utilize existing research in the field of NLP for the requirements analysis process and to implement a UML Class model generation tool. As such, it employs the research methodology of “design science”. This paradigm is focused on solving research problems by building and evaluating artifacts (Hevner et al., 2004). The artifacts in this research include the tool implementation and the architectural frameworks that implement it. The focus of this study is on the extraction of metadata, the primary source of composition for UML class models. There are also five sample requirements (Appendix A) as input for this design tool to gain corresponding UML class metadata result. A questionnaire survey (Appendix B) is conducted to evaluate the design tool and extracted results.

1.3. Thesis Overview

This research is comprised of seven parts. In the introduction (Chapter 1), the importance of an automated solution for the requirement analysis process and issues that existed in previous research are presented. The second chapter elaborates on the concepts used in UML class modelling and its metadata, describes the context of Natural Language Processing (NLP), and the usage of NLP in requirements analysis. The third chapter illustrates related research in this domain (i.e. previous NLP-based tools in generating UML class models), their development frameworks and limitations. The fourth chapter describes the architecture design and framework of our application. The fifth chapter explains the output results of several sample requirement data sets, and discusses the questionnaire result on the evaluation of our tool. The sixth chapter contains the conclusion, discusses current limitations of this research, and proposes further research directions.

2. Theoretical Background

This chapter presents the concepts that having essential relevance to this research. Firstly, the Unified Modelling Language (UML) class model and its elements are illustrated, followed by the metadata definition and implementation in the UML class model. Secondly, Natural Language Processing (NLP) is introduced including its application in the requirements analysis process of software engineering.

2.1. Unified Modelling Language Class Model

Unified Modelling Language (UML), developed by Object Management Group (OMG) in 1997, is an open standard software modelling language embraced by the industry and is exhibited by graphical notation to depict a complex system (Fowler, 2003; Miles & Hamilton, 2006). There are various representations of UML diagrams such as use case diagrams, activity diagrams, sequence diagrams, etc. The UML class diagram is the core type of UML diagrams that are used in real life when defining a system design (Fowler, 2003). Therefore, this research centres around the UML class diagrams to explore its components and generation. The basic elements of the UML class diagram are introduced as follows.

The UML class diagram is a static modelling technique that defines the objects and their relationships in a system. An object normally consists of a class, properties (attributes), and operations of that class (Fowler, 2003). The following picture (Figure 2) displays how a class is interpreted in the UML class diagram.

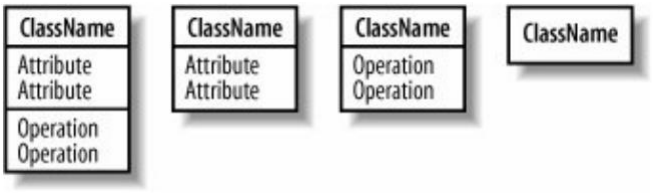


Figure 2. Class in UML notation (Miles & Hamilton, 2006)

As can be seen, the rectangular shape has three sections. The top section is the name of the class. A class has the characteristics of abstraction and encapsulation (Miles & Hamilton, 2006). The middle section contains attributes of that class. Normally an attribute has names and types. For example, a person has a name and age with string and integer type respectively. Thus, an example of a name attribute can be written as *"name: String"*. Operations are the behaviours or declaration of methods carried by a class (Fowler, 2003). Imagine a logistic system that a customer can view his or her order, which can be expressed in the customer class as *"ViewOrderDetails()"*. Figure 3 gives a concrete example of a UML class diagram containing its elements and notation expressions.

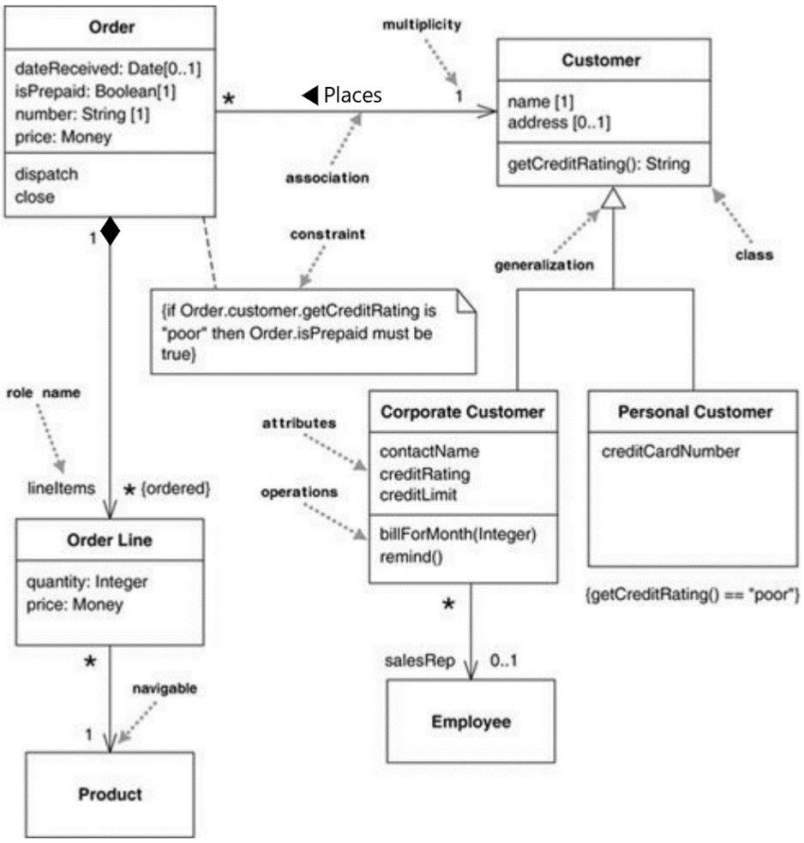


Figure 3. An Example of UML Class Diagram (Fowler, 2003)

There are different types of relationships in the UML class model. Figure 4 displays six types of relationships in the UML class model. For instance, the association is expressed as a solid line with a name directed from one class to

another target class in a diagram. *"A customer can place many orders"*, which can be depicted by a solid line between customer and order with the word "places", together with the notation of multiplicity. The composition is a stronger version of a relationship type. For instance, a house has rooms and the rooms do not exist separate from the house. Inheritance and subtyping are commonly understood similarly. However, these two should be separated. Subtyping indicates the compatibility of interfaces, where the attributes and operations in a class can be invoked to its subtyped class. While inheritance focuses on the reuse of implementations. For example, some operations for a class B are written in terms of the operation of class A, this is so-called inheritance (i.e., class B inherits from class A).

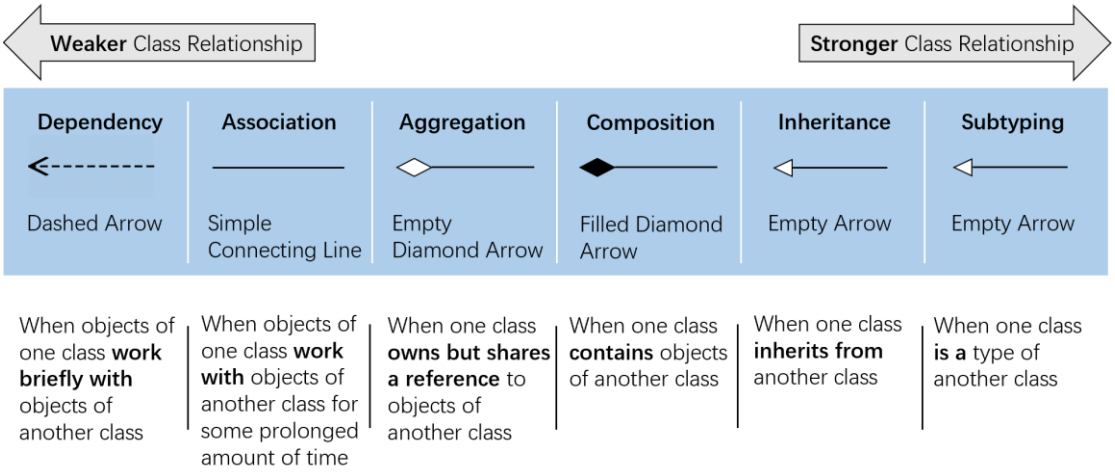


Figure 4. UML Classes Relationship (Adapt from Miles & Hamilton, 2006)

Multiplicity in the UML class model can be set for relationships between classes, and for attributes and operations of classes. It indicates the allowable number of objects (instances of classes) participating in a given relationship. It can be a specific number or a range. There are four indicators of multiplicity in table 1.

Indicator	Meaning	Example
1 (one to one)	Exactly one object	Each student carries exactly one ID card.

0..* or * (zero to many)	No object or at least one or more	A person might own no phone, one phone or many phones.
1..* (one to many)	At least one object or many	A building can contain many doors.
0..1 (zero to one)	No instance or have one instance	Car can be assigned to a road. A road can also be free of car.

Table 1. Multiplicities in UML class model (Adapt from Ambler, 2004)

2.1.1. Metadata in Unified Modelling Language Class Model

"Metadata" is a term that simply represents *"data about data"*. It both describes the organizational level of their structure and use of information, and the technical level of the description of a system which is used to manage that information (Hay, 2006). The business information collected from requirement analysis is typically the initial component of metadata captured in any development project (Hay, 2006). Those metadata can be refined and used as the data about a data model, where the model can be a database design, or in our research of following UML class standards to define classes and their relations. Table 2 gives an example to show the metadata in a UML class model for a bank:

Metadata	Data about a data model (UML class model)	Class: *"Customer"* Attributes: *"Name"* *"Phone number"*	Class: *"Employee"* Attributes: *"Position"* *"Name"* *"Phone number"* Relationships: *"An asset manager is responsible for*

			one customer only”
Instance data	Data about real-world things	Customer Name: *“Bob Lee”* Customer phone number: *“XX XXXXXXX”*	Manager Name: *“Sam Smith”* Manager phone number: *“XX XXXXXX”*
	Real-world things	A particular customer: *“Bob Lee”*	An asset manager: *“Sam Smith”*

Table 2. Data and metadata in UML class model

Since metadata should address both business and technical point of view (Hay, 2006), an objective of our application is to display the extracted UML class metadata in a well-understood and structural way for both of them.

Due to the complexity and ambiguity of natural language, it is difficult to perform the extraction of complete elements of a UML Class model. Our application abandons the identification of Operations, and Relationship types such as Dependency, Aggregation and Inheritance, as they are similar to Association, Composition and Subtyping (which are stronger Relationship types) in the requirements expressed in natural language and are difficult to distinguish directly. For example, *“A folder contains many files”* and *“A car has wheels”*. For the former sentence, if a folder is deleted, all the files it contains are deleted. It indicates a Composition Relationship between “Folder” and “File”. The latter sentence can be understood as a car needs wheels, but wheels do not need a car. Wheels can also apply to bicycles and motor vehicles. This suggests a weaker type of Relationship, i.e. Aggregation between “Car” and “Wheel”. The boundary is not explicit in natural language expressions.

The application does, however, identify Classes and Attributes, Association, Composition and Subtyping of Relationship types. The extraction of

Multiplicity is the most challenging task and this application develops a relatively simple extraction method for it.

2.2. Natural Language Processing

NLP is a way of analysing, understanding, gathering knowledge from a natural language with different forms (text or oral) by computerized means (Arellano et al., 2015; Joseph et al., 2016). Its origins go back to the beginning of the 1940s. Developments in artificial intelligence have boosted the field significantly, and NLP currently achieves sufficiently high-quality linguistic analysis to accomplish different tasks and serve for a range of applications (Arellano et al., 2015; Joseph et al., 2016; Jones, 2001).

As a multidisciplinary research area dealing with linguistics, NLP can bring benefits to Software Engineering. The NLP techniques can be employed in every phase in Software Development Life Cycle (SDLC) (Dawood & Sahraoui, 2017; Yalla & Sharma, 2015), particularly in requirement-related stages since requirements are normally written and expressed by natural language. Koerner et al (2014) summarized NLP can be applied in improving requirement specifications, finding domain-specific ontologies in requirements engineering, helping automatic model creation and text synthesis, and extracting the impact assessment from changes in software specification.

This research focuses on one of the above applications in requirement analysis: using NLP techniques to improve automatic model creation from NL requirements. The first step in automating the model creation is extracting information (i.e., UML class metadata) from the textual requirements. NLP techniques utilized in this area belong to the application of Information Extraction (IE) (Narawita & Vidanage, 2016). IE as an application of NLP, *"refers to the automatic extraction of structured information such as entities, relationships between entities, and attributes describing entities from unstructured sources."* (Sarawagi, 2007) In this research, extracting classes, attributes, and relationship information between classes addresses the tasks in IE application. Examples of NLP techniques for an IE system are as follows:

- Segmentation

  The segmentation tasks divide the text into tokens, typically following the rules such as using the whitespaces separate words and full stop separates sentences in English, or based on statistics such as using N-grams for segmentation in Chinese (Simões et al., 2009).
- Named Entity Extraction

  One of a classic technique in IE application includes Named Entity Recognition (NER). It identifies and classifies the specific types of named entities such as the name of people, places, organizations in domain-independent area or the name of disease, drug in domain-specific area (Sarawagi, 2007; Adnan & Akbar, 2019).
- Relation Extraction (RE)

  Relation Extraction is a task of extracting relationships between entities. Various techniques are applied in relation extraction, most of which are based on rules. A general approach is based on syntactic analysis, since the relations we want to extract are often in grammatical form. For example, a verb may refer to a relationship between entities, e.g., “is acquired by” relationship between pairs of companies (Sarawagi, 2007; Simões et al., 2009). Open Information Extraction (Open IE) is a notable task in recent years that extracting open-domain relation triples (i.e., “arg1”, “rel”, “arg2”) from the raw text. It was initially introduced by Banko et al (2007) where the system extracted a large set of relational triples from web. In general, the Open IE application starts with collecting sentences from corpuses and splitting each sentence into sets of entailed clauses, then shortening each clause maximally to produce sentence fragments in triplets by using shallow syntax and dependency methods (Angeli et al., 2015; Ali et al., 2019). For example, the output of sentence “Born in a small town, she took the midnight train going anywhere” in Stanford OpenIE is “(she; took; midnight train)” (Angeli et al., 2015).
- Ontology Induction

  The ontology induction refers to construct an ontology and map textual expressions to concepts and relations in that ontology. An IE application of ontology induction is extracting knowledge from unstructured text. Typically, the application usually begins with specifying domain-specific

lexical knowledge, extraction rules and an ontology, then the learning approaches are applied to the processed data. (Yildiz & Miksch, 2008; Poon & Domingos).

2.2.1. Natural language processing toolkits

There are several powerful NLP toolkits developed in different programming languages. These toolkits can be used to accomplish common NLP tasks. For example, Natural Language Toolkit (NLTK) and SpaCy in Python, Stanford CoreNLP developed by Java, Apache OpenNLP that has been used in previous research, Google's SyntaxNet as a rising tool in this market.

Cheng et al (2020) found that the publicly available NLP toolkits are pervasive in analysing software requirement documents. They investigated a set of different public NLP toolkits and found that NLTK and Stanford CoreNLP (including parts of it such as Standford Parser) achieve the highest accuracy in results of doing tokenization for all kinds of software requirement documents. Stanford CoreNLP is the most frequently used NLP library in this research field (Cheng et al., 2020).

Osman and Zălhan (2016), Dawood and Sahraoui (2017) analysed previous NLP-based UML generation tool and found the NLP toolkit from StanfordCoreNLP are widely adopted to analyse textual requirements, especially Stanford Parser. Osman and Zălhan (2016) also argued that *"the majority of NLP libraries belong to NLTK framework."*. The application is developed in Python environment. NLTK is the most powerful and convenient tool to use in this case. Stanford CoreNLP is a Java tool but has API in Python environment, and it's easy to use the packages in Python. Their frameworks and implementations in this application are elaborated in Chapter 4.

2.2.2. Natural language processing and machine Learning

Natural Language Processing (NLP) and Machine Learning (ML) are both subsets of Artificial Intelligence (AI). They are complementary to each other. Through NLP, computers can handle a number of tasks in human language, such as keywords recognition, text classification, translation, etc. To automate these tasks and applications, ML is utilized as the process of applying algorithms to the system. It teaches the system to understand, learn and improve from new data inputs without the need for explicit human reprogramming.

However, in order to achieve this and produce accurate results, it is often necessary to train a significant and clean data set for the system. While this has been achieved with great success in areas such as biological and medical applications, sufficient training data sets are the first obstacle to applying this to software requirements analysis, as requirements documents and their associated UML models are difficult to obtain. These data are commonly internal to the company and are kept confidential. Self-made requirements texts introduce bias, as people have their own natural language expression habits. It is also a time-consuming task to find UML experts to compile software requirements and to label those data.

3. Related Work

Several studies have integrated NLP techniques to support UML generation, and provided their respective development methods. Those studies depended significantly on the NLP technology available at the time. As a result, the tools developed in this study also makes use of newer state-of-the-art NLP tools. In addition, none of the existing tools can extract complete information from NL requirements such as classes, their relationships, attributes, direction, etc (Osman & Zălhan, 2016). Our research aims to broaden the set of extracted UML metadata. Moreover, human intervention is largely required in existing tools from sentence input to output result (Osman & Zălhan, 2016). Our aim is to minimise human intervention in conversion process supported by our automated tooling.

Given these objectives, the remainder of this chapter explores the development methods implemented by previous researchers in their tools, with the aim of extending their methods and frameworks. This provides insights into addressing existing issues, i.e., user intervention problems, immaturity of mechanisms.

Initially, the development of these tools was solely inspired by the experience of requirements analysts working with textual requirements. Traditionally, the requirement analyst selects and highlights nouns and verbs from the entire text to identify possible objects and operations (Overmyer et al., 2001). The experience of previous experts has led to a practical and natural way of identifying objects and methods by utilizing part-of-speech (POS) of the words (Chen, 1983; Barker, 1990; Overmyer et al., 2001). For example, linking nouns to classes, verbs to relationships, adjectives or prepositional phrases to attributes.

3.1. Linguistic Assistant for Domain Analysis

A representative study comes from Overmyer et al (2001). They relied on POS-tagging to develop a prototype tool named **Linguistic Assistant for Domain Analysis (LIDA)**. The tool tags the POS of words after the analyst imports a requirements document. Then a noun marking list is generated to indicate candidate classes. The analyst iteratively removes classes that do not qualify as classes. A similar process is employed with the adjective list and verb list. The final step for analyst is using LIDA Modeler to graphically associate identified classes, attributes and methods (Overmyer et al., 2001). Obviously, LIDA relies heavily on user intervention. The analyst has to check and refine the marking lists at each step. Furthermore, some elements of the UML class model, e.g., multiplicity, are excluded. LIDA was placed as an aid for UML experts when analysing requirement texts (Overmyer et al., 2001). It did not analyse the texts themselves, as it was limited by the development of NLP technology at that time.

3.2. Graphic Object-Oriented Analysis Laboratory

Perez-Gonzalez and Kalita (2002) developed a different way to construct UML models from software requirement description, by regulating the input text. They proposed a tool called **Graphic Object-Oriented Analysis Laboratory (GOOAL)**. The analysts must declare the problem domain name, sub-domain name, problem name and problem description before they use the tool. This tool used a semi-natural language (4W) to identify syntactic subjects and objects, and prepositional phrases of relations. They were then analysed sentence-by-sentence using role posets, a conceptual framework based on the linguistic notion of theta roles and mathematical notion of ordered sets, which can be used to produce tabular and graphical results of UML class metadata (Perez-Gonzalez & Kalita, 2002). In general, GOOAL imposed restrictions on requirement texts. Analysts have to rearrange the collected requirement description by following the regulations and need to validate the interpretation of 4W language results. The tool can only process simple problem domains (Perez-Gonzalez & Kalita, 2002).

Subsequently, with the development of natural language processing technology, NLP techniques in the application of information extraction (IE) systems suggest promising approaches that may assist the requirement analysis process. Researchers started to adopt a combination of NLP tools and human experience rules in the development of UML model generation application.

3.3. Class Model Builder

Harmain and Gaizauskas in 2003 proposed their **Class Model Builder (CM-Builder)** in producing UML models from requirements. Rather than relying on deep analysis approach as it required labour intensive manual process, researchers explored a domain independent “semantic” analysis approach which could compute richer syntactic analysis than based on surface analysis. They started to use NLP tools of the era and designed a pipeline of tokenization, sentence splitter, POS tagger, morphological analyser and parser in sequence on processing requirement text.

CM-Builder has been evaluated quantitatively to prove its practical value. The tool relies on a number of certain or frequent words that appear in the requirement specification. For example, “is made up of”, “is composed of”, “contains” indicates a relationship of aggregation relationship (Harmain & Gaizauskas, 2003). CM-Builder initiates an NLP-based methodology to produce a full-fledged tool, but similar to previous researches, it presents the users with lists at each step of the pipeline, rather than developing an integrated process without human intervention. Using the case study by Harmain and Gaizauskas (2003), Dewar et al (2005) investigated CM-Builder, LIDA (Overmyer et al., 2001) and GOOAL (Perez-Gonzalez & Kalita, 2002), and concluded that none of them could fully extract classes from NL requirement. In addition, CM-Builder has limitations when drawing candidate class models, because the mechanism for acquiring objects is not appropriate (Dewar et al., 2005; Osman & Zălhan, 2016).

3.4. UML Generator from Analysis of Requirements

In 2009, Babar and Deeptimahanti developed a semi-automated tool to generate static and dynamic UML models, called **UML Generator from Analysis of Requirements (UMGAR)**. The tool was implemented by means of innovative approaches. Firstly, the core component is a set of syntactic reconstruction rules that transform complex sentences to simple ones. Secondly, Rational Unified Process (RUP) and ICONIX process are combined. The former helps to identify all possible classes and methods, attributes and relationships, on which the latter enhances the class identification process in preparation for the generation of collaboration diagrams. While the previous studies dealt with a small number of requirements (< 200 words), they used Stanford Parser which can tagged a larger number of requirement text (Babar & Deeptimahanti, 2009). At the same time, other two efficient NLP tools were adopted. WordNet 2.1 for morphological analysis (converting plural into singular) and JavaRAP for noun form correction (Babar & Deeptimahanti, 2009). A remarkable feature is that UMGAR provides a generic XMI parser to generate XMI file as output (Babar & Deeptimahanti, 2009), so that user can visualize the output model in any other graphical modelling tool. However, UMGAR needs human intervention during the process of irrelevant classes elimination and relationship identifications (Osman & Zălhan, 2016). There are also restrictions on input sentences due to the syntactic reconstruction rules they create. Every input sentence has to satisfy the rules, otherwise the user will be asked to modify the sentence. However, their syntactic reconstruction rules have inspired later researchers when dealing with the normalization process of textual data.

3.5. Diagram Class Builder

Herchi and Abdessalem (2012) created the **Diagram Class Builder (DC-Builder)** by employing NLP and domain ontologies to produce UML class diagrams. The General Architecture for Language Engineering (GATE) framework was utilized in this tool, as it provided *“the foundational building blocks for higher level text understanding applications”* (Herchi & Abdessalem,

2012). A set of heuristic rules was integrated to extract classes, attributes and relationships. The extracted information then was saved into an initial structured XML file. A following domain ontology block was used for XML file refinement (Herchi & Abdessalem, 2012). Compare to previous tools, DC-Builder improves the accuracy of extracting results from requirement text, but excludes methods and multiplicity extraction.

3.6. Requirements Analysis to Provide Instant Diagrams

While in the same year, More and Phalnikar (2012) extended the research by Babar & Deeptimahanti, 2009 (UMGAR), and proposed a desktop tool called **"Requirements Analysis to Provide Instant Diagrams" (RAPID)**. The tool employed NLP technologies such as OpenNLP for lexical and syntactic analysis, RAPID's own Stemming Algorithm for the base words and WordNet2.1 for semantic correctness (Osman & Zălhan, 2016). The syntactic reconstruction rules from UMGAR (Babar & Deeptimahanti, 2009) were refined and adopted. This indicates that it still requires users to change input sentence if the sentence violates the rules.

3.7. Automatic Builder of Class Diagram

Automatic Builder of Class Diagram (ABCD) is another UML class generation tool proposed by Azzouz et al in 2015. The Stanford NLP toolkit was used for lexical and syntactical analysis. A pattern-matching NLP technique was developed to extract the types of relationship and multiplicity of the identified classes. Similarly, the output is saved as XMI format files and can be imported into other visualization tool to build diagrams. However, the ABCD system has weaknesses in dealing with the problem of redundant information extraction, and confusion in relationship and method identification. (Osman & Zălhan, 2016)

3.8. UML Generator

Narawita and Vidanage (2016) proposed a web-based application: **UML Generator**, to produce UML class and Use case diagram automatically. Similar basic processing required for requirement text such as part-of-speech tagging; tokenization was implemented. At the same time, they adopted a rule-based approach, defining a set of XML rules to structure the output information and filter words. In addition to these methodologies employed in previous studies, a Weka model was trained to recognize the type of relationship and multiplicity, and vote for use case. Finally, two diagrams i.e., UML class and use case diagrams are generated.

This research provides a state-of-the-art approach to the field of NLP-based UML generation, combining used NLP tools, rule-based algorithm and a trained Weka model to accomplish different tasks. However, user has to follow the structure “subject-object-predicate” to enter each requirement sentence. This indicates a limitation on processing complex sentences. Another tiny limitation is that user cannot input “class” in the text due to a small bug existed in Weka (Narawita & Vidanage, 2016). This leaves the researcher with the challenge of exploring other tools and development methodology for enhancement.

3.9. Discussion

These research works have provided valuable insights into how NLP can be employed in the software requirement analysis process. Each of these tools has its advantages and weaknesses that has been described in previous sub sections, which also indicates that the NLP-based tools have not yet risen as a common use in real practice of software requirement analysis.

The majority of those development methods rely on rules heavily and placed NLP toolkits as a complementary role to extract the elements of UML models. These rules have been shaped by the experience and knowledge of requirement analysis experts. The advantages of using this approach are, firstly, that the rules become more refined as the research develops. For example, UMGAR (2009) proposed syntactic reconstruction rules, and RAPID

(2012) utilized these rules and refined the extraction rules for each element. Secondly, the use of rules is primarily a declarative approach that leads to a highly transparent, readable, and maintainable system (Waltl et al., 2018). However, as this field of study has evolved, the rules have become sophisticated, and have been framed differently in previous tools. Some of the rules are not appropriate, such as the rule proposed by Abdessalem & Herchi (2014) to indicate the presence of an attribute when a noun phrase succeeds a “has/have” verb phrase, which is too absolute as it can also refer to a class that has a composition relationship to the antecedent subject. In addition, there are several studies (e.g. Perez-Gonzalez & Kalita, 2002; Babar & Deeptimahanti, 2009; More & Phalnikar, 2012; Narawita & Vidanage, 2016; etc.) that propose syntactic reconstruction rules for the input requirement sentences, or require a “Subject, Predicate, Object” format to simplify the requirement sentences. This approach adds an additional layer of activity to the requirement analysts, an activity of converting NL text into structural text performed by a human rather than the system itself.

Machine learning has been applied to complement the rules in the element extraction process. For example, Narawita & Vidanage (2016) used a trained Weka model to vote for relationship and multiplicity extraction results of a UML class model. Machine learning can assist in providing a more accurate result of information extraction. However, as mentioned in chapter 2.2.2, one of the first step and the first bottleneck when applying it to a new application is gaining or creating enough training data sets (Roh et al, 2019), as supervised learning requires (large) amount of training data. In the software requirements domain, pre-labelled samples are scarce since these requirements come from industry usually in closed domain. Ferrari et al (2017) provided a dataset of 79 public software requirement document. These documents are Software Requirement Specification (SRS) which is a documentary result from traditional requirement analysis process. A few documents contain NL requirement paragraphs as well as their corresponding UML class diagram, but they are inconsistent with each other. It is the case that such recent applications have little or no useful training data.

4. System Architecture

Based on the discussion of previous approaches to the development of UML class model generation tools, this study presents a novel architecture framework. Rather than formulate rules on input sentences, our system performs text structuring tasks by using NLP toolkits to transform raw text into a set of triplets with “Subject, Relation, Object” format. This reduces the unnecessary activity that user structure the requirement texts themselves.

Our proposed tool implementation is structured as a web-based application. The backend analysis model consists of two parts, namely text structuring process with NLP toolkits and rules to support element extraction. In this section, an overall design of this application is given, which consists of a high-level architecture, an activity diagram for application processing, and a set of functional requirements on UML class metadata extraction. The core component (i.e., extraction methodology) are elaborated in Chapter 4.2.

4.1. Conceptual application framework

4.1.1. High-level architecture

Figure 5 is the high-level architecture of our designed tool.

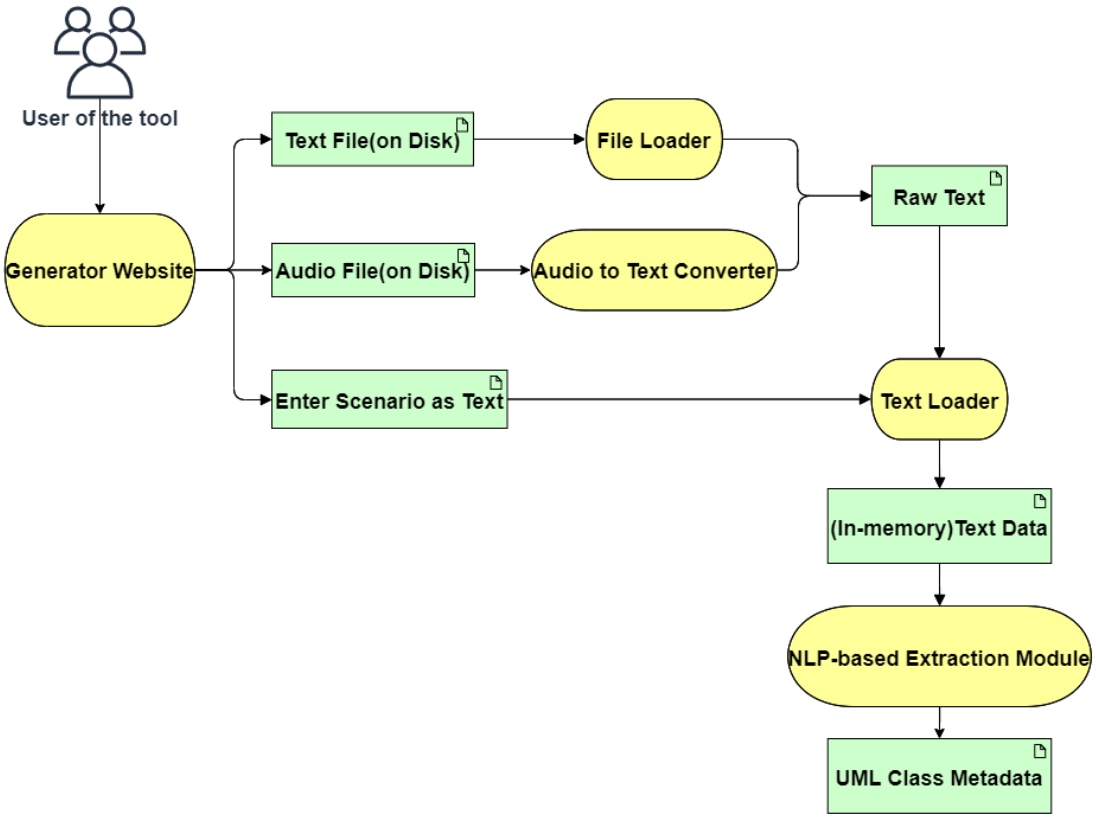


Figure 5. High-level Architecture of our application

There are five main sections in this web application. Each of them is connected by input and output objects such as file and text objects. Users of the system initially interact with the generator website by uploading a local text file that contains NL requirements, or an audio file that records a speech for requirements or entering NL requirements in the text area. The aim of providing these two options is to reduce time-consuming requirement transcription. Instead of spending time recording the requirements from stakeholders and subsequently transcribing them after meetings, the stakeholders and requirement analysts can sit together, using this application to view their requirements and make modifications immediately. The file loader and the audio to text converter extract the original text from the input requirement file. Text loader transfers these text data into memory. The application extracts UML class metadata by using NLP-based extraction module. It is a module that adopts NLP toolkits and extraction rules to identify UML class metadata such as the data of classes, attributes, relations and directions. The application displays the UML class metadata output in a text

area that users can check and modify the results. Therefore, the output is customizable and users can receive a direct result according to their requirement text.

In order to get other stakeholders such as business experts and end users involved in requirement analysis process, our application integrates an outside solution developed by Driessen (2020) to show a runnable prototype results based on the extracted results from requirements texts. Figure 6 displays another high-level architecture design of the integration work, continued with the output of Figure 5.

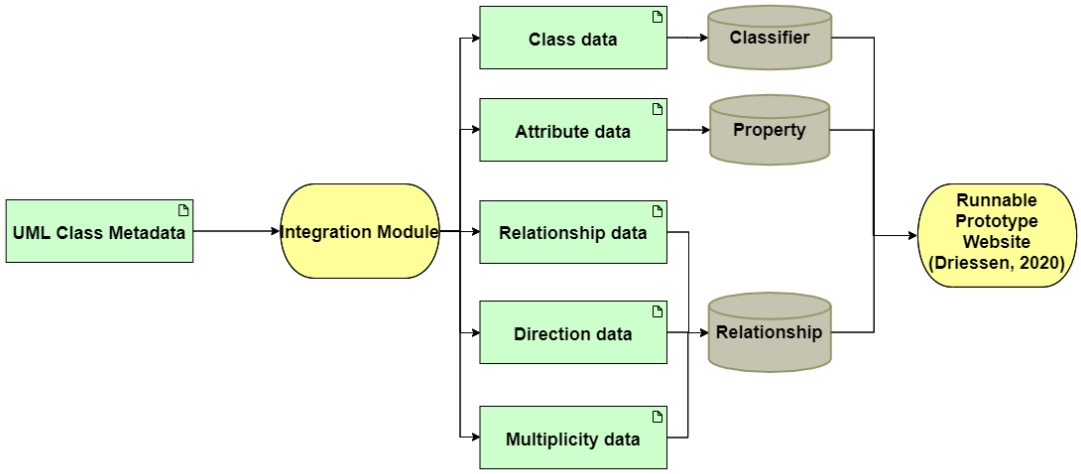


Figure 6. High-level Architecture for Integration

The extracted results are separated into different data sets, and saved in a database created by Driessen (2020). By working through the integration, other stakeholders can not only see the results of the UML metadata or visual model, but also have immediate access to a running prototype application. This allows them to highlight omissions, errors and modifications. A more detailed explanation and implementation is described in Chapter 4.3.

4.1.2. Application process

To describe the process of our application, figure 7 is a UML activity diagram emphasising sequential activities for each section, where the shaded swim

lane indicates functional modules (e.g., NLP-based extraction module) of the process. This defines the main back-end analysis procedure for UML class metadata extraction purposes. The NLP-based extraction module will be described in detail in Chapter 4.2.

In the initialization step, the user performs a decision activity on determining to upload a requirement file in form of text or audio, or copy and paste the requirement text on a text area provided by the generator website. Alternatively, the user can enter text directly into this area. If a user determines to upload an audio file, the “Audio to Text Converter” in Figure 5 utilized SpeechRecognition library with its’ built-in method of Google Speech API to transform the speech to text. SpeechRecognition is a library support for several speech recognition APIs such as Google Speech Recognition, Google Cloud Speech API, Microsoft Bing Voice Recognition, IBM Speech to Text and so on.

The generator website displays the converted audio requirement text or the uploaded requirement text in a text area. The user can check and modify the initial requirement text if needed, and enter a title for the requirement. Text loader loads them as stored text data. The NLP-based extraction module then performs structuring and extracting activities on this stored text data sequentially. The final activity is performed in the generator module, which displays the UML class metadata results to the user.

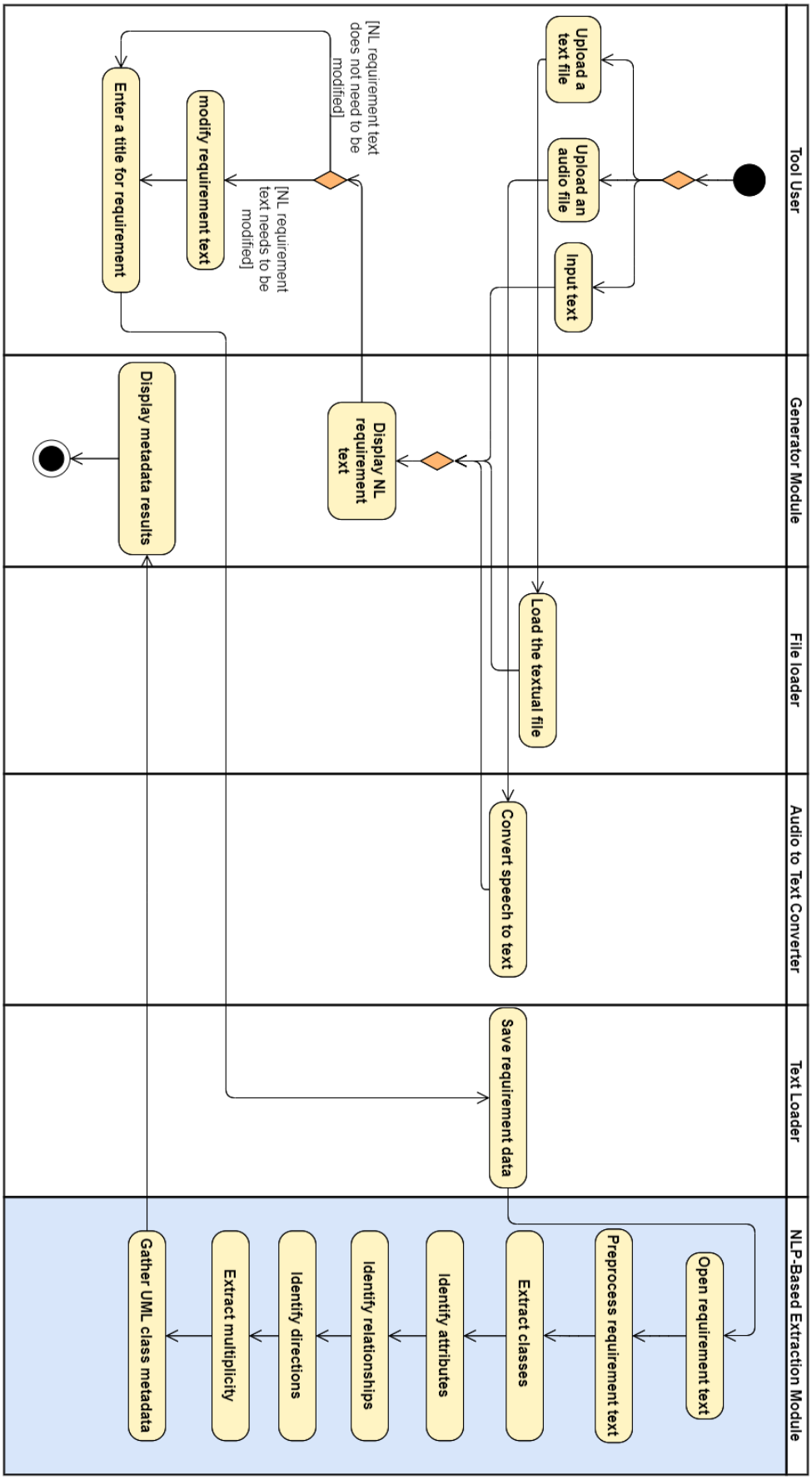


Figure 7. Activity Diagram Overviewing Process

4.1.3. Functional requirements

The NLP-based extraction module is the core component that processes and analyses NL requirement texts in this application. A specification of the functionality for the NLP-based extraction module is described as follows:

- Recognize Classes

Extract Classes from NL requirement text. The Class name must be in a singular format with initial capitalization.

- Recognize Attributes for each Class

Extract Attributes for identified Classes from NL requirement text if in existence. Take a sentence as an example: *"A customer has an id, name, and address."* The extraction result should be *"Class: Customer; Attributes: id, name, address"*.

- Extract Relationships between identified Classes

The Relationship result should involve four characteristics: Relationship type and value, Directionality, and Multiplicity. There are three sub-functional requirements in Relationship extraction:

a. Recognize the type and value of Relationship:

Detect different types of Relationship from the user input text. There are three types of Relationship encompassed in this extraction process: Association, Subtyping, and Composition. The result format should be "Relationship type: Relationship value". Take a sentence as an example, *"A customer places an order."* (which will be used as an example sentence for the remaining sub-functional requirements) The Relationship extraction should be *"Association: places"*.

b. Recognize Directionality between identified Classes

Identify Direction between Classes. The format should be "Directionality (from): Class name; Directionality (to): Class name". An extraction result for example sentence should be *"from: Customer; to: Order"*.

c. Extract Multiplicities between Classes

The Multiplicity value should be attached to each Class. To continue and integrate with Directionality result: "Directionality (from): Class name; Multiplicity value; Directionality (to): Class name; Multiplicity value". An

extraction result for example sentence is *“from: Customer; multiplicity: 0..*; to: Order; multiplicity: 0..*”*.

- Display a complete UML Class metadata result

Integrate all the data gathered from Class, Attribute, and Relationship extraction, and display them to the user. For example, the ultimate results of the example sentence displayed on the front end are as follow:

Class: Customer

Class: Order

Association: places

from: Customer

*multiplicity: 0..**

to: Order

*multiplicity: 0..**

4.2. NLP-based Extraction Module Architecture

To achieve the functional requirements (i.e., class and attribute identification, relationship identification with type, value, direction and multiplicity), the NLP-based extraction module is decomposed into two main components: i) a text structuring process with NLP tools, and ii) a rule-based extraction block to extract UML class metadata from structured text data.

Since both IE and Open IE are often placed as an early stage before pursuing higher level tasks in a more specialized NLP application (Singh, 2018; Chikkamath et al., 2018), we utilize the steps in our text structuring block to transfer the raw text into a list of tuples containing triplets. Then, rules are defined for further metadata extraction. Figure 8 elaborates the process architecture of the NLP-based extraction module.

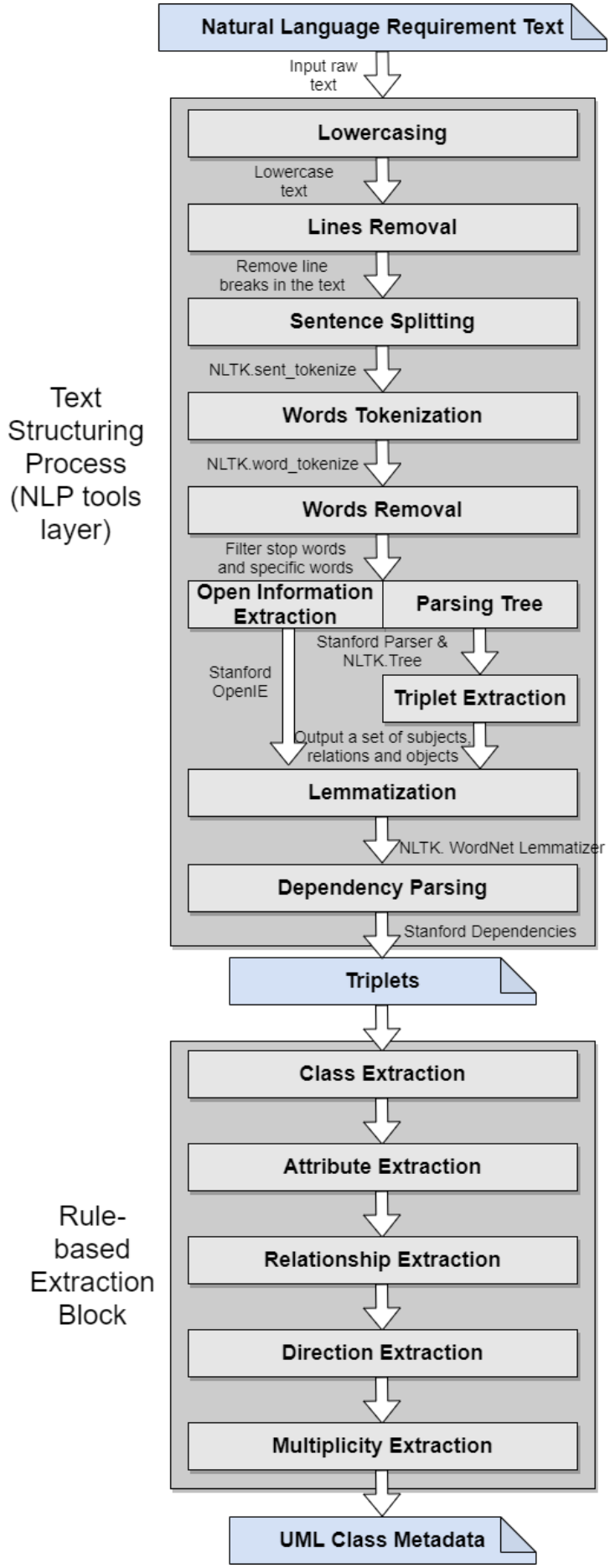

Natural Language Requirement Text
Input raw text
Lowercasing
Lowercase text
Lines Removal
Remove line breaks in the text
Sentence Splitting
NLTK.sent_tokenize
Words Tokenization
NLTK.word_tokenize
Words Removal
Filter stop words and specific words
Open Information Extraction
Parsing Tree
Stanford Parser & NLTK.Tree
Stanford OpenIE
Triplet Extraction
Output a set of subjects, relations and objects
Lemmatization
NLTK. WordNet Lemmatizer
Dependency Parsing
Stanford Dependencies
Triplets
Class Extraction
Attribute Extraction
Relationship Extraction
Direction Extraction
Multiplicity Extraction
UML Class Metadata
Text Structuring Process (NLP tools layer)
Rule-based Extraction Block

Figure 8. NLP-based Extraction Module Architecture Design

4.2.1. Text Structuring Process (NLP tools layer)

Text cleaning is an initial and integral part of any NLP-based system (Palmer, 2010). In this application, requirement documents and input NL requirement texts are typically unstructured or semi-structured data which is difficult to process immediately. The objective of this block is to transform raw text data into a predictable and analysable form for subsequent processing tasks. There are various information extraction (IE) steps for text cleaning and structuring such as lowercasing the characters, remove punctuations and stop words, stemming and lemmatization, etc.

With the rapid development of NLP technologies, there are many NLP toolkits available for performing common NLP activities, which enable the development of NLP-based applications without having to start from scratch (Pinto et al., 2016). In text structuring module, we work with Natural Language Toolkits (NLTK) developed in Python, as it provides a mature IE pipeline architecture framework and gives freedom for developers to use those steps in a framework. In addition, we utilize the library package from Stanford CoreNLP. It provides open information extraction package (Stanford Open IE) that is utilised for triplet extraction.

In general, there is no rigid standard procedure for an information extraction system. The steps and methods should satisfy the needs, and adapt to the purpose of a program. The purpose in this component is to output the triplets that represent the input requirement text. The following is a sequential listing of the relevant steps:

1. Lowercasing

Lowercasing is the simplest and a common text cleaning technique. The idea behind it is normalizing all the words from a text in a same casing format, so that they can be treated the same way and without causing further problems. For example, Python interprets uppercase and lowercase letters differently.

“Customers” and “customers” occur at different places in a textual document, but a system would treat them separately rather than process them as a single semantic concept. With mixed-case text data, lowercasing can eliminate variation and reduce vocabulary size. However, there are cases where lowercasing might have a negative by increasing ambiguity (Camacho-Collados & Pilehavar, 2018). Some typical examples are “Apple” company or “apple” as a fruit to be identified, “IT” as an abbreviation of information technology while “it” as referring to an object.

In our application, the module analyses requirement texts in English. Without lowercasing, the system might treat a word which is in the beginning of a sentence with a capital letter different from the same word which appears later in the sentence without any capital letter. This will lead to a decline in accuracy.

2. Line removal

NL requirement texts typically consist of a collection of paragraphs. Frequently, those paragraphs are distinguished by line breaks. When users upload their NL requirement document, strings of the entire requirement text including line breaks will return to the back-end program. To keep the string being displayed in one line containing the entire requirement text, we should remove all the line breaks in advance.

3. Sentence Splitting

After the previous steps, the text data is integrated without any breaks and is returned as a string. The objective of our structuring component is to convert raw text in structured format (i.e. triplets containing subject, relation and object) for every sentence, so that each triplet represents a sentence. Therefore, it becomes vital to segment the string of text into sentences to achieve the above-stated purpose. Sentence splitting in our case is considered as a base step because we need to keep the data in list of sentences before the step of open information extraction. We can simply split a sentence by delimiters like a period (.) Taking a simple text as an example *“A customer places one or more orders. An order is for multiple products.”*

Sentence splitting output will be *['a customer places one or more orders.', 'an order is for multiple products.']* A simple piece of code using default sentence tokenizer from NLTK is as follows:

```
>>> import nltk
>>> from nltk.tokenize import sent_tokenize
>>> text='A customer places one or more orders. An order is for multiple products. '.lower()
>>> sents=sent_tokenize(text)
>>> print(sents)
['a customer places one or more orders.', 'an order is for multiple products.']
```

4. Word Tokenization

Tokenization in general is a process of breaking up textual data by locating the word boundaries into smaller and more meaningful components called tokens (Palmer, 2000). The common types of tokenization include sentence and word tokenization. In our structuring process, we break down a text document into sentences and tokenize them into words. For example, word tokenization for sentence *"A customer places one or more orders."* will be *['a', 'customer', 'places', 'one', 'or', 'more', 'orders', '.']* Word tokenization is in necessary because it can be provided as an input for further text processing procedures such as punctuation elimination, lemmatization, stemming, etc. In our program, the purpose of word tokenization is to prepare for data cleaning with stop words removal. Sample Python code using the method word_tokenize() in NLTK after sentence splitting is shown as follows:

```
>>> for s in sents:
...     word_tokens = nltk.word_tokenize(s)
...     print(word_tokens)
...
['a', 'customer', 'places', 'one', 'or', 'more', 'orders', '.']
['an', 'order', 'is', 'for', 'multiple', 'products', '.']
```

5. Word Removal

To remove unwanted words, our application specifies a list of excluded words. Some of the words belongs to stop words in NLP application. Stop words refer to the words in natural language that do not add much meaning in a text. Those words can be eliminated without ruining the meaning of a sentence. Examples of stop words are *"the"*, *"a"*, *"an"*, etc.

One of the reasons to remove stop words is that system can concentrate on more valuable information in downstream processing steps rather than spending time on analysing meaningless words. Nevertheless, stop words removal is not an obligation in every application of data cleaning. It should be considered thoughtfully because stop words mean differently in different applications. In our research, rather than using a corpus of stop words provided by NLP tools, we recompile an excluded word list based on the corpus of stop words provided by NLTK and append other specific words that are meaningless in requirement text as metadata for UML class model. The code below is our predefined excluded words lists in our application:

```
# word lists
excluded_words_1 = ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your', 'yours', 'yourself',
                    'yourselves', 'he', 'him', 'his', 'himself', 'she', 'her', 'hers', 'herself', 'it', 'its', 'itself',
                    'themselves', 'what', 'who', 'whom', 'this', 'these', 'those', 'examples', 'but', 'because',
                    'until', 'while', 'at', 'about', 'between', 'if', 'with', 'through', 'during', 'after', 'above',
                    'below', 'up', 'down', 'able', 'form', 'of', 'off', 'over', 'under', 'again', 'further', 'then',
                    'once', 'here', 'there', 'where', 'why', 'how', 'all', 'both', 'each', 'few', 'most', 'other',
                    'some', 'no', 'nor', 'not', 'only', 'own', 'same', 'than', 'too', 'very', 's', 't', 'just', 'don',
                    "don't", 'should', 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn',
                    'didn', 'doesn', "doesn't", 'hadn', 'hasn', 'haven', 'isn', 'ma', 'mightn', 'mustn', 'needn',
                    'shan', 'shouldn', 'wasn', "wasn t", 'weren', 'weren t', 'won', 'won t', 'wouldn', 'wouldn t',
                    'multiple', 'many', 'forward', 'etc', 'shall', 'also', 'therefore', 'might', 'able', 'various',
                    'necessary', 'several', 'usually', 'must', 'finally', 'different', 'firstly', 'corresponding',
                    'enough', 'relevant', '’s', 'furthermore', 'desired', 'typically', 'initially', 'additional']
excluded_words_2 = ['their', 'such', 'as', 'them', 'will', 'that', 'when', 'they', 'for', 'may', 'types', 'specific',
                    'particular']
design_elements = ['system', 'user', 'application', 'data', 'computer', 'object', 'information', 'interface', 'online']
```

The separate excluded words list is used before Open IE and after Open IE respectively. The purpose is to remove certain words without violating the integrity of a sentence, so that Open IE can process the sentence and extract a more accurate result.

After obtaining triplet results, a further “design element” word removal is performed in order to clean the non-relevant terms. This is based on a list containing high-level words such as “application”, “system”, “user”, “data”, etc, because these words are related to (system) design elements, which should be avoided as classes (Narawita & Vidanage, 2016; More & Phalnikar, 2012).

6. Open Information Extraction

Open Information Extraction (Open IE) in NLP is a task of generating a structured, machine-readable representation of the information in a text,

usually in the form of triplets, where a triplet usually consists of subject, predicate, and object in sequence to represent a fact (Chikkamath et al., 2018; Khairova et al., 2020). Taking the example sentence *"A customer places one or more orders."*, represented in an appropriate structure for computers to process is *[("Customer", "places", "Order")]*. The subject and object arguments are often expressed by nouns or noun phrases, while the predicate indicates a relation expressed by verbs frequently. Our application adopts Open IE to extract triplets from requirement text, and subsequently the system identifies further extraction of UML Class metadata from those triplets. Example code using StanfordCoreNLP in Python using the OpenIE package is as follows:

```
>>> import logging
>>> from stanfordcorenlp import StanfordCoreNLP
>>> # settings
... class SubStanfordCoreNLP(StanfordCoreNLP):
...     def __init__(self, path_or_host, port=None, memory='4g', lang='en', timeout=1500, quiet=True,
...                  logging_level=logging.WARNING):
...         super(SubStanfordCoreNLP, self).__init__(path_or_host, port, memory, lang, timeout, quiet, logging_level)
...
...     def open_ie(self, sentence):
...         r_dict = self._request('openie', sentence)
...         openies = [(ie['subject'], ie['relation'], ie['object'])
...                    for s in r_dict['sentences'] for ie in s['openie']]
...         return openies
...
>>> nlp = SubStanfordCoreNLP('C:\stanford-corenlp-full-2018-10-05')
>>> text = 'An order is for multiple products.'.lower()
>>> word_tokens = nltk.word_tokenize(text)
>>> filtered_tokens = [w for w in word_tokens if w not in stop_words1]
>>> sent = ' '.join(filtered_tokens)
>>> ies = nlp.open_ie(sent)
>>> print(ies)
[('order', 'is for', 'products')]
```

7. Parsing tree

A parsing tree in NLP is a way of representing the syntactical structure of a text in a tree graph. The syntactical structure is produced after basic NLP tasks such as tokenization, part of speech tagging, chunking to reveal and group syntax of a sentence. In this process, the parsing tree is utilized when the step 6 (Open Information Extraction) fails to extract some specific sentence.

Typical syntactical categories are S (Sentence), NP (Noun Phrase) where it usually contains labels such as NN (Noun), NNS (Plural nouns), etc., DT (Determiner), VP (Verb Phrase), PP (Prepositional Phrase), and so on. Figure 9 is an example of a parsing tree for sentence *"The quick brown fox jumps over the lazy dog"*:

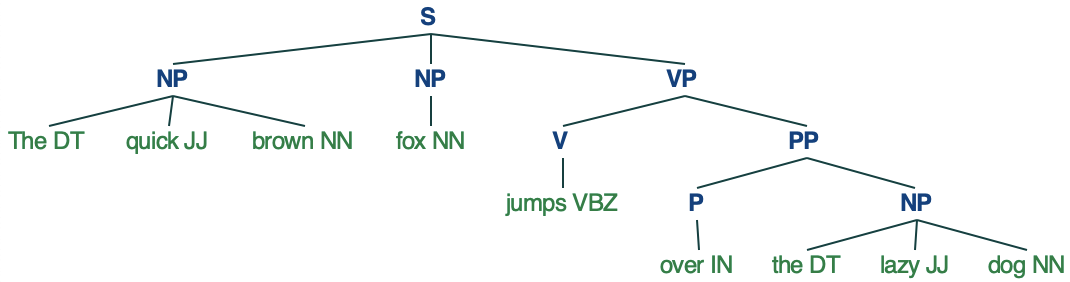


Figure 9. Parsing tree example

This step helps us in identifying the main parts like subject, predicate and object in a given sentence (Btoush & Hammad, 2015).

While Stanford OpenIE is intended for large-scale relation extraction from text such as Wikipedia, currently, it sometimes fails to extract triplets for certain sentences such as a sentence without object or verbal components, or a sentence where the POS of object is defined as verbs by Stanford OpenIE. When it fails to process a sentence, it will return empty. In this case, Stanford Parser is employed to parse the sentence that cannot be extracted by Stanford OpenIE, and use NLTK.Tree to represent it before adding our own rules to extract triplets. Thus, parsing tree is a replacement process once Stanford OpenIE fails. The Classes and Relationships can be extracted by exploring the labels (i.e., syntactic categories such as NP, VP, NN) and corresponding leaves (i.e., values like "fox", "dog") of a tree. An example of codes for computing parsing tree is as follows. The rules to extract triplet is illustrated in Chapter 4.2.2.

```
>>> from nltk.tree import Tree
>>> parser = nlp.parse(sent)
>>> tree = Tree.fromstring(parser)
>>> print(tree)
(ROOT
  (S
    (NP (DT an) (NN order))
    (VP (VBZ is) (PP (IN for) (NP (NNS products))))
    (. .)))
```

8. Lemmatization

Lemmatization refers to turning a word into its corresponding lemma (i.e., dictionary form). For instance, *"places"*, *"placing"*, *"placed"* are all forms of the word *"place"*, so that *"place"* is the lemma of all those words. There are various purposes and emphasises to use lemmatization in different applications. For example, in web document clustering for search engines, lemmatization is employed to reduce the number of tokens with identical meanings but different forms, and increase the system performance. For our program, except for the reason stated above, an extracted Class name must be in singular form. Therefore, we adopt lemmatization as a final step to transform the plural class name into singular. NLTK offers lemmatization using WordNet's built-in morphological analysis function:

```
>>> from nltk.stem import WordNetLemmatizer
... lemmatizer = WordNetLemmatizer()
... print(lemmatizer.lemmatize('customers',pos='n'))
customer
>>> print(lemmatizer.lemmatize('Customers',pos='n'))
Customers
```

As can be seen, lemmatization in NLTK returns the input word unchanged if it cannot be found in WordNet and confirms the necessity of lowercasing step.

9. Dependency Parsing

Dependency parsing is a process of representing the grammatical structure of a sentence based on the dependencies between words in the sentence. Figure 10 displays a result of analysing *"A customer places an order. An order*

is placed by a customer" by dependency parser through StanfordCoreNLP demo.

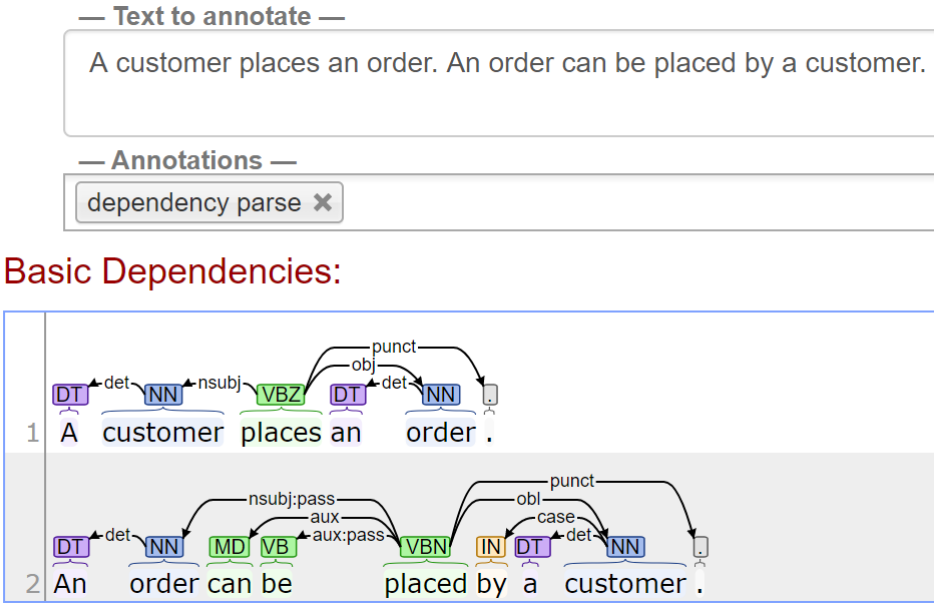


Figure 10. Stanford Dependency Parser Demo

As can be seen from the dependency parsing result, there is a "det" (determiner) relation between "order" (NN) and its determiner "An" (DT). The "aux" (auxiliary) and "auxpass" (passive auxiliary) in the second sentence indicates dependency relationship between "placed" and "can", "placed" and "be", respectively. In our application, dependency parsing is used to recognize passive and active voice according to the label "nsubj" (nominal subject) and "nsubjpass" (passive nominal subject). In this way the Directionality between Classes can be determined. Lemmatization is cooperated in this step to transferred the Relationship value from passive voice to active voice. The built-in method dependency_parse() of StanfordCoreNLP and lemmatization to transform verb format for passive Relationship value are developed in the code as follow (with the corresponding result from an input sentence *"An order is placed by a customer"*):

```
# using dependency parser to check and define direction between classes
tri = list(s)
joint_s = ' '.join(tri)
dep = nlp.dependency_parse(joint_s)
print(dep)

firstelement = []
for d in dep:
    if d[0] not in firstelement:
        firstelement.append(d[0])

if raw_dir[0] != raw_dir[1]:
    if 'nsubj' in firstelement:
        dir['from'] = raw_dir[0]
        dir['to'] = raw_dir[1]
    elif 'nsubjpass' in firstelement:
        dir['from'] = raw_dir[1]
        dir['to'] = raw_dir[0]
    else:
        dir['from'] = raw_dir[0]
        dir['to'] = raw_dir[1]

    return dir

else:
    return None
if 'nsubjpass' in firstelement:
    pos = nlp.pos_tag(s[1])
    for p in pos:
        if p[1] == 'VBN':
            convertrel.append(p[0])
    # using lemmatizer to transfer passive verb to active
    relname = lemmatizer.lemmatize(convertrel[0], pos='v')
```

```
an order is placed by a customer .
[('order', 'is placed by', 'customer'), ('order', 'is', 'placed')]
[('order', 'is placed by', 'customer')]
[('ROOT', 0, 3), ('nsubjpass', 3, 1), ('auxpass', 3, 2), ('case', 5, 4), ('nmod', 3, 5)]
[[('Association', 'place'), ('from', 'Customer'), ('to', 'Order'), ('multiplicity', ['0..*', '0..*'])]]
```

Algorithm

Having described all of the architectural steps for the text structuring process (with Python code examples), an algorithm integrating these steps is as follows:

Algorithm 1 Text pre-processing (NLP tools layer)

Input: NL requirement text file from user input;
Output: *Triplets* (subject, relation, object) for each sentence;

Lowercase the characters from requirement text file, and concatenate paragraphs into a string variable *initial_text*;
Tokenize *initial_text* by sentence as a list variable S;
for each sentence (i) in S **do**
    Tokenize i by words into a list variable *tokens*;
    Remove *excluded_words_1* and *design_elements* from *tokens*;
    Join the cleaned *tokens* into a string as a prepared sentence;
    Use *Stanford OpenIE* to extract triplets from the sentence;
    if *Stanford OpenIE* fails **then**
        Use *Stanford Parser* and *NLTK.tree* to get a parsing tree;
        Utilize tree structure and part-of-speech tag to extract triplet;
        Save triplet into a list variable *Triplets*;
    else
        Remove *excluded_words_2* from triplets;
        Save the cleaned triplets into the list *Triplets*;
    end if
    for each triplet (t) in *Triplets* **do**
        Join the three arguments from t to a simple sentence s;
        Use *Stanford Dependency* to parse the s;
        if "nsubjpass" exists **then**
            Interchange the position of subject and object;
            Use *NLTK.Lemma* to transform the form of predicate;
            Update *Triplets*;
        end if
    end for
return *Triplets*;

4.2.2. Rule-based extraction block

A rule-based system is an automated system encoding human expert's knowledge in a narrow area, and is usually made up of sets of rules or assertions, where the rules are expressed as if-then statements (Grosan & Abraham, 2011; Narawita & Vidanage, 2016). In our application, rules are formulated to overcome the limitations of NLP tools and for further information extraction steps which are specific to UML class meta data.

Additional triplet extraction rules

When Stanford OpenIE fails to process a sentence, extra rules are defined to identify triplets from a parsing tree of that sentence. A heuristic rule of identifying Subject-Relation-Object is that nouns are often referred to subjects and objects, while relations are usually expressed by verbs. Therefore, we have following rules on triplet extraction when exploring a parsing tree:

1. Find the first label NP (noun phrase) occurs in a sentence, if there is any label as NN (noun, singular or mass), NNS (plural noun), and NNP (singular proper noun) in the noun phrase, then assign the corresponding value as a subject.
2. Find if there is a label VP (verb phrase), NP (noun phrase), PP (prepositional phrase) occurs after NP (noun phrase), then if there is any label as VB (verb with base form), VBN (verb, past participle), VBZ (3rd person singular present verb) or VBP (non3rd person singular present verb) in those three types of phrases, assign the text value as a predicate in a triplet.
3. Continue from the second rule set to explore if there are any labels NN, NNS or NNP in VP, NP, PP, then assign the text value as an object.

Since natural language is complex and ambiguous, the accuracy of applying the above rules cannot compare with that of Stanford OpenIE. The rules perform very well in syntactic structures like NP+VP (e.g., *"A customer can place one or more orders"*). However, they are less effective with a more complicated sentence structure composed of multiple NP and VP labels or a sentence with more than three candidate Classes (concept) value.

UML meta data extraction rules

The syntactic reconstruction rules which are developed in UMGAR (Babar & Deeptimahanti, 2009) and reinforced in RAPID (More & Phalnikar, 2012), focused on restricting input sentence from users. Other rules are focused on metadata extraction, for example, More and Phalnikar (2012), Shinde et al (2012) designed more than 10 rules on Class, Attribute and Relationship extractions. Our application defines fewer and more straightforward rules for

metadata extraction due to the results of the last NLP toolkits layer process, presenting a set of clean and structured triplets. For example, “Subject, Predicate, Object” directly implied as (Candidate Class, Relationship, Candidate Class).

With all the triplets result from previous block (Chapter 4.2.1), and the additional triplet extraction rules discussed above, the UML specific extraction rules are defined as follows:

1. Class extraction

We have discussed a rule for avoiding Class extraction in the previous block: remove the words related to (system) design elements such as “application”, “system”, “user”, etc. After these unintended nouns are removed, the values of every subject and object from the triplet results become a UML Class.

2. Attribute extraction

If a Class (subject and object value in triplet) has a value like “name”, “date”, “id”, “code”, “address”, etc, which are commonly regarded as attributes when defining UML Class models, then it is an Attribute. We collect and store a predefined list including the most popular Attribute words in our program. It is used to check the extracted subjects and objects value from triplets. The predefined Attribute word list contains the words as follow:

```
attribute_words = ['id', 'first name', 'last name', 'name', 'address', 'email', 'number','no', 'code', 'date', 'type',
                   'volume', 'birth', 'password', 'price', 'quantity', 'location', 'resolution date', 'creation date',
                   'crime code', 'course name', 'time slot', 'quantities', 'delivery date', 'prices',
                   'delivery address', 'scanner', 'till', 'illness conditions', 'diagnostic result', 'suggestions',
                   'birth date', 'order number', 'total cost', 'entry date', 'delivery status', 'description',
                   'product number']
```

However, the word list can be extended manually rather than learning from user’s modification of the extracted results. To extend the attribute glossary, more requirements texts should be obtained for manual analysis to identify common attribute terms.

3. Relationship extraction

The second value (i.e., predicate expressed by verbs or a verb phrase) from triplets becomes the name of a Relationship between classes. For example,

“A customer places one or more orders.”, *“places”* is the name of relationship between *“Customer”* and *“Order”*. We have mentioned in the functional requirements that this application focuses on three types of Relationship extraction. The rules utilised for Relationship type identification are:

- If the Relationship name is equal to one of the words or phrases “have”, “has”, “contains”, “contain”, “consists of”, “composed of”, “hold”, “include”, “maintain”, “maintains”, “divided to”, “has part”, “comprise”, “carry”, “involve”, “imply”, “embrace” and “is for”, then it indicates a Composition Relationship.
- If the Relationship name is equal to or includes one of the words or phrases like “is a”, “is a kind of”, “can be”, “is”, “are”. Then it indicates a Subtyping Relationship, and the Relationship name should be removed.
- If the Relationship name does not satisfy any of the previous rules, then it is defined as Association Relationship.

4. Relationship Direction extraction

The Direction of Relationships between classes is defined by utilizing the active and passive voice in the relationship name. If it is in active, the direction is defined from subject to object. If it is passive voice, the direction is defined from object to subject. For instance, “A customer places one or more orders.” (“Customer”, “places”, “Order”) “An order is placed by customers.” (“Order”, “is placed by”, “Customer”). Since the predicate in the latter triplet is in passive voice, the direction result for both triplets is from “Customer” to “Order”. These can be achieved by using **Stanford Dependencies** where “nsubj” represent active and “nsubjpass” represent passive. Therefore,

- If “nsubj” exists in dependency parsing result, the direction is from subject to object.
- “If “nsubjpass” exists in dependency parsing result, the direction is from object to subject.

5. Multiplicity extraction

Multiplicity is difficult to extract from a triplet that contains three arguments. Furthermore, requirement documents frequently omit explicit reference to association multiplicity. In addition, natural language can be ambiguous with

respect to multiplicity. For example, an NL requirement might be described as “A teacher gives lectures”, whereas in real life, it happens that more than one teacher gives a lecture at a school or university. If the algorithm strictly follows the structure and content of requirement text, the accuracy of multiplicity identification declines. The widest range “0..*” (zero to many) is not a hundred percent accurate result, it is not a faulty outcome. The users can refine the multiplicity result into smaller range. If the Relationship type between two Classes is Composition, the Multiplicity is given as “1” for the starting Direction (from) and “0..*” to the end Direction (to).

The very limited capabilities of our rule-based extraction module are:

- the identification method for Attributes is restricted to a predefined word set.
- The identification method for Multiplicity simply performs the widest range of “0..*”. The only update is based on the detection of the Composition Relationship type.

4.3. Implementation

Based on the requirements and architecture design, a web application has been developed in the Python environment using Django, which is a Python-based high level web framework. The back-end NLP and Rule processing is also implemented in Python. The implementation covers the architecture and algorithm described in Chapter 4.1 and 4.2, and uses an external program (Ralph Driessen (2020)) and dependent libraries (i.e. NLTK, Stanford CoreNLP and SpeechRecognition).

Our web application consists of three front-end web pages:

- The home page enables users to upload requirement text files, audio files, or input text directly. A separate pane on this page displays the extracted UML meta data in structured textual format.
- The second page enables users to modify uploaded requirement texts.
- The third page integrates a prototype run-time application environment based on the extracted UML Class meta data (Ralph Driessen (2020)).

This environment enables users to immediately obtain feedback on the UML Class meta data extracted in the form of a running prototype. It provides a rapid feedback mechanism on the UML Class model extracted from the NL requirements text by running the resulting application. It also contains a visual UML class modeler component to view the extracted meta data in that form.

Home page

The homepage of this application is displayed in Figure 11. The application can open and read textual requirements from two different sources including text files (.txt) and audio files (.wav). User can either uploads a file or input text in “Requirement text” area. When user click “Convert to text” button, text data from uploaded file will be displayed in “Requirement text” area. A title must be input in order to store data and enable requirement modification in the “Manage Requirement” page.

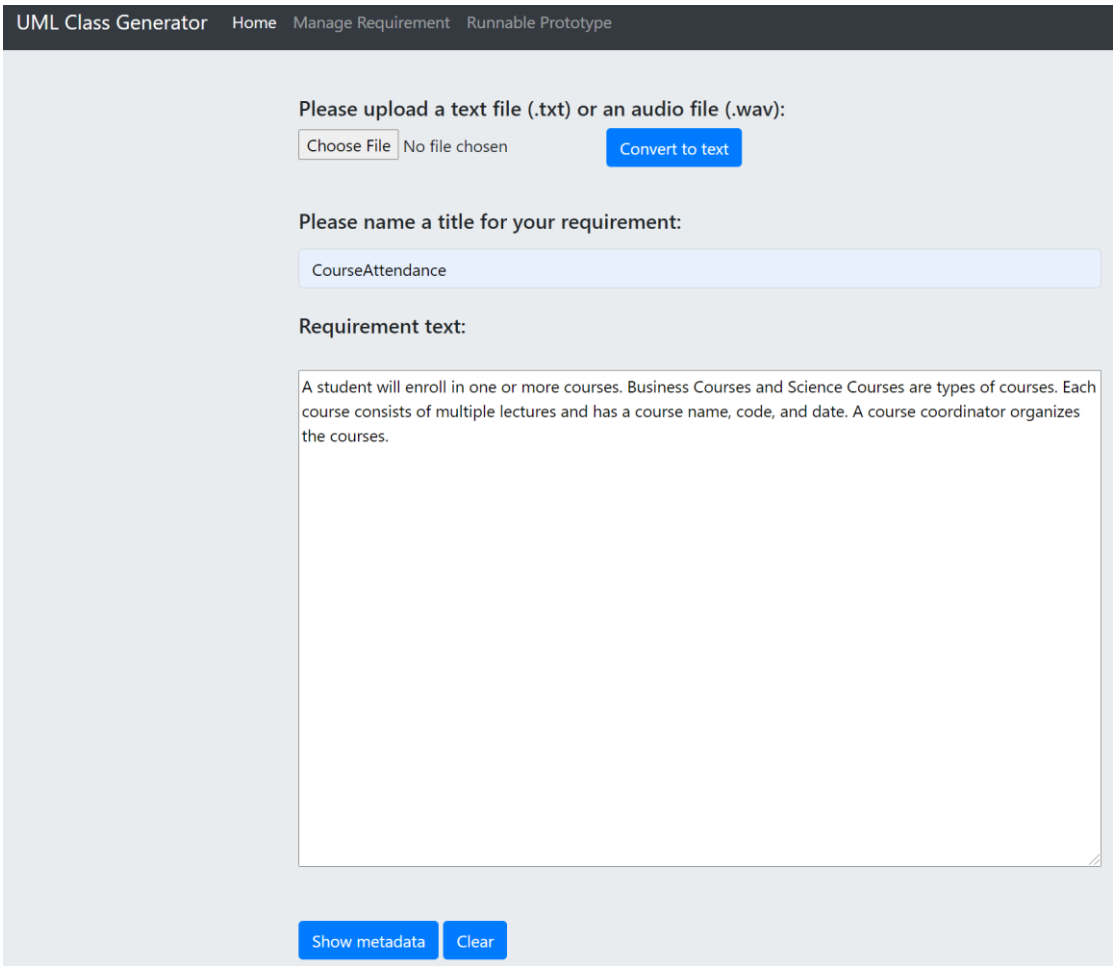


Figure 11. Home page of application

Apart from using Stanford CoreNLP and NLTK libraries, this application also uses SpeechRecognition, a python library for speech-to-text, or text-to-speech conversion, that has support for several engines and APIs online and offline such as Google Speech Recognition, IBM Speech to Text, Snowboy Hotword Detection, etc. By default, Google Speech Recognition is utilized in this application to transform the speech content from an uploaded .wav file to text. The Python code to achieve this is as follows:

```
if 'convert' in request.POST:
    form = UploadForm(request.POST)
    if form.is_valid():
        file = Document(file=request.FILES['file'])
        file.save()
        if request.FILES['file'].content_type == 'audio/wav':
            sound = AudioSegment.from_wav(file.file)
            chunks = split_on_silence(sound,
                                      min_silence_len=500,
                                      silence_thresh=sound.dBFS - 14,
                                      keep_silence=500,
                                      )

            folder_name = 'audio-chunks'

            if not os.path.isdir(folder_name):
                os.mkdir(folder_name)
            whole_text = ''

            for i, audio_chunk in enumerate(chunks, start=1):
                chunk_filename = os.path.join(folder_name, f'chunk{i}.wav')
                audio_chunk.export(chunk_filename, format='wav')
                with sr.AudioFile(chunk_filename) as source:
                    audio_listened = store.record(source)
                    try:
                        text = store.recognize_google(audio_listened)
                    except:
                        print('Sorry...run again...')
                        return render(request, 'index.html', {'form': form, 'msg': 'Something wrong with conversion, please try again!'})
                    else:
                        text = f'{text.capitalize()}. '
                        print(chunk_filename, ': ', text)
                        whole_text += text
            return render(request, 'index.html', {'form': form, 'text': whole_text})
```

An example of converting .wav file to text is in Figure 12. The application can perform speech recognition of a long audio file, and handle full stops and silence in a speech. Due to pronunciation specifics, or background noise in an audio file, the conversion cannot achieve a hundred percent accurate conversion result. However, the user can modify the converted sentences directly in the text area.

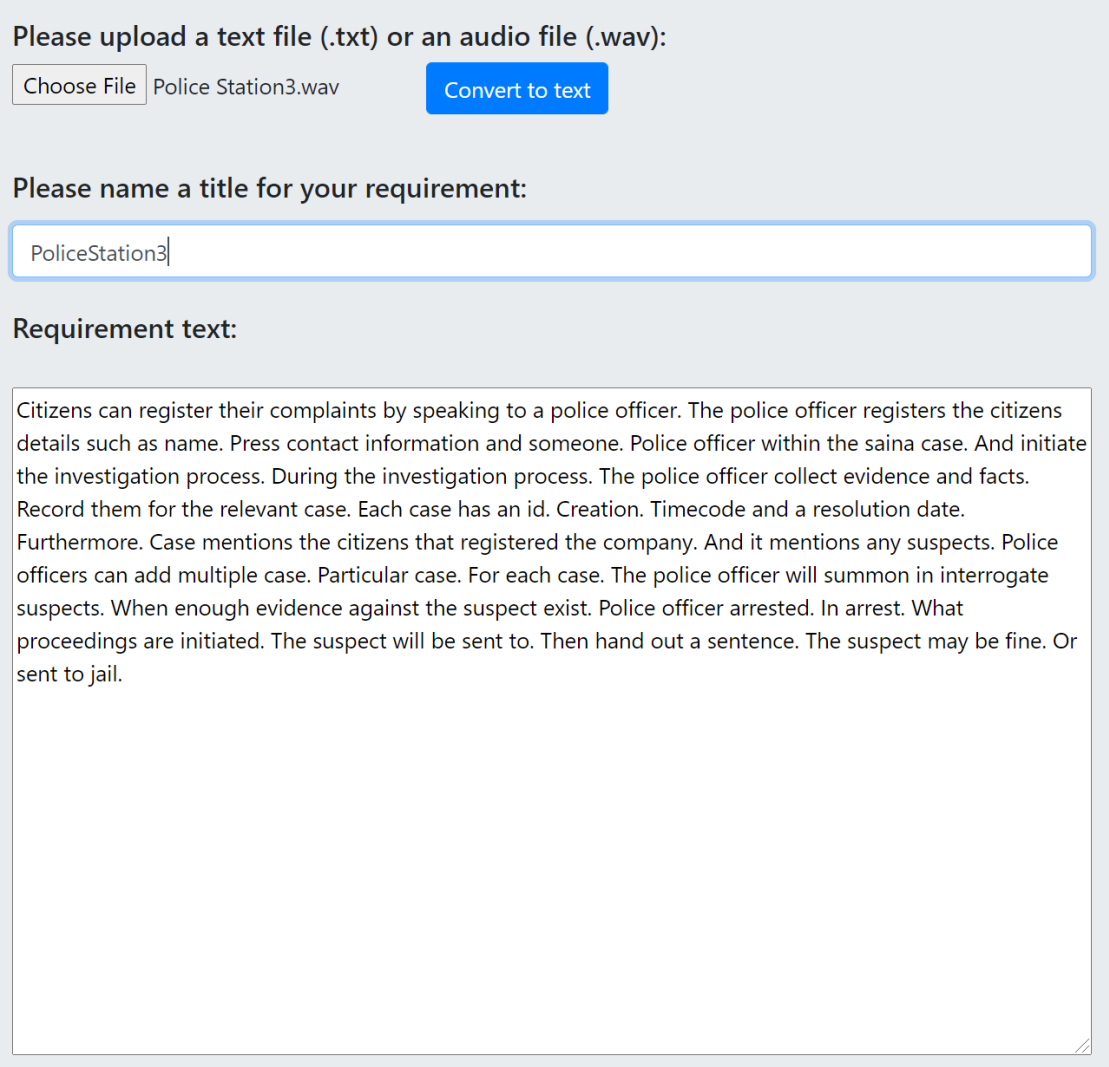


Figure 12. Example of audio file conversion

When user clicks “Show metadata” button, there is a text area following the button to display the UML class metadata result (Figure 13). The backend implements the design architecture and algorithm of the NLP based extraction module.

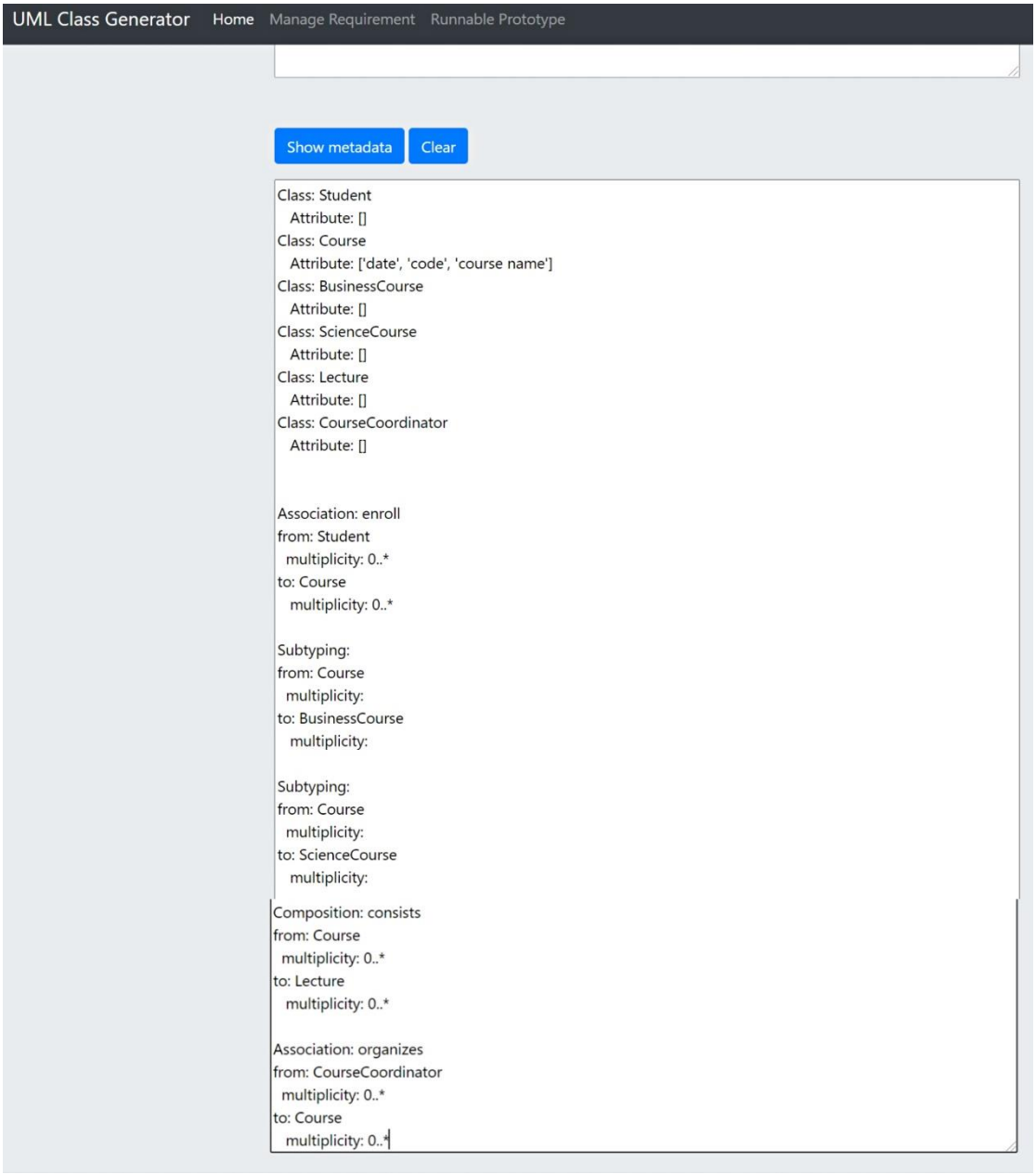


Figure 13. Example of UML class metadata result

Manage Requirement

Figure 14 shows the page for requirement modification. A select menu loads a previously saved requirement title from the database. The corresponding contents will be displayed in the text area. The user can make changes to the requirements text. This page implements the Read and Update functions in CRUD (Create, Read, Update and Delete).

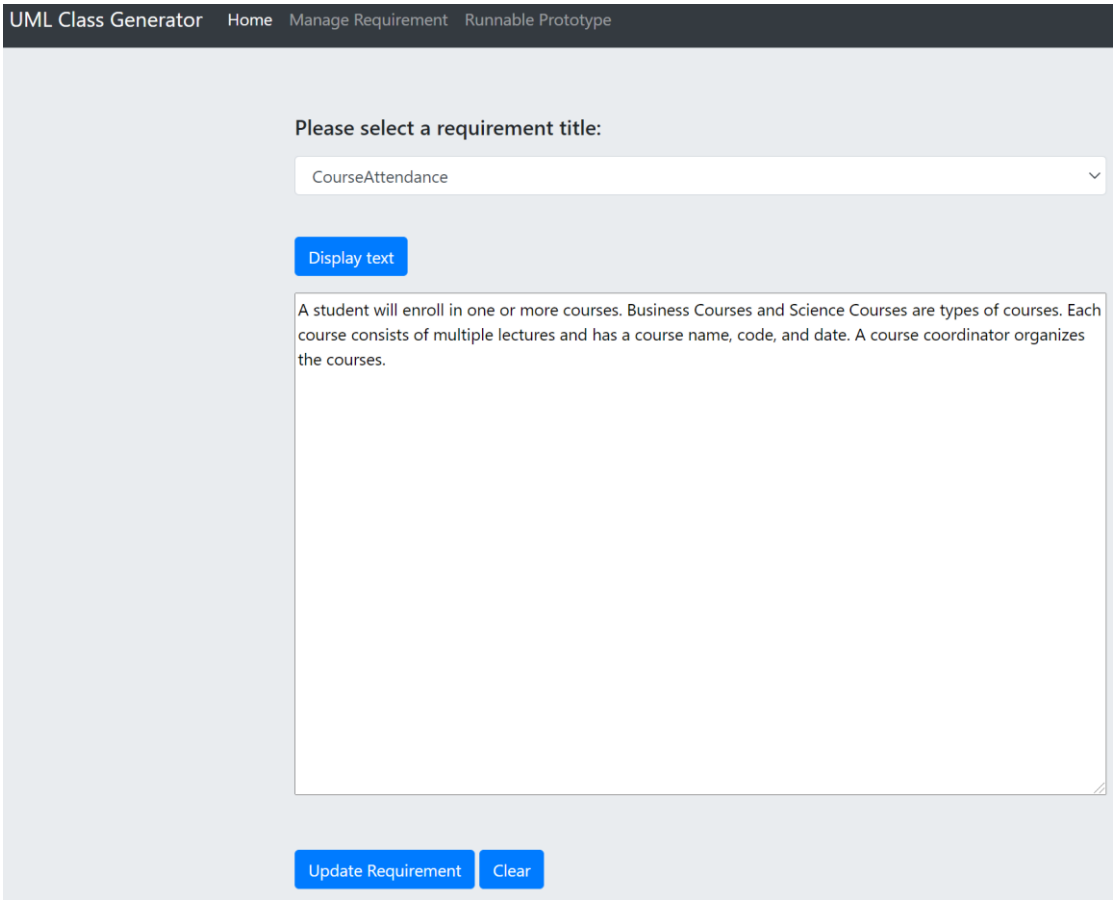


Figure 14. Manage requirement page

Runnable Prototype

This page integrates an application developed by Ralph Driessen (2020). The original purpose of the application is to execute a runnable prototype application based on the metadata of the UML class model entered by the users. Users can manually add class, attribute, operation, link relationship between classes with multiplicity, and create an application to link the required metadata. The application is activated by clicking on a “Run” button. The source code of the defined application will be created automatically.

Integration brings a win-win situation. On the one hand, for our research, end users and stakeholders can obtain an initial operational prototype from their requirements text in a rapid way. This provides them with an easy-to-understand solution for validating the generated UML class metadata in the form of an actual (prototype) application. Requirement analysts can also get faster feedback from end-users and business experts on the results of UML

class metadata. On the other hand, for the tool developed by Ralph Driessen (2020), its software (prototype) development process extends and includes the stage of requirement analysis, adding the functionality of our automated metadata generation tool - rather than manually adding UML class metadata from scratch.

Figure 15 displays a simple menu page including different parts of the models that are editable. For example, users can manually create classes by filling in class names and properties. The classes will be saved as meta data in a database.

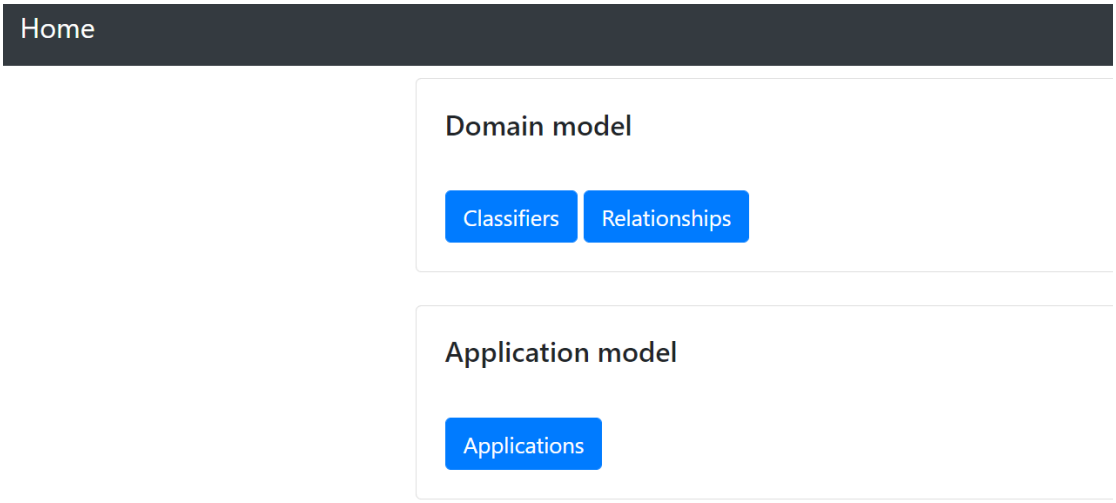


Figure 15. Menu page of runnable prototype page

To integrate with this application, the NLP generated UML class metadata is saved into the database of this application by calling its methods. A simple example of the Python integration code to achieve this meta data integration is displayed in Figure 16. All of the extracted data in the form of UML Classes, Attributes and Relationships are stored in one go, and the application can be executed after a few simple steps to define an Application Model.

```
# save into Ralph's demo
for item in clslist:
    classifier = Class(name=item)
    classifier.save()
    ClassifierGenerator(classifier).generate()

# save relation and direction into Ralph's demo
for item in sum:
    # direction
    cls_from_name = item[1][1]
    cls_to_name = item[2][1]
    # get class name
    cls_from = Classifier.objects.get(name=cls_from_name)
    cls_to = Classifier.objects.get(name=cls_to_name)

    # passing an instance rather than a variable
    relationship = Relationship(name=item[0][1], classifier_to=cls_to, classifier_from=cls_from)
    relationship.save()
    RelationshipGenerator(relationship).generate()
```

Figure 16. An Example of Integration codes

Taking the *"CourseAttendance"* from Figure 11 as an example, the corresponding metadata outputs in integrated application is shown in Figure 17.

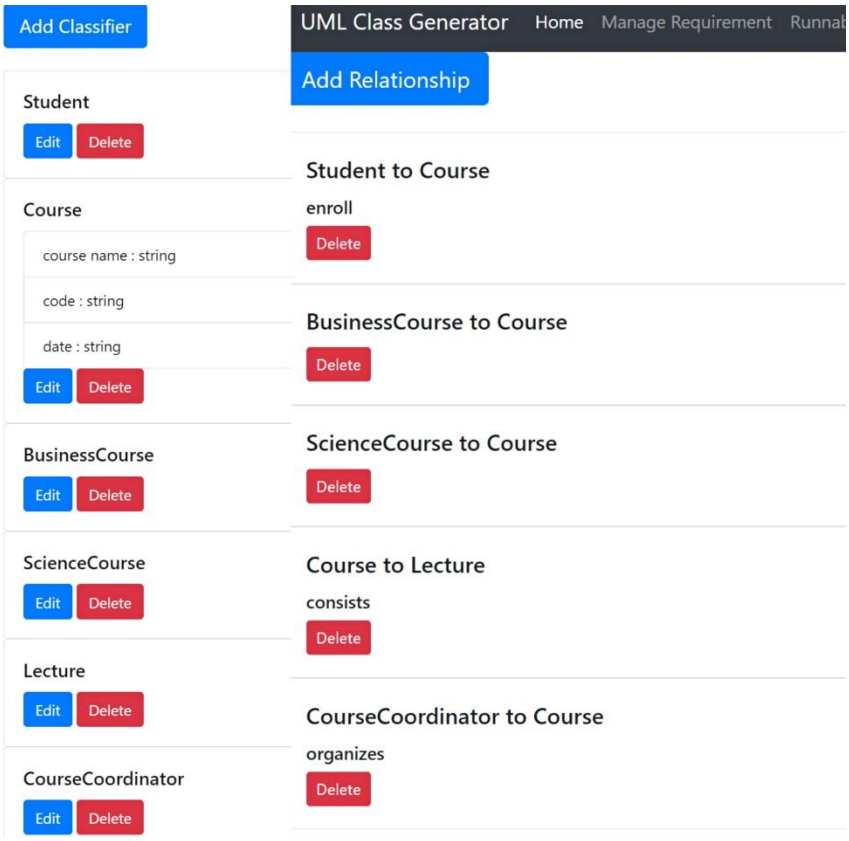


Figure 17. Results from "CourseAttendance" in Runnable Prototype

Figure 18 displays the process and an example running application when creating a new *"coursesystem"* application. We link the Class "Course" and its Attributes *"course name"*, *"code"* and *"date" to* the example application. The application will be running by clicking the "Run" button. After that, user can add any course information to the created application. The data will be saved in database. User can view detail information of their *"course list"*.

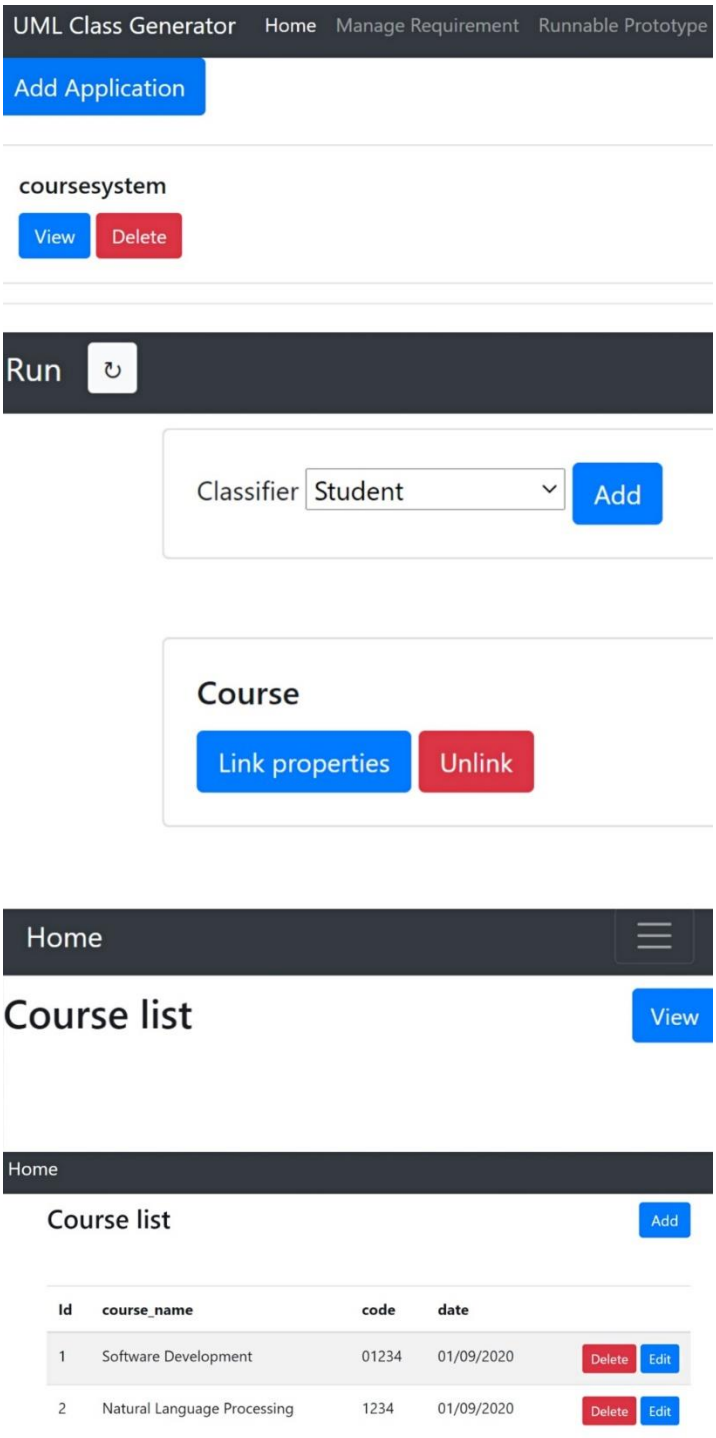


Figure 18. An Example of Executable Application

As can be seen, user can immediately obtain a runnable prototype and make changes. It is a way of providing end users with an initial prototype, enabling them evaluate the implications of their requirements, and facilitate rapid software development and model validation.

5. Example Requirements and Results

In the previous chapter we elaborated the architecture framework, extraction process, and the NLP toolkits utilised for our UML class metadata generation application, and described its implementation.

In this chapter, we will demonstrate the functionality of the tool by using a set of 5 sample textual requirements documents. For each of these, the generated UML Class meta data is shown in structured textual format, and also by means of a visual UML Class diagram.

A number of imperfections (“bugs”) are highlighted and classified. Furthermore, their causes are explained – in most cases they are the result of bugs in the underlying libraries utilised.

We conclude this chapter by discussing written feedback from an industry expert in UML tool development as to the functionality and desirability of our implementation.

5.1. Input Data Preparation

Ferrari et al (2017) published a dataset containing 79 public NL software requirement documents collected from the web, the majority of which are software requirement specifications (SRS). SRS is normally regarded as a result of the requirement engineering process after collecting requirements from stakeholders (Pekar et al, 2014). The SRS document contains the general system requirements, and specific information such as functional requirements, non-functional requirements and system design (e.g. UML diagrams, design models). However, the focus of this research is on extracting UML Class information from requirements expressed by stakeholders, rather than from well-organised documentation results from the process of collecting and analysing requirements. Therefore, instead of using

this dataset directly as input in our application, five sample requirements are compiled with reference to some paragraphs from the dataset.

5.2. “Police Station System”: Requirements and Results

This section describes the “Police Station System” requirements test case, in terms of its input and the corresponding output results. The UML class diagram has been drawn manually according to the meta data results generated in order to give an intuitive feel. Several classes of bugs have been annotated to the UML diagram, and are further explained in a section following the diagram. Four additional example requirements (and their corresponding outputs) are exhibited in appendix A.

Bug Classification and Causes

To extract UML class metadata, this program relies on two NLP toolkits: Stanford CoreNLP and NLTK. These two toolkits provide various pre-built methods to accomplish different tasks in NLP. In our project, the OpenIE (Open Information Extraction) from Stanford CoreNLP is one of the main models we use to extract triplets (Subject – Relation - Object) for each sentence. For example, *“An order is for multiple products”*. The OpenIE result is (‘Order’, ‘is for’, ‘Products’). With this triplet, we can define there are classes as “Order” and “Product”, a relationship between them named as “is for”.

We use OpenIE because it is useful when there is limited or no training data for relation extraction tasks, and it is easy to extract the information required from open domain triples. However, there are unexpected results for some particular sentences, which result in small errors emerge when identifying UML class metadata. In this case, we have characterized those errors into 4 different types:

- **Type 1: Incomplete information extraction.**

OpenIE fails to extract information from part of a sentence. An example is displayed in figure 19 that OpenIE extracts the relationship between “police

officer” and “duty sergeant”, but fail to identify and include relationship between “police officer” and “crime officer”.

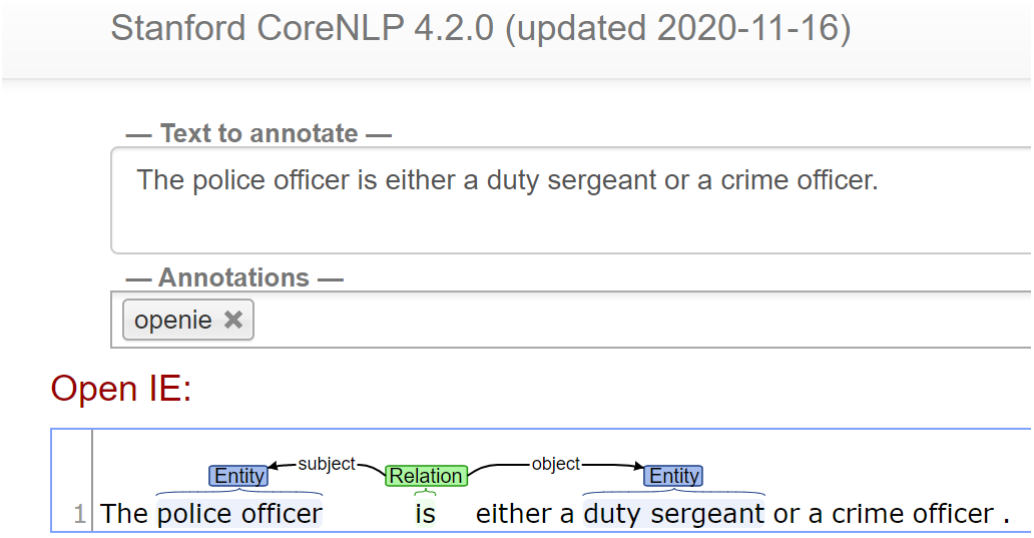


Figure 19. Example of Failure

- **Type2: Multiple entities problems.**

Three or more entities exist in one sentence, and those entities have relationships with each other. For example, “Police officer can add multiple case entries to a particular case.”, the OpenIE result is (‘police officer’, ‘add multiple case entries’, ‘case’; ‘police officer’, ‘add’, ‘case entries’)

- **Type 3: Noun + Prep + Noun problem.**

For example, “location in supermarket” will be extracted as an object

- **Others**

For example, OpenIE misunderstands the sentence, the sentence has co-reference problems, or result is correct according to the sentence, but leads to dangle problems in the diagram, etc.

The diagram will highlight the type of errors by different color and attach a specific error explanation. Every test case consists of requirement text, the output metadata from our program, and UML class diagram with error explanation (see Appendix A).

Input scenario 1:

Requirement title
Police Station System
Requirement text
“Citizens can register their complaints by speaking to a police officer. The police officer is either a duty sergeant or a crime officer. The police officer registers the citizen’s details such as name, address, contact information and so on. The police officer will then assign a case, and initiate the investigation process. *During the investigation process, the police officer collects evidence and facts, and records them for the relevant case. Each case has an id, a creation date, a crime code, and a resolution date. Furthermore, a case mentions the citizen that registered the complaint, and it mentions any suspects. Police officers can add multiple case entries to a particular case.* *For each case, the police officer will summon and interrogate suspects. When enough evidence against a suspect exists, a police officer will arrest the suspect. Following an arrest, court proceedings are initiated, and the suspect will be sent to a court. The court will then hand out a sentence, and the suspect may be fined, or sent to jail. ”*

The following is the intermediate output (i.e, triplets result after Text Structuring Process) based on the above requirements texts:

[('citizens', 'speaking to', 'police officer'), ('citizens', 'can register', 'complaints'), ('police officer', 'is', 'duty sergeant'), ('police officer', 'registers', 'citizen details'), ('police officer', 'initiate', 'investigation process'), ('police officer', 'assign', 'case'), ('police officer', 'collects', 'evidence'), ('police officer', 'collects', 'facts'), ('case', 'has', 'resolution date'), ('case', 'has', 'id'), ('case', 'has', 'creation date'), ('case', 'has', 'crime code'), ('case', 'mentions', 'citizen'), ('case', 'mentions', 'suspects'), ('police officers', 'can add case entries to', 'case'), ('police officers',

'can add', 'case entries'), ('police officer', 'summon', 'case'), ('police officer', 'interrogate', 'suspects'), ('police officer', 'arrest', 'suspect'), ('court proceedings', 'are initiated following', 'arrest'), ('court', 'hand out', 'sentence'), ('suspect', 'sent to', 'jail')]

The UML class metadata output generated by our NLP tool:

```
Class: Citizen
   Attribute: []
Class: PoliceOfficer
   Attribute: []
Class: Complaint
   Attribute: []
Class: DutySergeant
   Attribute: []
Class: CitizenDetail
   Attribute: []
Class: InvestigationProcess
   Attribute: []
Class: Case
   Attribute: ['resolution date', 'id', 'creation date', 'crime code']
Class: Evidence
   Attribute: []
Class: Fact
   Attribute: []
Class: Suspect
   Attribute: []
Class: CaseEntry
   Attribute: []
Class: CourtProceeding
   Attribute: []
Class: Arrest
   Attribute: []
Class: Court
   Attribute: []
Class: Sentence
   Attribute: []
Class: Jail
   Attribute: []
```

Association: speaking to
from: Citizen
  multiplicity: 0..*
to: PoliceOfficer
   multiplicity: 0..*

Association: can register
from: Citizen
  multiplicity: 0..*
to: Complaint
   multiplicity: 0..*

Subtyping:
from: PoliceOfficer
  multiplicity:
to: DutySergeant
   multiplicity:

Association: registers
from: PoliceOfficer
  multiplicity: 0..*
to: CitizenDetail
   multiplicity: 0..*

Association: initiate
from: PoliceOfficer
  multiplicity: 0..*
to: InvestigationProcess
   multiplicity: 0..*

Association: assign
from: PoliceOfficer
  multiplicity: 0..*
to: Case
   multiplicity: 0..*

Association: collects
from: PoliceOfficer
  multiplicity: 0..*
to: Evidence
   multiplicity: 0..*

Association: collects
from: PoliceOfficer
  multiplicity: 0..*
to: Fact
   multiplicity: 0..*

Association: mentions
from: Case
  multiplicity: 0..*
to: Citizen
   multiplicity: 0..*

Association: mentions
from: Case
  multiplicity: 0..*
to: Suspect
   multiplicity: 0..*

Association: can add
from: PoliceOfficer
  multiplicity: 0..*
to: CaseEntry
   multiplicity: 0..*

Association: can add case entries to
from: PoliceOfficer
  multiplicity: 0..*
to: Case
   multiplicity: 0..*

Association: summon
from: PoliceOfficer
  multiplicity: 0..*
to: Case
   multiplicity: 0..*

Association: interrogate
from: PoliceOfficer
  multiplicity: 0..*
to: Suspect
   multiplicity: 0..*

Association: arrest
from: PoliceOfficer
  multiplicity: 0..*
to: Suspect
   multiplicity: 0..*

Association: are initiated following
from: CourtProceeding
  multiplicity: 0..*
to: Arrest
   multiplicity: 0..*

Association: sent to
from: Suspect
  multiplicity: 0..*
to: Court
   multiplicity: 0..*

Association: hand out
from: Court
  multiplicity: 0..*
to: Sentence
   multiplicity: 0..*

Association: sent to
from: Suspect
  multiplicity: 0..*
to: Jail
   multiplicity: 0..*

Corresponding UML class diagram

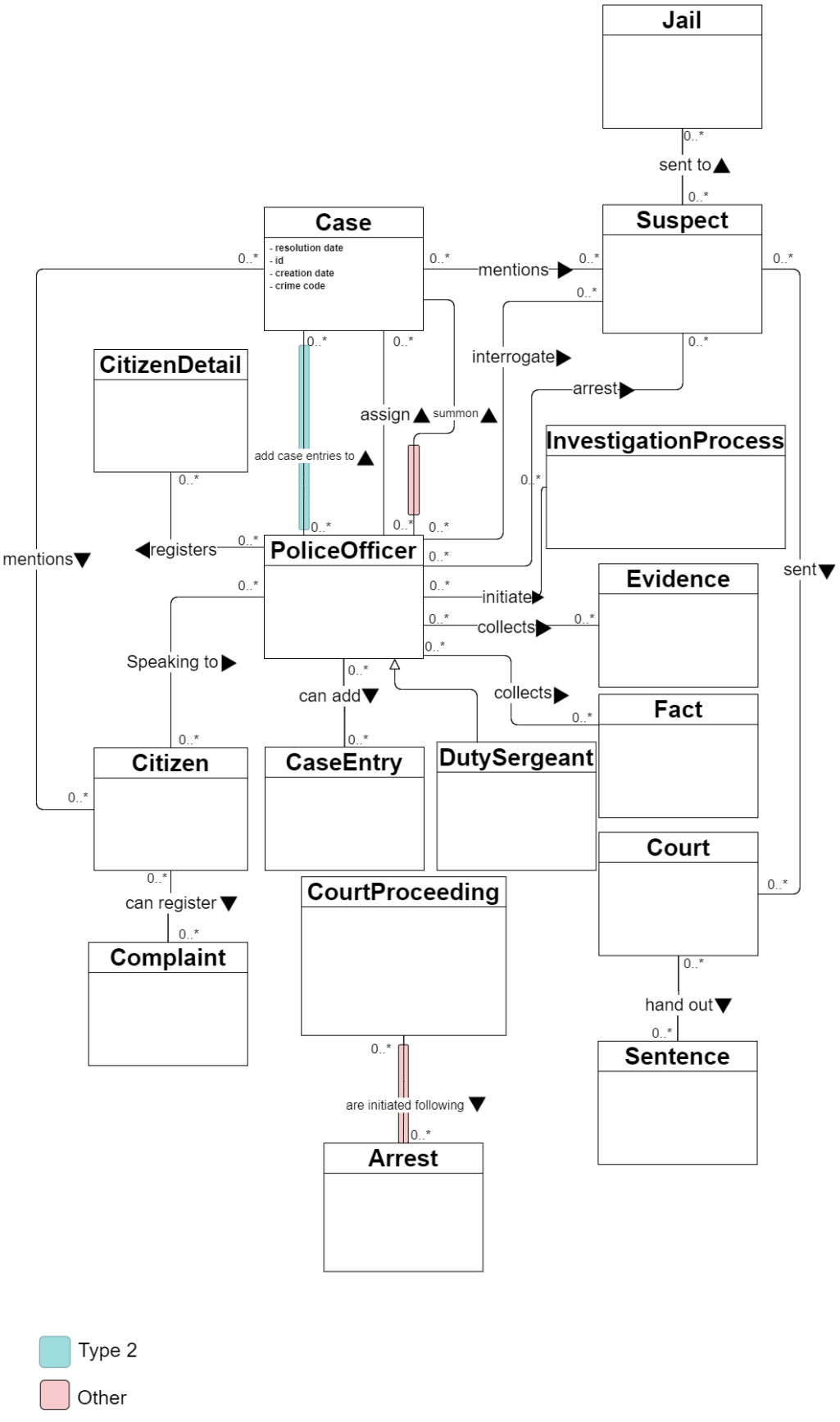


Figure 20. UML class diagram from police station system

Error explanation:

Type 1: Incomplete information extraction (OpenIE fails to extract information from part of a sentence)

- “The police officer is either a duty sergeant or a crime officer.” Triplets: [('police officer', 'is', 'duty sergeant')] OpenIE fails to extract “police officer is a crime officer”.

  (**Note:** OpenIE has incomplete information extraction problem when processing the sentence containing the words: “can be”, “such as”, “is”, “are”, etc, which sometimes indicates subtyping relations. If the sentence contains more than one object, OpenIE will extract the first object appeared in the sentence.)
- “The police officer registers the citizen’s details such as name, address, contact information and so on.” Triplets: [('police officer', 'registers', 'citizen details')] The OpenIE fails to extract information from the latter part of the sentence.

Type 2: Multiple entities problem (Three or more entities exist in one sentence, and those entities have relationships with each other)

- “Police officer can add multiple case entries to a particular case.” Triplets: [**('police officers', 'add case entries to', 'case')**, ('police officers', 'add', 'case entries')] The ideal result is (‘case entry’, ‘is for’, ‘case’) or (‘case’, ‘has’, ‘case entry’) accompany (‘police officers’, ‘add’, ‘case entries’)

Others (understanding problem and dangle problem)

- “For each case, the police officer will summon and interrogate suspects.” Triplets: [**('police officer', 'summon for', 'case')**, ('police officer', 'interrogate', 'suspects')] OpenIE separate the sentence as “for each case, the police will summon”; “The police officer will interrogate suspects” to process.
- “Following an arrest, court proceedings are initiated.” Triplets: **[('court proceedings', 'are initiated following', 'arrest')]** It is not wrong according to the sentence but results in dangle problem in the UML class diagram.

5.3. Discussion of UML Experts Verification Results

The results of sample requirements texts, a screen-recorded video of our application, and a questionnaire are delivered to twenty UML experts (from industry and academia) as a verification and evaluation session. In the questionnaire, the UML experts were asked, more specifically, to give a score to the accuracy of UML class metadata results, give feedback on our tool, and their experience and opinion on NLP-based UML model generation tools. The design of questionnaire is exhibited in Appendix B. Unfortunately, due to the recipients being busy, it resulted in only 2 responses.

As explained earlier, our results point out and explain some of the errors in the extraction results from the use of Stanford OpenIE package. To begin with, respondents were asked to rate the results from two different perspectives. When disregarding the highlighted errors, the two evaluators gave a score of 5 and 3 respectively (10 as maximum score). When considering that the situation and error was highlighted and explained, both of them gave 7 for the accuracy of the results, and affirmed that our tool can assist the requirement analysis process. One of the evaluators specified that typically, the first goal in elicitation process is to seek an 80/20 gain. The accuracy of our results has not yet achieved 80% of accuracy from his perspective because of dependent libraries. However, the other respondent stressed that even with the bugs remained, using our tool is a good way to ramp up a design rapidly.

Evaluators regarded our textual extraction results easy to read and modify. While an evaluator embraced the opinion that textual results can be easy to manipulate and copy-paste, and thus reducing the time on dealing with layout, the other raised that visual representation is a benchmark that has been established in this particular UML class modelling field. Therefore, the diagram generation module should be added as a feature in this tool to meet this requirement.

As for the integrated work (“Runnable Prototype” page), the evaluators gave positive attitude to its’ possibility on getting end users more involved during

requirement analysis process. This integration effort is regarded as a way to check the understanding of requirements, correct, qualify and extend the requirements.

The questionnaire also received suggestions on the improvement of this tool in the future, which are displayed as follows:

- Adding a machine learning feedback loop to improve the precision.
- Even if without machine learning applied, the tool should be improved in a way of maintaining synchronization between text and model results, particularly when users modify the results. This means that the metadata results can be modified by users and make changes on the original requirements texts.
- Parsing problems should be handled, and subtype and instance should be distinguished. Therefore, more efforts on improving the extraction methodology should be made in order to cover more elements identification.
- The tool cannot detect synonyms and other terminology defined in the text. It would be better for the tool to generate a general description of terminology of the extracted domain. To achieve this, ontology induction would be valuable to this tool.

Neither evaluator had access to or used an NLP-based UML model generation tool, but we received some of their ideas and requests for such a tool. While one of the respondents claim to reduce drawing time, another UML expert desire a characteristic of interactivity for such a tool. A Siri-like conversation tool that generating and clarifying qualified questions to verify and correct the requirements when they are emerged would be preferred. This can be achieved to develop a chatbot that addressing the issue of input requirement regulations, and a validation process during the conversation, thus to generate a precise result from user’s requirement. However, requirement analysis process should co-ordinate various perspectives from different stakeholders, a chatbot is more specific to an individual. In this case, the involvement of a wider range of stakeholders remains an issue.

6. Conclusion

This thesis presents a new architecture framework from the perspective of assisting software requirements analysis process by combining the use of rules and NLP tools in a different manner. Other studies have typically limited the user's requirements input text to a significant extent, indicating that the steps of text cleaning and structuring are left to humans, and the analysis process of those tools are not fully automated. The tool we have developed replaces this step by utilizing the NLP toolkits, i.e., transforming the requirements text into structured triplet text data, improving the efficiency on text structuring from raw requirements text.

Furthermore, previous research tended to develop extraction tools for UML experts or requirements analysts as users. Such tools did increase the speed of requirements analysis to some extent, but created a separation of communication with the main owners of the requirements – the stakeholders –, and thus in responding to changes in requirements and validation feedback. Software nowadays needs to evolve rapidly, in which analysts are required to interview and communicate with all stakeholders of the software, and adapt to their changes and feedback immediately. Under this circumstance, instead of reorganising the collected requirements themselves and analysing them with a tool after communication, this research integrates with a run-time application solution developed by Driessen (2020). The integrated application allows the analysts to immediately acquire UML class metadata results, and a run-time prototype based on them, immediately after stakeholders express their requirements. The stakeholders can view the initial prototype from their requirements and modify it.

6.1. Limitations

While the framework developed provides an improved solution for extracting UML class metadata from requirements texts and the integration effort offers a

possibility for all stakeholders involved in the requirement analysis process, there are few things to keep in mind. Firstly, the sample of requirements text data (Chapter 5.2 & Appendix A) on which these extraction results are based is quite small. A larger sample of realistic requirements data, perhaps spread across various software development organizations, is necessary to expand the test cases for this tool. This expansion could also allow for greater extraction mechanisms to be developed or evolved, particularly as software requirements expressed in NL are complex and most of the time contain a number of dedicated words or specific abbreviations. Secondly, the mechanism for extracting multiplicity become difficult in our frameworks since the structured triplet data contains three arguments, while the multiplicity is usually expressed in adjective or numeral words, which were neglected in triplet's extraction. Thirdly, the only methodology of attribution extraction in this research is restricted to a set of attribute words, and the mechanism cannot extend the attribute glossary or Rules by catching or learning from user input. Fourthly, while our tool can extract less structured requirements texts (the tool uses NLP toolkits to process text structuring), suggestively, the tool becomes less effective when processing over-complex sentences or compound sentences (i.e, contains reference words like "which"). Lastly, the questionnaire produced for the evaluation of the results and tools collected very limited responses. Also, the only participants in the questionnaire were UML experts or requirements engineers. More stakeholders need to be included to assess our tool.

6.2. Future Work

Due to the limitations of requirements data, the extraction methodology in this research forgoes any machine learning or deep learning techniques, opting instead for an approach that uses NLP tools and rules. An aspect of further research from this might be training models for relationship and multiplicity prediction, or conducting supervised learning for ontology induction once sufficient realistic software requirements data are obtained. This can provide a more intelligent solution for this research field.

As mentioned in chapter 5, the Stanford OpenIE package we used in this tool failed to extract triplets from incomplete sentences. In this case, future works could explore other rising NLP toolkits, such as TensorFlow-based Google SyntaxNet, perhaps able to deal with more free text, or compare NLP toolkits used in the development of similar extraction tools.

Furthermore, chapter 5 also mentions an extensibility problem in that the mechanism cannot update the glossary or create Rules based on user modifications. This indicates a data driven implementation in the future research work. For example, the extracted UML Class meta data should update when user changes Multiplicity, Relationship, Class value, or add a new Class name, new attribute in prototype application. Besides that, once the extracted UML Class metadata is updated, is it possible to update or renew the corresponding requirements text at the same time? A two-way integration and generation should be worked out in the future.

In addition, the integration effort offered the possibility to involve all the stakeholders in requirement analysis process and these requirements owners were supposed to evaluate the results. However, they were not fully included in our verification questionnaire. UML experts were the only group of users who participated in this questionnaire and evaluated the results by giving a score. What could not be confirmed was the evaluation criteria they used to give this score, in other words, the evaluation indicators were in a black box. These in turn raise some further research questions, e.g., what are the metrics used to evaluate the accuracy of the extraction results? How do other stakeholders (end-users, business specialists, software developer, etc.) validate the results? Is there a framework to reconcile these judgements from different stakeholders' perspectives? Our evaluation session is set up by sending results and a screen video. However, the ideal process would be to conduct task-based evaluations by providing a tool that the evaluator can play around in practice. In this case, ensure consistency of input is required and evaluators should be provided with input texts of similar length and similar complexity. Then, a series of task-based activities and questions can be undertaken. This was not achieved in our study due to the time-consuming

nature of finding and contacting candidate evaluators, and such evaluation sessions (particularly in the absence of relevant social networks in industry and the uncertainty of response time), but could be carried out in the future studies.

Another interesting research angle might be extending our tool to automate the generation of UML class diagrams, as our methodology focuses on extracting UML class metadata. In our questionnaire, there is a respondent who also mentioned that some people prefer a more intuitive or visual model (i.e., a diagram). Based on this research tool, a diagram editor can be developed to meet the demands of this group of users.

7. References

1. Driessen, R. (2020). UML Class Models as First-Class Citizen: Metadata at Design-time and Run-time. *Leiden University. Leiden Institute of Advanced Computer Science (LIACS)*. Pp 1-42.

2. Berzins, V., Martell, C., Luqi., Adams, P. (2008). Innovations in Natural Language Document Processing for Requirements Engineering. In: Paech B., Martell C. (eds) Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs. Monterey Workshop 2007. Lecture Notes in Computer Science, vol 5320. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-89778-1_11

3. Westland, J, C. (2002). The cost of errors in software development: evidence from industry. *Journal of Systems and Software*. 62(1). pp 1-9. https://www.sciencedirect.com/science/article/pii/S0164121201001303

4. Kof, L. (2005). Natural Language Processing: Mature Enough for Requirements Documents Analysis?. In: Montoyo A., Munoz R., Merais E. (eds) Natural Language Processing and Information Systems. NLDB 2005. Lecture Notes in Computer Science, vol 3513. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11428817_9.

5. Boehm, B. & Basili, V, R. (2001). Software defect reduction top 10 list. *IEEE Computer*. 34(1). pp. 135-137 https://www.researchgate.net/publication/220476082_Software_Defect_Reduction_Top_10_List

6. Gupta, A, K. & Deraman, A. (2019). A Framework for Software Requirement Ambiguity Avoidance. *International Journal of Electrical and Computer Engineering (IJECE) 9(6).* Pp.5436-5445.

https://www.researchgate.net/publication/337664522_A_framework_for_software_requirement_ambiguity_avoidance

7. Agarwal, R. & Sinha, A, P. (2003). Object-Oriented Modeling with UML: A Study of Developers' Perceptions. *Communications of the ACM.* 46(9). Pp.248-256. https://www.researchgate.net/publication/220424961_Object-oriented_modeling_with_UML_A_study_of_developers'_perceptions

8. Narawita, C, R. & Vidanage, K. (2016). UML Generator – An Automated System for Model Driven Development. *International Conference on Advances in ICT for Emerging Regions (ICTer)*. Pp. 250-256. https://www.researchgate.net/publication/312964481_UML_generator_-_an_automated_system_for_model_driven_development

9. Deshpande, D. (2012). Textual Requirement Analysis for Object Model Designing by Using NLP. *International Journal of Innovative Research in Science, Engineering and Technology*. 1(2). Pp. 270-276. https://www.ijirset.com/upload/december/23_Textual.pdf

10. Arellano, A., Carney, E. & Austin, M, A. (2015). Natural Language Processing of Textual Requirements. *The Thenth International Conference on Systems*. Pp. 93-97. https://www.researchgate.net/publication/290225986_Frameworks_for_Natural_Language_Processing_of_Textual_Requirements

11. Osman, C, C. & Zălhan, P, G. (2016). From Natural Language Text to Visual Models: A Survey of Issues and Approaches. *Informatica Economică. 20(4).* Pp. 44-61. https://www.researchgate.net/publication/311966768_From_Natural_Language_Text_to_Visual_Models_A_survey_of_Issues_and_Approaches

12. Hevner, A, R., March, S, T., Park, J. & Ram, S. (2004). Design Science in Information System Research. MIS Quarterly. 28(1). Pp. 75-105. https://www.researchgate.net/publication/201168946_Design_Science_in_Information_Systems_Research

13. Fowler, M. (2003). UML Disstilled 3rd Edition: A Brief Guide to the Standard Object Modeling Language. Pp.14-25 & pp. 35-46. http://ce.sharif.edu/courses/96-97/2/ce418-1/resources/root/Books/UMLDistilled.pdf [Accessed date: 05-03-2020]

14. Hamilton, K. & Miles, R. (2006). Learning UML 2.0. Pp. 22 & Pp. 89-125. https://flatis.moe/uploads/uploads/uml.pdf [Accessed date: 05-03-2020]

15. Joseph, S, R., Hlimani, H., Letsholo, K., Kaniwa, F., & Sedimo, K. (2016). Natural Language Processing: A Review. *International Journal of Research in Engineering and Applied Sciences*. 6(3). Pp. 207-210. https://www.researchgate.net/publication/309210149_Natural_Language_Processing_A_Review [Accessed date: 03-09-2020]

16. Jones, K, S. (2001). Natural Language Processing: A Historical Review. *University of Cambridge*. Pp. 2-10. https://www.cl.cam.ac.uk/archive/ksj21/histdw4.pdf [Accessed date: 03-09-2020]

17. Dawood, O.S., Sahraoui, A.E.K. (2017). From Requirements Engineering to UML Using Natural Language Processing – Survey Study. European Journal of Industrial Engineering. 2(1). pp. 44-50. doi:10.24018/ejers.2017.2.1.236. https://www.researchgate.net/publication/314486237_From_Requirements_Engineering_to_UML_using_Natural_Language_Processing_-_Survey_Study

18. Yalla, P., Sharma, N. (2015) Integrating Natural Language Processing and Software Engineering. International Journal of Software Engineering and Its Applications. 9(11). Pp. 127-136. https://www.researchgate.net/publication/292299148_Integrating_Natural_Language_Processing_and_Software_Engineering

19. Koerner, S, J., Landhäußer, M., Tichy, W. (2014). From Requirements to UML Models and Back: How Automatic Processing of Text can Support Requirements Engineering. *Software Qual J.* (22). Pp 121-149. https://www.researchgate.net/publication/257665306_From_requirements_to_UML_models_and_back_How_automatic_processing_of_text_can_support_requirements_engineering

20. Sarawagi, S. (2007). Information Extraction. Foundation and Trends in Databases. 1(3). Pp 261-377. https://books.google.nl/books?id=AqHpDoYPjLQC&printsec=frontcover#v=onepage&q&f=false

21. Cheng X., Kong X., Liao L., Li B. (2020). A Combined Method for Usage of NLP Libraries Towards Analyzing Software Documents. In: Dustdar S., Yu E., Salinesi C., Rieu D., Pant V. (eds) *Advanced Information Systems Engineering*. CAiSE 2020. Lecture Notes in Computer Science, vol 12127. Springer, Cham. https://doi.org/10.1007/978-3-030-49435-3_32

22. Steven, B., Loper, E., Klein, E. (2009). Natural Language Processing with Python. O'Reilly Media Inc. https://www.researchgate.net/publication/220691633_Natural_Language_Processing_with_Python

23. Azzouz, Z.B., Karaa, W.B.A., Singh, A., Dey, N., Ashour, A.S., Ghezala, H.B. (2015). Automatic Builder of Class Diagram (ABCD): An Application of UML Generation From Functional Requirements. Software Practice and Experience, 46(12), pp. 1443-1458. doi: 10.1002/spe.2384

https://www.researchgate.net/publication/283571085_Automatic_Builder_of_Class_Diagram_ABCD_an_Application_of_UML_Generation_From_Functional_Requirements

24. Overmyer, S.P., Benoit, L., Owen, R. (2001). Conceptual Modeling through Linguistic Analysis Using LIDA. Proceedings of the 23rd International Conference on Software Engineering (ICSE), pp. 401-410. doi: 10.1109/ICSE.2001.919113. https://www.semanticscholar.org/paper/Conceptual-modeling-through-linguistic-analysis-Overmyer-Lavoie/e43ee1a276c3b6c1f13f1942e49026058718fc63

25. Barker, R. (1990). Case Method: Entity Relationship Modelling. Addison-Wesley Professional, ISBN 10: 0201416964.

26. Chen, P.P. (1983). English Sentence Structure and Entity-Relationship Diagrams. Information Sciences, 29(2-3), pp. 127-149. doi: https://doi.org/10.1016/0020-0255(83)90014-2

27. Perez-Gonzalez, H, G., Kalita, J, K. (2002) Automatically Generating Object Models from Natural Language Analysis. Proceedings of the 17th Annual ACM SIGPLAN Conference on Object-oriented Programming, System, Languages, and Applications (OOPSLA' 02). pp. 86-87.

28. Harmain, H, M., Gaizauskas, R. (2003) CM-Builder: A Natural Language-based CASE Tool for Object-oriented Analysis. Automated Software Engineering, 10(2), pp. 157-191. doi: 10.1023/A:1022916028950 https://www.researchgate.net/publication/226432934_CM-Builder_A_natural_language-based_CASE_tool_for_object-oriented_analysis

29. Herchi, H., Abdessalem, W, B. (2012) From User Requirements to UML Class Diagram. Proceedings of International Conference on Computer Related Knowledge (ICCRK' 2012). Sousse, Tunisia.

https://www.researchgate.net/publication/232808848_From_user_requirements_to_UML_class_diagram

30. More, P., Phalnikar, R. (2012) Generating UML Diagrams from Natural Language Specifications. International Journal of Applied Information Systems. 1(8). pp. 19-23. doi: 10.5120/ijais12-450222. https://www.researchgate.net/publication/258650012_Generating_UML_Diagrams_from_Natural_Language_Specifications

31. Dewar, R, G., Li, K., Pooley, R, J. (2005). Object-Oriented Analysis Using Natural Language Processing. Linguistic Analysis. https://www.semanticscholar.org/paper/Object-oriented-Analysis-Using-Natural-Language-Dewar-Li/e9e464dc4bf52d93adadb140c410a950ed04aea9

32. Palmer, D. 2010. Handbook of Natural Language Processing. Chapter 2. Text Preprocessing. pp. 9-28. https://karczmarczuk.users.greyc.fr/TEACH/TAL/Doc/Handbook%20Of%20Natural%20Language%20Processing,%20Second%20Edition%20Chapman%20&%20Hall%20Crc%20Machine%20Learning%20&%20Pattern%20Recognition%202010.pdf

33. Palmer, D. 2000. Handbook of Natural Language Processing. Chapter 2. Tokenisation and sentence segmentation. pp. 11. https://books.google.nl/books?hl=en&lr=&id=VoOLvxyX0BUC&oi=fnd&pg=PA11&ots=ww82HJ3Ot-&sig=5CyuoDHxUHKxKd2JiXUjIT_hSvY&redir_esc=y#v=onepage&q&f=false

34. Camacho-Collados, J., Pilehvar, M, T. 2018. On the Role of Text Preprocessing in Neural Network Architectures: An Evaluation Study on Text Categorization and Sentiment Analysis. *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. pp. 40-46. Association for Computational Linguistics.

https://www.aclweb.org/anthology/W18-5406.pdf

35. Hay, D, C. 2006. Data Model Patterns: A Metadata Map. Chapter 1. About metadata models. pp.1- 7. https://books.google.nl/books?hl=en&lr=&id=YxDBaWj9itkC&oi=fnd&pg=PP1&dq=Data+Model+Patterns:+A+Metadata+Map+&ots=PkZoK1FcwI&sig=aHk1BnrzPm4MBZ49uqLUQ4BkZyU&redir_esc=y#v=onepage&q=Data%20Model%20Patterns%3A%20A%20Metadata%20Map&f=false

36. Singh, S. 2018. Natural Language Processing for Information Extraction. arXiv:1807.02383 [cs.CL]. pp.1-24. https://arxiv.org/abs/1807.02383v1

37. Khairova N., Petrasova S., Mamyrbayev O., Mukhsina K. (2020) Open Information Extraction as Additional Source for Kazakh Ontology Generation. In: Nguyen N., Jearanaitanakij K., Selamat A., Trawiński B., Chittayasothorn S. (eds) Intelligent Information and Database Systems. ACIIDS 2020. Lecture Notes in Computer Science, vol 12033. Springer, Cham. pp. 86-96. https://doi.org/10.1007/978-3-030-41964-6_8

38. Chikkamath, M., Ponnalagu, K., Prasad, P, V, R, D. & Veera, P, R, M. 2018. Extracting Conjunction Patterns in Relation Triplets from Complex Requirement Sentence. I*nternational Journal of Computer Trends and Technology (IJCTT)*. 60(3). pp. 133-143. https://www.researchgate.net/publication/327934144_Extracting_Conjunction_Patterns_in_Relation_Triplets_from_Complex_Requirement_Sentence

39. Pinto, A., Oliveira, H, G. & Alves, A, O. 2016. Comparing the Performance of Different NLP Toolkits in Formal and Social Media Text. https://drops.dagstuhl.de/opus/volltexte/2016/6008/

40. Btoush, E, S. & Hammad, M, M. (2015). Generating ER Diagrams from Requirement Specifications Based on Natural Language Processing. *International Journal of Database Theory and Application*. 8(2). pp 61-70. https://www.researchgate.net/publication/275952818_Generating_ER_Diagrams_from_Requirement_Specifications_Based_On_Natural_Language_Processing

41. Grosan C., Abraham A. (2011). Rule-Based Expert Systems. In: *Intelligent Systems*. Intelligent Systems Reference Library, vol 17. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-21004-4_7

42. Ambler, S. (2004). The Object Primer: Agile Model-Driven Development with UML 2.0 (3rd ed.). Cambridge: Cambridge University Press. doi: 10.1017/CBO9780511584077

43. Simões, G., Galhardas, H., & Coheur, L. (2009). Information Extraction tasks: a survey. https://www.semanticscholar.org/paper/Information-Extraction-tasks-%3A-a-survey-Sim%C3%B5es-Galhardas/92810be04dc7ecb5c7659e5b925bda2733baee11

44. Yildiz, B., & Miksch, Silvia. (2008). Motivating ontology-driven information extraction. https://www.researchgate.net/publication/228675059_S_Motivating_ontology-driven_information_extraction

45. Poon, H., & Domingos, P. (2010). Unsupervised Ontology Induction From Text. Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics. Pp. 296 – 305. https://www.aclweb.org/anthology/P10-1031.

46. Angeli, G., Premkumar, M, J., & Manning, C, D. (2015). Leveraging Linguistic Structure For Open Domain Information Extraction. In: Processings of the Association of Computational Linguistics (ACL). Pp.

344-354. Doi: 10.3115/v1/P15-1034. https://nlp.stanford.edu/software/openie.html

47. Banko, M., Cafarella, M, J., Soderland, S., Broadhead, M., & Etzioni, O. (2007). Open Information Extraction from the web. IJCAI'07: Proceedings of the 20th international joint conference on artificial intelligence. Pp. 2670-2676. https://dl.acm.org/doi/10.5555/1625275.1625705

48. Ali, S., Mousa, H., & Hussein, M. (2019). A Review of Open Information Extraction Techniques. IJCI. International Journal of Computers and Information. Pp. 20-28. Doi: 10.21608/ijci.2019.35099. https://www.researchgate.net/publication/334298170_A_Review_of_Open_Information_Extraction_Techniques

49. Pekar, V. Felderer, M., & Breu, R. (2014). Improvement Methods for Software Requirement Specifications: A Mapping Study. Proceedings – 2014 9th International Conference on the Quality of Information and Communications Technolog, QUATIC 2014. Pp. 242 – 245. Doi: 10.1109/QUATIC.2014.40. https://www.researchgate.net/publication/289094810_Improvement_Methods_for_Software_Requirement_Specifications_A_Mapping_Study

50. Ferrari, A., Spagnolo, G., & Gnesi, S. (2017). PURE: A Dataset of Public Requirements Documents. In 2017 IEEE 25th International Rquirements Engineering Conerence (RE). Pp. 502 – 505. Doi: 10.1109/RE.2017.29. https://www.researchgate.net/publication/320028192_PURE_A_Dataset_of_Public_Requirements_Documents

51. Waltl, B., Bonczek, G., & Matthes, F. (2018). Rule-based Information Extraction: Advantages, Limitations, and Perspectives. file:///C:/Users/surface/Downloads/Wa18b.pdf

52. Roh, Y., Heo, G., & Whang, S, E. (2019). A Survey on Data Collection for Machine Learning: A Big Data – AI Integration Perspective. In IEEE Transactions on Knowledge and Data Engineering. Doi:

10.1109/TKDE.2019.2946162.
https://ieeexplore.ieee.org/document/8862913

Appendix A: Sample Requirement Data and Results

A.1. Input Scenario 2 (Supermarket System)

Requirement title
Supermarket System
Requirement text
“The supermarket is organized into aisles. Each aisle contains various product categories. Food, drinks, sanitary items, and cleaning products are all product categories. Each product category contains many products. These products are stacked into shelves. Each product has a name, code, and location in the supermarket. *A customer enters the supermarket, and picks up a shopping cart. The desired products will be placed in the shopping cart. The customer will then go to the cashier desk, and pay for the products. Each cashier desk has a cashier, a scanner, and a till. After receiving payment from the customer, cashiers will hand them a receipt.”*

Intermediate output:

[('supermarket', 'is organized into', 'aisles'), ('aisle', 'contains', 'product categories'), ('items', 'are', 'product categories'), ('sanitary items', 'are', 'product categories'), ('cleaning products', 'are', 'product categories'), ('drinks', 'are', 'product categories'), ('food', 'are', 'product categories'), ('product category', 'contains', 'products'), ('products', 'are stacked into', 'shelves'), ('location', 'is in', 'supermarket'), ('product', 'has', 'code'), ('product', 'has', 'location in supermarket'), ('product', 'has', 'name'), ('product', 'has', 'location'), ('customer', 'picks', 'shopping cart'), ('customer', 'enters', 'supermarket'), ('customer', 'go to', 'cashier desk'), ('customer', 'pay', 'products'), ('cashier desk', 'has', 'cashier'), ('cashier desk', 'has', 'till'), ('cashier desk', 'has', 'scanner'), ('cashiers', 'receiving', 'payment'), ('cashiers', 'hand', 'receipt'), ('cashiers', 'receiving payment from', 'customer')]

The UML class metadata output:

```
Class: Supermarket
   Attribute: ['location']
Class: Aisle
   Attribute: []
Class: ProductCategory
   Attribute: []
Class: Item
   Attribute: []
Class: SanitaryItem
   Attribute: []
Class: CleaningProduct
   Attribute: []
Class: Drink
   Attribute: []
Class: Food
   Attribute: []
Class: Product
   Attribute: ['code', 'name', 'location']
Class: Shelf
   Attribute: []
Class: LocationInSupermarket
   Attribute: []
Class: Customer
   Attribute: []
Class: ShoppingCart
   Attribute: []
Class: CashierDesk
   Attribute: ['till', 'scanner']
Class: Cashier
   Attribute: []
Class: Payment
   Attribute: []
Class: Receipt
   Attribute: []
```

```
Association: is organized into
from: Supermarket
  multiplicity: 0..*
to: Aisle
   multiplicity: 0..*

Composition: contains
from: Aisle
  multiplicity: 0..*
to: ProductCategory
   multiplicity: 0..*

Subtyping:
from: Item
  multiplicity:
to: ProductCategory
   multiplicity:

Subtyping:
from: SanitaryItem
  multiplicity:
to: ProductCategory
   multiplicity:

Subtyping:
from: CleaningProduct
  multiplicity:
to: ProductCategory
   multiplicity:

Subtyping:
from: Drink
  multiplicity:
to: ProductCategory
   multiplicity:
```

Subtyping:
from: Food
  multiplicity:
to: ProductCategory
   multiplicity:

Composition: contains
from: ProductCategory
  multiplicity: 0..*
to: Product
   multiplicity: 0..*

Association: are stacked into
from: Product
  multiplicity: 0..*
to: Shelf
   multiplicity: 0..*

Composition: has
from: Product
  multiplicity: 0..*
to: LocationInSupermarket
   multiplicity: 0..*

Association: picks
from: Customer
  multiplicity: 0..*
to: ShoppingCart
   multiplicity: 0..*

Association: enters
from: Customer
  multiplicity: 0..*
to: Supermarket
   multiplicity: 0..*

Association: placed in
from: Product
  multiplicity: 0..*
to: ShoppingCart
   multiplicity: 0..*

Association: go to
from: Customer
  multiplicity: 0..*
to: CashierDesk
   multiplicity: 0..*

Association: pay
from: Customer
  multiplicity: 0..*
to: Product
   multiplicity: 0..*

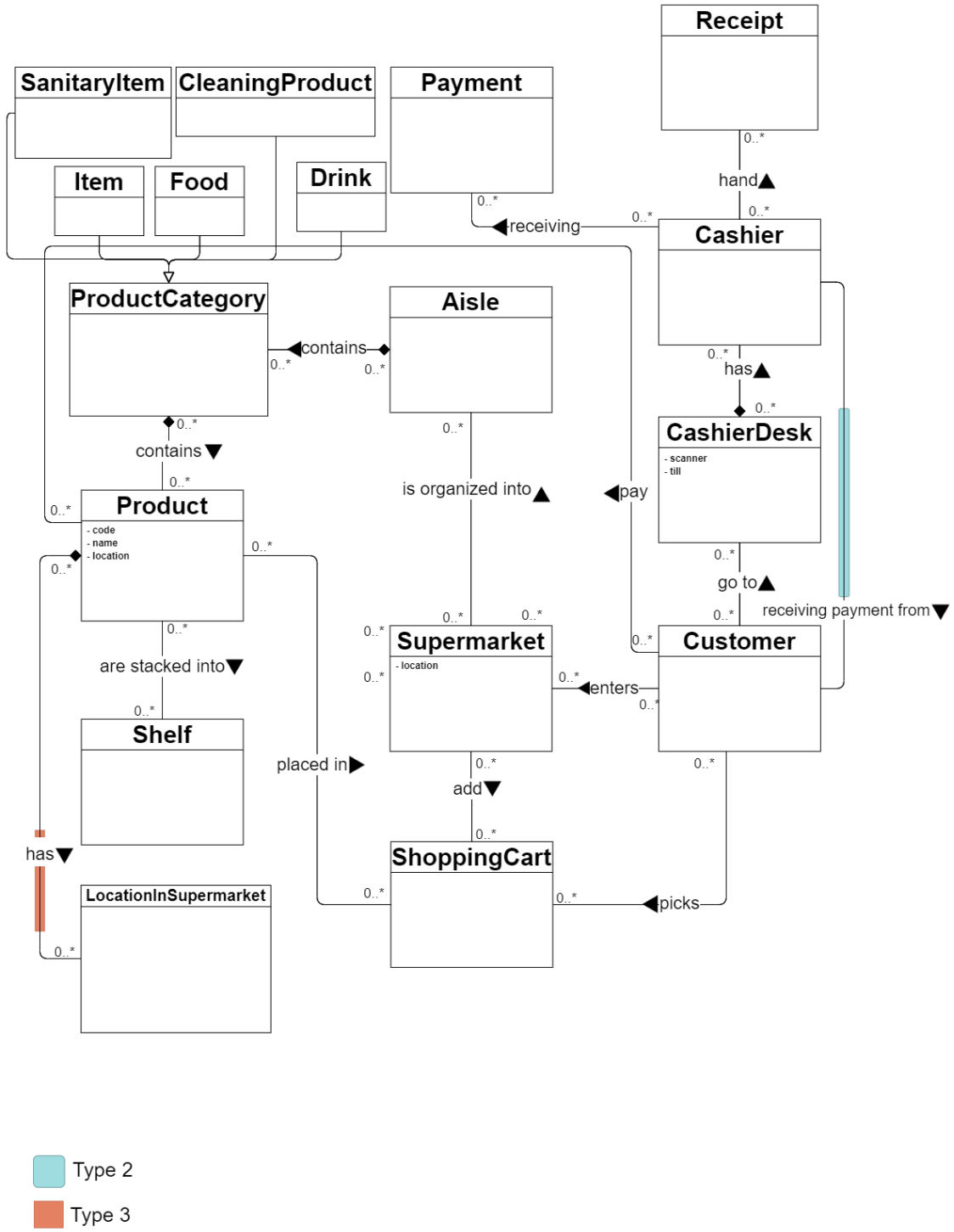
Composition: has
from: CashierDesk
  multiplicity: 0..*
to: Cashier
   multiplicity: 0..*

Association: receiving
from: Cashier
  multiplicity: 0..*
to: Payment
   multiplicity: 0..*

Association: hand
from: Cashier
  multiplicity: 0..*
to: Receipt
   multiplicity: 0..*

Association: receiving payment from
from: Cashier
  multiplicity: 0..*
to: Customer
   multiplicity: 0..*

UML class diagram:



Error explanation:

Type 2: Multiple entities problem (Three or more entities exist in one sentence, and those entities have relationships with each other)

- “After receiving payment from the customer, cashiers will hand them a receipt.” Triplets: [('cashiers', 'receiving', 'payment'), ('cashiers', 'hand', 'receipt'), **('cashiers', 'receiving payment from', 'customer')**] The ideal

result is [(‘cashier’, ‘receive’, ‘payment’), (‘cashier’, ‘hand’, ‘receipt’), (‘customer’, ‘proceed’, ‘payment’), (‘customer’, ‘receive’, ‘receipt’)]

Type 3: Noun + Prep + Noun problem (when there is a preposition between two nouns)

- “Each product has a name, code, and location in the supermarket.” Triplets: [('location', 'is in', 'supermarket'), ('product', 'has', 'code'), **('product', 'has', 'location in supermarket')**, ('product', 'has', 'name'), ('product', 'has', 'location')]

Others: (Duplication problem)

- “Food, drinks, sanitary items, and cleaning products are all product categories.” Triplets: [('items', 'are', 'product categories'), ('sanitary items', 'are', 'product categories'), ('cleaning products', 'are', 'product categories'), ('drinks', 'are', 'product categories'), ('food', 'are', 'product categories')]

A.2. Input Scenario 3 (Online Shopping System)

Requirement title
Online Shopping System
Requirement text
“The online shopping system allows customers to search for products by category, and to order them. Each category contains sub categories or products. Cars, bicycles and motorbikes are sub categories. Customers can search for products matching their search criteria. An administrator manages the categories and product information. Customers can create accounts. An account will consist of various information such as name, address, phone number, email, and so on. *Customers can add one or more products to the shopping cart. The shopping cart lists the products, and shows their price. It also shows the total price of the items in the shopping cart. Customers can remove products from the shopping cart before checkout.* *The payment process is triggered when customers confirm the order. Customers will pay for the products, and receive a confirmation email. The confirmation email shows the order information. Order information consists of customer, products, prices, quantities, delivery address, and delivery date.”*

Intermediate output:

[('shopping', 'allows', 'customers'), ('customers', 'search', 'products by category'), ('customers', 'search', 'products'), ('category', 'contains', 'products'), ('category', 'contains', 'sub categories'), ('bicycles', 'are', 'sub categories'), ('cars', 'are', 'sub categories'), ('bicycles', 'are', 'categories'), ('motorbikes', 'are', 'categories'), ('motorbikes', 'are', 'sub categories'), ('cars', 'are', 'categories'), ('customers', 'can search', 'products'), ('administrator', 'manages', 'categories'), ('administrator', 'manages', 'product'), ('customers', 'can create', 'accounts'), ('account', 'consist', 'name'), ('customers', 'add', 'products', 'shopping', 'cart'), ('shopping cart', 'shows', 'price'), ('shopping cart', 'lists', 'products'), ('items', 'is in', 'shopping cart'), ('customers', 'can remove products before', 'checkout'), ('customers', 'can remove products from', 'shopping cart'), ('customers', 'can remove', 'products'), ('customers', 'confirm', 'order'), ('customers', 'receive', 'confirmation email'), ('customers', 'pay', 'products'), ('confirmation email', 'shows', 'order'), ('order', 'consists', 'quantities'), ('order', 'consists', 'delivery date'), ('order', 'consists', 'prices'), ('order', 'consists', 'customer'), ('order', 'consists', 'products'), ('order', 'consists', 'delivery address')]

The UML class metadata output:

```
Class: Shopping
   Attribute: []
Class: Customer
   Attribute: []
Class: ProductsByCategory
   Attribute: []
Class: Product
   Attribute: []
Class: Category
   Attribute: []
Class: SubCategory
   Attribute: []
Class: Bicycle
   Attribute: []
Class: Car
   Attribute: []
Class: Motorbike
   Attribute: []
Class: Administrator
   Attribute: []
Class: Account
   Attribute: ['name']
Class: ShoppingCart
   Attribute: ['price']
Class: Item
   Attribute: []
Class: Checkout
   Attribute: []
Class: Order
   Attribute: ['quantities', 'delivery date', 'prices', 'delivery address']
Class: ConfirmationEmail
   Attribute: []
```

Association: allows
from: Shopping
multiplicity: 0..*
to: Customer
multiplicity: 0..*

Association: search
from: Customer
multiplicity: 0..*
to: ProductByCategory
multiplicity: 0..*

Association: search
from: Customer
multiplicity: 0..*
to: Product
multiplicity: 0..*

Composition: contains
from: Category
multiplicity: 0..*
to: Product
multiplicity: 0..*

Composition: contains
from: Category
multiplicity: 0..*
to: SubCategory
multiplicity: 0..*

Subtyping:
from: Bicycle
multiplicity:
to: SubCategory
multiplicity:

Subtyping:
from: Car
multiplicity:
to: SubCategory
multiplicity:

Subtyping:
from: Bicycle
multiplicity:
to: Category
multiplicity:

Subtyping:
from: Motorbike
multiplicity:
to: Category
multiplicity:

Subtyping:
from: Motorbike
multiplicity:
to: SubCategory
multiplicity:

Subtyping:
from: Car
multiplicity:
to: Category
multiplicity:

Association: can search
from: Customer
multiplicity: 0..*
to: Product
multiplicity: 0..*

Association: manages
from: Administrator
multiplicity: 0..*
to: Category
multiplicity: 0..*

Association: manages
from: Administrator
multiplicity: 0..*
to: Product
multiplicity: 0..*

Association: can create
from: Customer
multiplicity: 0..*
to: Account
multiplicity: 0..*

Association: add
from: Customer
multiplicity: 0..*
to: Product
multiplicity: 0..*

Association: lists
from: ShoppingCart
multiplicity: 0..*
to: Product
multiplicity: 0..*

Association: is in
from: Item
multiplicity: 0..*
to: ShoppingCart
multiplicity: 0..*

Association: can remove products before
from: Customer
  multiplicity: 0..*
to: Checkout
   multiplicity: 0..*

Association: can remove products from
from: Customer
  multiplicity: 0..*
to: ShoppingCart
   multiplicity: 0..*

Association: can remove
from: Customer
  multiplicity: 0..*
to: Product
   multiplicity: 0..*

Association: confirm
from: Customer
  multiplicity: 0..*
to: Order
   multiplicity: 0..*

Association: receive
from: Customer
  multiplicity: 0..*
to: ConfirmationEmail
   multiplicity: 0..*

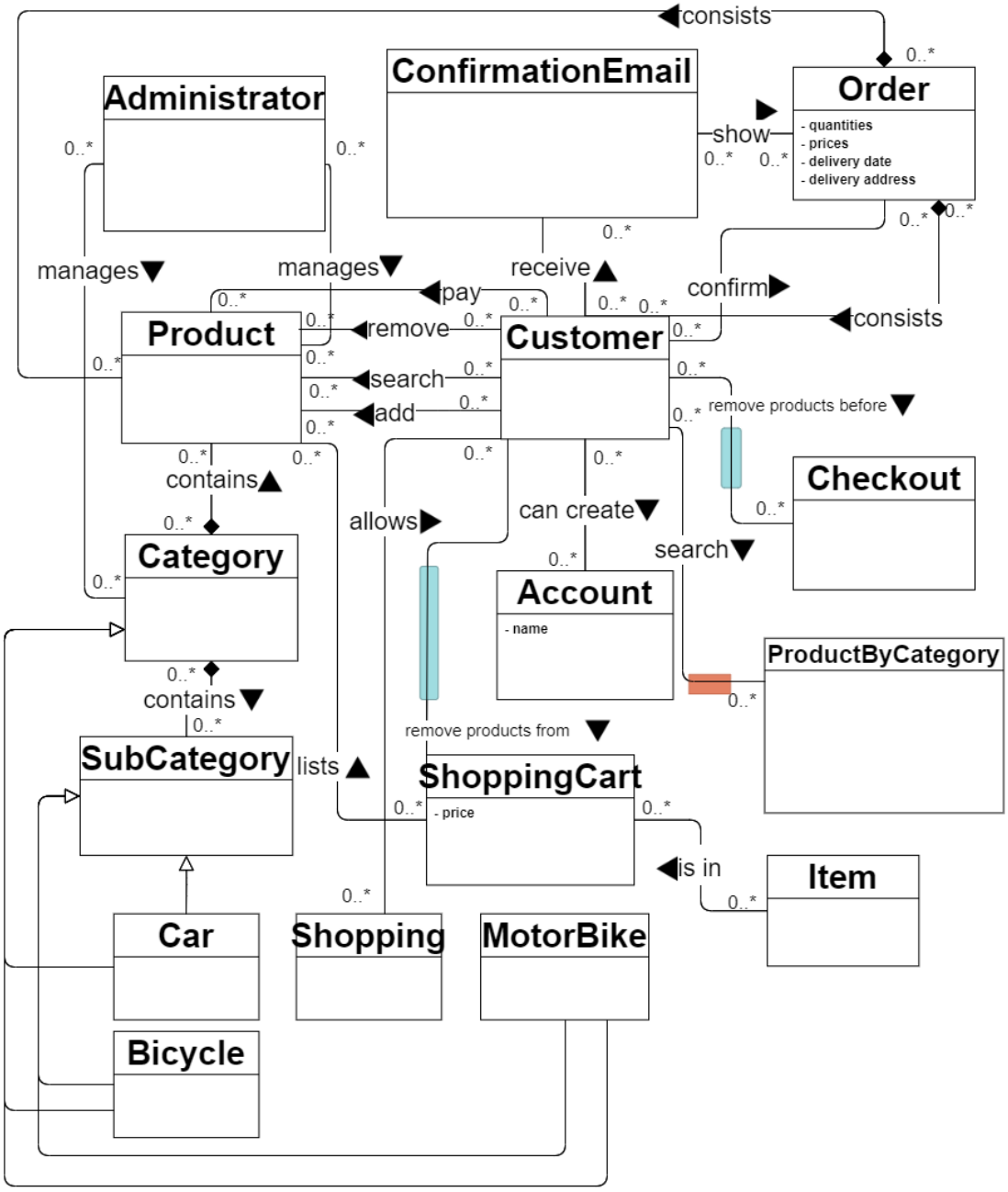
Association: pay
from: Customer
  multiplicity: 0..*
to: Product
   multiplicity: 0..*

Association: shows
from: ConfirmationEmail
  multiplicity: 0..*
to: Order
   multiplicity: 0..*

Composition: consists
from: Order
  multiplicity: 0..*
to: Customer
   multiplicity: 0..*

Composition: consists
from: Order
  multiplicity: 0..*
to: Product
   multiplicity: 0..*

UML class diagram:



Error explanation:

Type 1: Incomplete information extraction. (OpenIE fails to extract information from part of a sentence)

- “An account will consist of various information such as name, address, phone number, and email.” Triplets: [('account', 'consist', 'name')]
- “It also shows the total price of the items in the cart. Triplets: [('items', 'is in',

'cart')]

Type 2: Multiple entities problem (Three or more entities exist in one sentence, and those entities have relationships with each other)

- "Customers can remove products from the shopping cart before checkout." Triplets: [('customers', 'remove', 'products'), **('customers', 'remove products from', 'shopping cart'), ('customers', 'remove products before', 'checkout')**] The ideal result: [('customer', 'remove', 'products'), ('shopping cart', 'has', 'products')]

Type 3: Noun + Prep + Noun problem (when there is a preposition between two nouns)

- "The online shopping system allows customers to search for products by category, and to order them." [('shopping', 'allows', 'customers'), **('customers', 'search', 'products by category')**, ('customers', 'search', 'products')]

Others: (Duplication Problem)

- "Cars, bicycles and motorbikes are sub categories." Triplet: [('bicycles', 'are', 'sub categories'), ('cars', 'are', 'sub categories'), ('bicycles', 'are', 'categories'), ('motorbikes', 'are', 'categories'), ('motorbikes', 'are', 'sub categories'), ('cars', 'are', 'categories')]

A.3. Input Scenario 4 (Course Attendance System)

Requirement title
Course Attendance System
Requirement text
“A student will enroll for one or more courses. Business Courses and Science Courses are types of courses. Each course consists of multiple lectures, and have a course name, code and date. A course coordinator organizes the courses. Each course has one or more lecturers, a location, time slot, and set of dates. *Lecturers will give lectures, and administrators will make announcements that are for a particular course. Each course will have assignments and an exam. Students must attend to the lectures, complete the assignments and take the exam. Lecturers will give grades. A course grade consists of an assignment grade and exam grade. An exam is either a first exam or a re-take. Students will receive their course grades by email.”*

Intermediate output:

[('student', 'enroll', 'courses'), ('business courses', 'are', 'courses'), ('science courses', 'are', 'courses'), ('course', 'have', 'course name'), ('course', 'have', 'code'), ('course', 'have', 'date'), ('course', 'consists', 'lectures'), ('course coordinator', 'organizes', 'courses'), ('course', 'has', 'dates'), ('course', 'has', 'location'), ('course', 'has', 'timeslot'), ('administrators', 'make', 'announcements'), ('lecturers', 'give', 'lectures'), ('course', 'have', 'exam'), ('course', 'have', 'assignments'), ('students', 'attend to', 'lectures
'), ('students', 'complete', 'assignments'), ('students', 'take', 'exam'), ('lecturers', 'give', 'grades'), ('course grade', 'consists', 'exam grade'), ('course grade', 'consists', 'assignment grade'), ('exam', 'is', 'first'), ('exam', 'is', 'exam'), ('exam', 'is', 'first exam'), ('students', 'receive', 'course grades')]

The UML class metadata output:

```
Class: Student
   Attribute: []
Class: Course
   Attribute: ['course name', 'code', 'date', 'location', 'time slot']
Class: BusinessCourse
   Attribute: []
Class: ScienceCourse
   Attribute: []
Class: Lecture
   Attribute: []
Class: CourseCoordinator
   Attribute: []
Class: Administrator
   Attribute: []
Class: Announcement
   Attribute: []
Class: Lecturer
   Attribute: []
Class: Exam
   Attribute: []
Class: Assignment
   Attribute: []
Class: Grade
   Attribute: []
Class: CourseGrade
   Attribute: []
Class: ExamGrade
   Attribute: []
Class: AssignmentGrade
   Attribute: []
Class: FirstExam
   Attribute: []
```

Association: enroll
from: Student
  multiplicity: 0..*
to: Course
   multiplicity: 0..*

Association: give
from: Lecturer
  multiplicity: 0..*
to: Lecture
   multiplicity: 0..*

Subtyping: are
from: BusinessCourse
  multiplicity: 0..*
to: Course
   multiplicity: 0..*

Composition: have
from: Course
  multiplicity: 0..*
to: Exam
   multiplicity: 0..*

Subtyping: are
from: ScienceCourse
  multiplicity: 0..*
to: Course
   multiplicity: 0..*

Composition: have
from: Course
  multiplicity: 0..*
to: Assignment
   multiplicity: 0..*

Composition: consists
from: Course
  multiplicity: 0..*
to: Lecture
   multiplicity: 0..*

Association: attend to
from: Student
  multiplicity: 0..*
to: Lecture
   multiplicity: 0..*

Association: organizes
from: CourseCoordinator
  multiplicity: 0..*
to: Course
   multiplicity: 0..*

Association: complete
from: Student
  multiplicity: 0..*
to: Assignment
   multiplicity: 0..*

Association: make
from: Administrator
  multiplicity: 0..*
to: Announcement
   multiplicity: 0..*

Association: take
from: Student
  multiplicity: 0..*
to: Exam
   multiplicity: 0..*

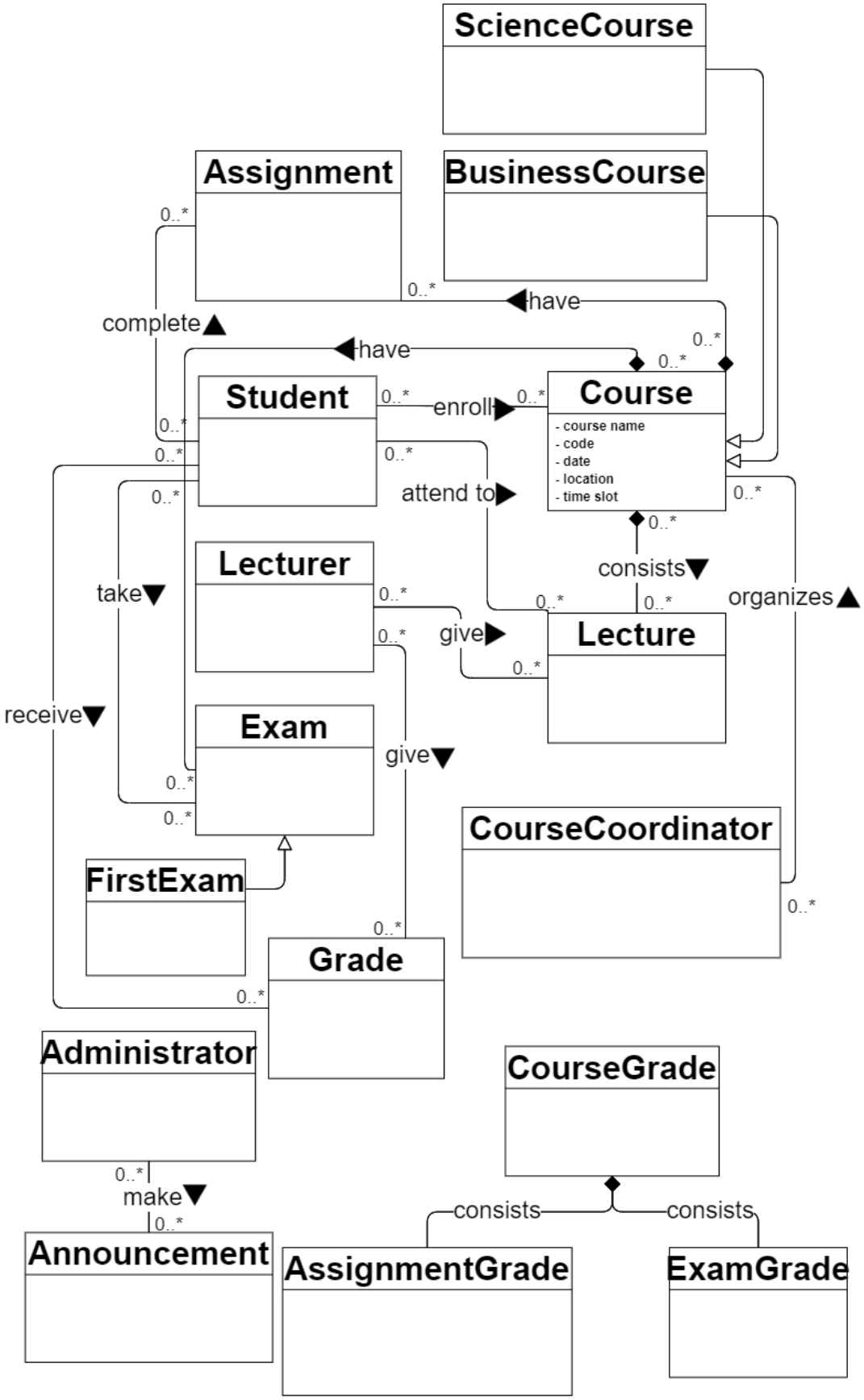
Association: give
from: Lecturer
  multiplicity: 0..*
to: Grade
   multiplicity: 0..*

Composition: consists
from: CourseGrade
  multiplicity: 0..*
to: ExamGrade
   multiplicity: 0..*

Composition: consists
from: CourseGrade
  multiplicity: 0..*
to: AssignmentGrade
   multiplicity: 0..*
Subtyping:
from: Exam
  multiplicity:
to: FirstExam
   multiplicity:

Association: receive
from: Student
  multiplicity: 0..*
to: Grade
   multiplicity: 0..*

UML class diagram:



Error explanation:

Type 1: Incomplete information extraction. (OpenIE fails to extract information from part of a sentence)

- “Each course has one or more lecturers, a location, time slot, and set of dates.” Triplets: [('course', 'has', 'location'), ('course', 'has', 'date'), ('course', 'has', 'time slot')] OpenIE fail to extract (‘course’, ‘has’, ‘lecturers’).
- “An exam is either the first exam or a re-take.” Triplets: ('exam', 'is', 'first exam'). OpenIE fails to extract (‘exam’, ‘is’, ‘re-take’)

A.4. Input Scenario 5 (Hospital System)

Requirement title
Hospital System
Requirement text
“Patients will initially visit their doctor when they are ill. The doctor will then diagnose the patients and write a report. The diagnosis can be a physical one, a psychological one or a psychiatric one. The report contains illness conditions, diagnostic result, and suggestions that are for treatment. If the doctor suggests medication, the report will list the specific drugs. The patient will then collect their drugs from the pharmacy, and pay for the bill. If the doctor suggests an operation, the patient will be requested to make an appointment for the operation with the hospital. A surgeon and an operation team are assigned to perform the operation. A surgeon typically performs multiple operations on a particular day. Following the operation, the patient will receive care in the hospital. Care can involve physiotherapy, osteotherapy or mental support. The patients will be assigned a bed that is on a ward, and will be cared for by nurses. A nurse will care for several patients.”

Intermediate output:

[('patients', 'visit', 'doctor'), ('doctor', 'diagnose', 'patients'), ('doctor', 'write', 'report'), ('diagnosis', 'be', 'physical'), ('report', 'contains', 'result'), ('report', 'contains', 'suggestions'), ('report', 'contains', 'diagnostic result'), ('report', 'contains', 'illness conditions'), ('report', 'list', 'drugs'), ('doctor', 'suggests', 'medication'), ('patient', 'pay', 'bill'), ('patient', 'collect', 'drugs'), ('patient', 'make', 'appointment'), ('patient', 'make', 'appointment operation'), ('doctor', 'suggests', 'operation'), ('operation team', 'perform', 'operation'), ('surgeon', 'performs', 'operations'), ('surgeon', 'performs operations on', 'day'), ('patient', 'receive care following', 'operation'), ('patient', 'receive care in', 'hospital'), ('patient', 'receive', 'care'), ('care', 'can involve', 'osteotherapy'), ('care', 'can involve', 'physiotherapy'), ('care', 'can involve', 'mental support'), ('care', 'can involve', 'support'), ('patients', 'be cared', 'nurses'), ('nurse', 'care', 'patients')]

The UML class metadata output:

```
Class: Patient
   Attribute: []
Class: Doctor
   Attribute: []
Class: Report
   Attribute: ['suggestions', 'diagnostic result', 'illness conditions']
Class: Diagnosis
   Attribute: []
Class: Physical
   Attribute: []
Class: Result
   Attribute: []
Class: Drug
   Attribute: []
Class: Medication
   Attribute: []
Class: Bill
   Attribute: []
Class: Appointment
   Attribute: []
Class: AppointmentOperation
   Attribute: []
Class: Operation
   Attribute: []
Class: OperationTeam
   Attribute: []
Class: Surgeon
   Attribute: []
Class: Day
   Attribute: []
Class: Hospital
   Attribute: []
Class: Care
   Attribute: []
Class: Osteotherapy
   Attribute: []
Class: Physiotherapy
   Attribute: []
Class: MentalSupport
   Attribute: []
Class: Support
   Attribute: []
Class: Nurse
   Attribute: []
```

Association: visit
from: Patient
  multiplicity: 0..*
to: Doctor
   multiplicity: 0..*

Association: suggests
from: Doctor
  multiplicity: 0..*
to: Medication
   multiplicity: 0..*

Association: diagnose
from: Doctor
  multiplicity: 0..*
to: Patient
   multiplicity: 0..*

Association: pay
from: Patient
  multiplicity: 0..*
to: Bill
   multiplicity: 0..*

Association: write
from: Doctor
  multiplicity: 0..*
to: Report
   multiplicity: 0..*

Association: collect
from: Patient
  multiplicity: 0..*
to: Drug
   multiplicity: 0..*

Subtyping:
from: Diagnosis
  multiplicity:
to: Physical
   multiplicity:

Association: make
from: Patient
  multiplicity: 0..*
to: Appointment
   multiplicity: 0..*

Composition: contains
from: Report
  multiplicity: 0..*
to: Result
   multiplicity: 0..*

Association: make
from: Patient
  multiplicity: 0..*
to: AppointmentOperation
   multiplicity: 0..*

Association: list
from: Report
  multiplicity: 0..*
to: Drug
   multiplicity: 0..*

Association: suggests
from: Doctor
  multiplicity: 0..*
to: Operation
   multiplicity: 0..*

Association: perform
from: OperationTeam
  multiplicity: 0..*
to: Operation
   multiplicity: 0..*

Association: performs
from: Surgeon
  multiplicity: 0..*
to: Operation
   multiplicity: 0..*

Association: performs operations on
from: Surgeon
  multiplicity: 0..*
to: Day
   multiplicity: 0..*

Association: receive care following
from: Patient
  multiplicity: 0..*
to: Operation
   multiplicity: 0..*

Association: receive care in
from: Patient
  multiplicity: 0..*
to: Hospital
   multiplicity: 0..*

Association: receive
from: Patient
  multiplicity: 0..*
to: Care
   multiplicity: 0..*

Subtyping:
from: Care
  multiplicity:
to: Osteotherapy
   multiplicity:

Subtyping:
from: Care
  multiplicity:
to: Physiotherapy
   multiplicity:

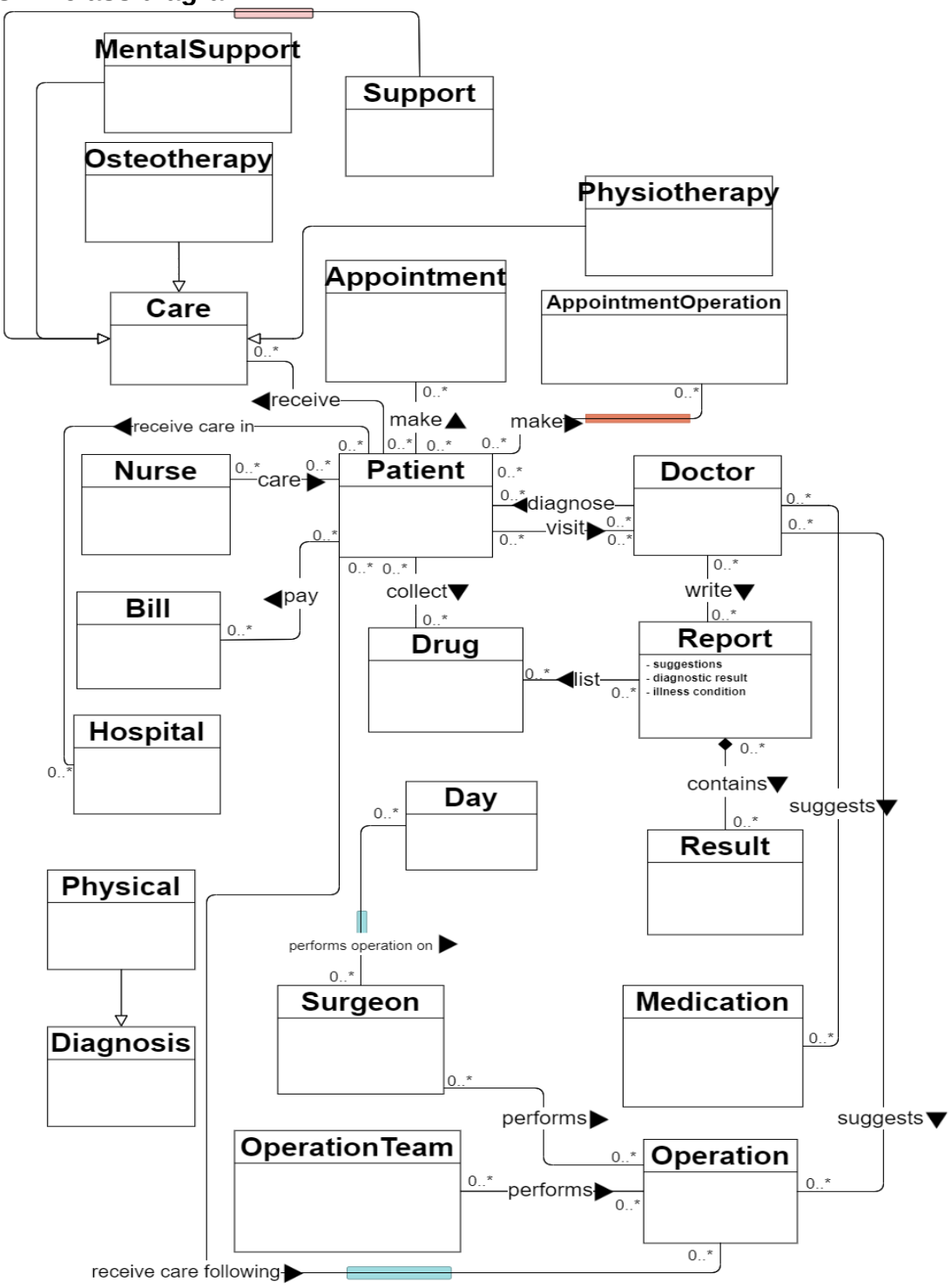
Subtyping:
from: Care
  multiplicity:
to: MentalSupport
   multiplicity:

Subtyping:
from: Care
  multiplicity:
to: Support
   multiplicity:

Association: be cared
from: Patient
  multiplicity: 0..*
to: Nurse
   multiplicity: 0..*

Association: care
from: Nurse
  multiplicity: 0..*
to: Patient
   multiplicity: 0..*

UML class diagram:



Error explanation:

Type 1: Incomplete information extraction. (OpenIE fails to extract information from part of a sentence)

- "The diagnosis can be a physical, a psychological or a psychiatric." Triplets: [('diagnosis', 'be', 'physical')]
- "The patients will be assigned a bed on a ward, and will be cared for by nurses." Triplets: [('patients', 'be cared', 'nurses')]

Type 2: Multiple entities problem (Three or more entities exist in one sentence, and those entities have relationships with each other)

- "A surgeon typically performs multiple operations on a particular day." Triplets: [('surgeon', 'performs', 'operations'), **('surgeon', 'performs operations on', 'day')**] The ideal result should be: [('surgeon', 'performs', 'operations'), ('operation', 'is in', 'day')], or simply ('surgeon', 'performs', 'operations')
- "Following the operation, the patient will receive care in the hospital." Triplets: [**('patient', 'receive care following', 'operation'), ('patient', 'receive care in', 'hospital')**, ('patient', 'receive', 'care')] The ideal result should be: [('patient', 'receive', 'care'), ('care', 'is in', 'hospital')]

Type 3: Noun + Prep + Noun problem (when there is a preposition between two nouns)

- "If the doctor suggests an operation, the patient will be requested to make an appointment for the operation with the hospital." Triplets: [('patient', 'make', 'appointment'), ('patient', 'make', 'appointment operation'), ('doctor', 'suggests', 'operation')] OpenIE seizes "appointment for operation" as an object, data preprocessing in this program remove stop words like "for", so the object result is "appointment operation".

Others (Duplication problem)

- "The report contains illness conditions, diagnostic result, and suggestions that are for treatment." Triplets: [('report', 'contains', 'result'), ('report', 'contains', 'suggestions'), ('report', 'contains', 'diagnostic result'), ('report', 'contains', 'illness conditions')] Duplication problem with result and diagnostic result.

"Care can involve physiotherapy, osteotherapy or mental support." Triplets: [('care', 'can involve', 'osteotherapy'), ('care', 'can involve', 'physiotherapy'),

('care', 'can involve', 'mental support'), ('care', 'can involve', 'support')]

Duplication problem with the last triplet.

Appendix B: Questionnaire

Evaluation for UML Class Metadata Generation Tool – Online Questionnaire	
Dear UML experts, *Thank you for taking the time out of your busy schedule to help with our survey work. Your contribution will take an important role in our research. This questionnaire consists of two parts, with 13 questions in total. The first part aims at asking for feedback based on the document and video we have sent to you by email. The second part is asking general opinions regarding Natural Language Processing (NLP) and requirement analysis.* *Your participation will greatly help us in verifying the results of our research. Your contribution will be personally acknowledged in the introduction of the thesis, and you will receive a copy of the thesis once finished.* *Best Regards,* *Tiantian Tang, Master student ICT in Business*	
1. How would you rate the accuracy of our UML class metadata results if you take the identified bugs into consideration? *(The bugs are rooted from the libraries we used for our program. In the document, we have identified and categorized those bugs. We have also marked and explained those errors in the UML class diagram section. For this question, please include the errors we have marked to rate the accuracy of results.)*	Compeletly inaccurate ● 1 ● 2 ● 3 ● 4 ● 5 ● 6 ● 7 ● 8 ● 9 ● 10 Compeletely accurate

2. How would you rate the accuracy of our UML class metadata results exclusive of identified bugs? *(For this question, please take out the identified errors and think our result accuracy.)*	Compeletly inaccurate ● 1 ● 2 ● 3 ● 4 ● 5 ● 6 ● 7 ● 8 ● 9 ● 10 Compeletely accurate
3. According to the UML class metadata results and screen demo recording, do you think our application can help with software requirement analysis?	● Yes ● No ● Maybe ● Don't know
Could you please briefly explain why you gave this answer to Question 3?	
4. After watching the screen recording video, which of the following advantages do you think of our results have?	● Accurate ● Easy-to-read ● Easy-to-modify ● Other advantages
Could you describe it more specifically if you choose "Other advantage" in Question 4?	
5. We did not have the time to develop a graphical UML Class modeler (a separate project is under way), but do you feel the textual metadata format is a readable, and useful alternative?	● Yes ● No ● Maybe ● Don't know
Could you please briefly explain why you gave this answer to Question 5?	
6. Do you think the "Runnable Prototype" page of our application makes it easier to get end users (your customer) more involved during the requirements analysis process by enabling them to comment on a (simple) application prototype?	● Yes ● No ● Maybe ● Don't know

Could you please briefly explain why you gave this answer to Question 6?	
7. Do you think our application would have value in real word requirements elicitation settings? (if the identified bugs are removed)	● Yes ● No ● Maybe ● Don't know
Could you please briefly explain why you gave this answer to Question 7?	
8. Do you have any other suggestion for improving our application?	
9. Have you ever used a similar UML model generator product (with NLP techniques applied)?	● Yes ● No
10. If yes with question 9, are you satisfied with that product?	● Satisfied ● Somewhat satisfied ● Not satisfied ● Don't know
11. Could you explain the advantages and disadvantages of the NLP based UML tools you have used?	
12. What would you look for in an NLP based UML product?	
13. Do you have any other comments to add for what we have not covered in this questionnaire?	
(Optional) Would you like to provide an additional requirements text as input for our application? If so, please e-mail Tiantian Tang t.tang@umail.leidenuniv.nl and we will return the results.	