



Universiteit Leiden

ICT in Business and the Public Sector

UML Use Case generation from textual requirements using
NLP techniques

Name: Anna Roussou

Student-no: s2955156

Date: 15/11/2022

1st supervisor: Dr. Guus J. Ramackers

2nd supervisor: Dr. Suzan Verberne

MASTER'S THESIS

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

Abstract

Business establishments have come to realise the benefits of involving users in the software development life cycle. User requirements, expressed in natural language, are gathered and transformed into UML Use Case models, which represent the interaction between the users and the system. To reduce the time and cost needed for this process, this research presents an NLP pipeline that generates UML Use Case metadata from textual user requirements. The pipeline consists of transformer-based models, trained with supervised learning, and rule-based matching. To train the models, we synthesised a new data set that consists of requirements texts, use cases and user stories. We annotated the “*Actors*”, “*Systems*” and “*Use Cases*” in these documents, as well as the relationships between these use case elements. To extract “*Preconditions*”, “*Post-conditions*” and “*Triggers*”, we created rule-based patterns that match the respective phrases in a requirements text. Furthermore, we integrated the NLP pipeline into the Prose to Prototype project, and implemented a UML use case metadata model, together with a set of APIs, to facilitate the future development of a UML Use Case diagramming tool.

Acknowledgements

First of all, I would like to thank my first supervisor and mentor, Dr. Guus Ramackers, who entrusted me with the task of developing a UML Use Case modeller for the P2P project. He was always there when I needed him and he helped me overcome all the obstacles I encountered. He kept me motivated throughout the whole project and he sought to improve my general skillset. As a UML expert, he helped me grasp the fundamental ideas behind the creation of UML Use Cases and create proper annotations for the dataset. I would also like to thank my second supervisor, Dr. Suzan Verberne, because she showed a real interest in this research project. As an NLP expert, she provided guidance and she helped me understand the NLP concepts. With her expert opinion and keen eye for detail, I managed to create a quality data set and train the NLP models properly.

I would also like to thank Willem-Peter van Vlokhoven, Pepijn Griffioen and Max Boone, for their assistance in integrating the Use modeller into P2P system. I would be lost without them!

Finally, I would like to thank Greg, Nala and Chelf for their constant support during this challenging period.

Contents

1	Introduction	6
1.1	Problem statement	6
1.2	Research objectives	7
1.3	Research method	8
1.4	Academic contribution	9
1.5	Overview	9
2	Background and Related Work.....	10
2.1	UML Use Case models.....	10
2.2	User Stories	11
2.3	Prose to Prototype project	12
2.4	Related Work.....	13
2.5	NLP models architecture	14
2.5.1	Transition-based Named Entity Recogniser.....	15
2.5.2	Named Entity Recogniser with RoBERTa-base transformer	18
2.5.3	Span categoriser.....	19
2.5.4	Relation extractor	20
3	Data	22
3.1	Data Collection.....	22
3.1.1	Public Requirements Data Set (PURE)	22
3.1.2	User Stories Data Set.....	23
3.1.3	Tera-PROMISE	23
3.2	Data Annotation.....	24
3.2.1	Annotation tools	24
3.2.2	Annotation guidelines.....	25
3.2.3	Annotation process	26
3.2.4	Annotation statistics	29
4	Methods.....	31
4.1	NLP Supervised Learning techniques	31
4.1.1	Implementation and Libraries.....	32
4.2	Rule-based matching	33
5	Experiments and Results	35
5.1	Actor, System and Use Case extraction.....	35
5.1.1	NER with Tok2Vec component and Span Categorisation	35
5.1.2	NER with Transformer component	38

5.1.3	Token-Level evaluation of the Transformer – NER model.....	39
5.2	Relation Extraction.....	40
5.3	Rule-based Matcher.....	41
5.4	Evaluation with out-of-sample data.....	42
6	Integration into Prose to Prototype.....	48
6.1	System Design.....	48
6.2	Use Case model specification.....	48
6.2.1	UML Use Case metadata generation pipeline	48
6.2.2	Use Case metamodel	50
6.2.3	Use Case model methods.....	51
7	Discussion.....	52
7.1	Limitations.....	53
7.2	Future work	53
8	Conclusion.....	55
	References	56
	Appendix	61
	Appendix A.....	61
	Appendix B.....	64

1 Introduction

In the first chapter we present an overview of the research project, from its conceptualisation to its realisation. We first state the problems that arise during the software requirements elicitation phase regarding the interaction between business stakeholders and IT experts. Then, we present the research objective, namely a solution that facilitates the communication among stakeholders during the software development process. To highlight our research approach, we list a series of research questions that were our main focus in this project. We also describe the research methodology followed in this research project, as well as our academic contribution. Finally, we provide a chapter overview of the thesis.

1.1 Problem statement

When developing a new system, large and small corporations follow the *Software Development Lifecycle* (SDLC) framework to tackle the transformation of complex business requirements into a new software system [1]. The SDLC framework consists of six phases: plan and requirements analysis, design, implementation, testing, deployment and maintenance.

During the planning and requirements analysis phase, business experts and senior stakeholders are being interviewed or participate in workshops in order provide information regarding the expected functionalities of the developed software [2]. This information is provided in natural language, in a textual or vocal form. This stage in SDLC is considered to be the most fundamental, as it is the pillar on which the project will be built.

After gathering user and system requirements, the next phase of the SDLC framework is to design the system features based on these requirements. The designs derived from the requirements have the form of UML models and pseudo-code and their functionality is to provide structured and clear information to the system developers.

In agile development methods the steps of the SDLC framework are implemented iteratively, namely over multiple short sprints. This approach deliberately involves business experts in the whole system development life cycle. Requirements in the form of user stories and use case models are being gathered and prioritised at the start of each iteration and at a later state of the program increment, a testing phase is being added as the final task of each iteration. During testing phase, users are being presented with a working demo and provide feedback based on how well the demo fits the requirements. As a result, many changes in system requirements occur, which need to be manually documented and modelled by IT experts.

Although software systems are developed based on the specifications provided by the business experts, their involvement in the development process is challenging. The reason is that the available UML modelling tools are addressed only to software developers and are not equipped with functionalities to facilitate communication and rapid feedback between business stakeholders and IT experts. Furthermore, the currently available UML modelling tools do not provide automated model building and model adaptations. This procedure, especially in large projects, becomes a challenging and time-consuming task for the IT experts, while also increases project costs.

The majority of the early solutions presented in the research field, require constant human intervention during the process of UML model generation [3], leading in poor results regarding time and cost reduction. On the other hand, proposed solutions that opted for 100% automated requirements-to-UML model transformation are only feasible if the text input has, at least, a semi-structured form.

The proposed solution to the problems stated above is the development of a web-based UML Use Case modeller that is focused on transforming unstructured textual requirements to UML Use Case models with the utilisation of advanced NLP techniques. Besides unstructured text, requirements in the form of use case and user story templates are also considered. To address the challenge of the ambiguity of natural language, rule-based NLP techniques supplement the model-based NLP algorithms. To reduce human intervention, most of the manual tasks will be automated. However, this research intends to present an interactive human-in-the-loop approach, as a user assisted information system will perform better on the ambiguity challenge, and also address the issue of incompleteness of specifications texts in the early requirements analysis stages [4].

The UML Use Case generator will be a component of the larger Prose to Prototype tool, among other UML modellers such as Class and Activity [4].

1.2 Research objectives

The main objective of this research is to reduce time and cost for both the users and the system analysts by enabling rapid development of high-quality requirements. In addition, it aims to involve business experts and users more effectively in the requirements analysis stage by providing them with tools that enhance interaction among them. This research focuses specifically on the development of a UML Use Case transformation tool, since use cases represent system requirements from the user's perspective. The derived research question is:

How can the requirements definition process be improved by implementing NLP techniques to automate the development of UML Use Case models?

Subsequently, available model-based NLP tools will be studied and their fit to this project will be evaluated. These frameworks can be used for a wide variety of NLP tasks such as sentiment analysis, question answering and named entity recognition. To fine tune the model-based tools for this specific project, rule-based models that use linguistic rules and patterns will be utilised. This hybrid approach will answer the question:

Which combination of ruled-based and model-based techniques best fit this application?

We base the first research objective on these research questions, namely:

ROI: Develop a prototype to transform requirements texts and user stories into a UML use case model using a combination of supervised NLP and rule-based techniques.

To train the NLP models with supervised learning we need a large annotated data set that encapsulates specific aspects of business requirements documents and user stories. The data set will also facilitate the development of rule-based models, as it will provide insight about the rules that we need to create to extract relevant information. Therefore, we also need to consider:

How can we build a quality data set for training, evaluation and development of the UML Use Case transformation model?

The actors presented in the use case requirements texts make use of different functionalities in the software. These functionalities must be adequately represented in the prototype, to allow users a better understanding of the software implementation. As such, it is also important to explore:

How can the prototype reflect software requirements from multiple points of view?

The research questions stated above lead to the definition of the second research objective:

RO2: Research, synthesise and annotate a requirements data set to adequately train and evaluate NLP models.

Although the objective of an automated solution is to minimise human intervention, flawless information extraction from texts cannot be achieved fully, mainly due to the ambiguity of natural language. To overcome this issue, this research intends to involve the human element mainly in tasks that further improve the performance of the application. The derived question is:

How can a human-in-the-loop approach augment the automated solution?

The third research objective is based on the last research question:

RO3: Explore available models and methods that best fit the human-in-the-loop approach.

1.3 Research method

The research method applied in this this is *Design science*, a research method that produces knowledge in a domain that derives from the design and development of a new solution to a research problem rather than review and evaluation of existing solutions and is usually implemented in Information Systems. According to Peffers et al. [5] design science “includes six steps: problem identification and motivation, objectives for a solution, design and development, demonstration, evaluation, and communication.”

Problem identification and motivation: In the first chapter, the research problem of minimising manual tasks during the requirements analysis phase of SDLC is defined, and the value of utilising the available NLP tools and relevant datasets to produce a solution is presented.

Objectives of a solution: The objectives of the research paper are inferred from the problem definition. This part includes the characteristics of the artifact that we developed, namely an application that uses NLP techniques to extract information from requirements and transform them into UML Use Case models. In the second chapter, the proposed solution is compared to the current ones in order to highlight the advantages of the solution and further promote its importance in the academic, as well as, the business world.

Design and development: In the second chapter, we describe the architectural design of the NLP models and in the third chapter we present the development of an annotated

requirements data set. The development of the proposed solution is presented in the fourth and the fifth chapter.

Demonstration: In the sixth chapter that includes the integration of the model to the ngUML project, a concrete example accompanied by detailed documentation is presented to show the efficacy of the model to solve the problem.

Evaluation: In the fifth chapter, we present the results of the experiments with various models and we evaluate and discuss their performance. In the seventh chapter, the observed results are compared to the objectives of the solution described in the previous chapters in the seventh chapter.

Communication: The utility and novelty of the proposed solution are communicated in Chapter 7, as well as its current limitations and proposed future improvements, to motivate researchers and relevant audiences to study and further develop the artifact.

1.4 Academic contribution

The academic contribution of this research is to provide a solution that facilitates the communication between IT and key-users, that involves the latter more in the software development life cycle. The proposed solution is the development of a pipeline that transforms user requirements and user stories into Use Case models. The pipeline consists of novel NLP models that are trained with supervised learning on a custom created data set that has been annotated by us. For this research topic, this is the first time that supervised learning is used to train models on a large data set. Furthermore, this research aims to offer useful insight of the techniques and challenges of developing such a tool. It also provides motivation to future researchers to experiment with it and further improve it. More specifically, the exploration of multiple NLP models, will provide useful knowledge regarding the suitability of those tools for the purposes of transforming text into UML models.

1.5 Overview

Chapter 1 of this thesis was an introduction to the research project. We described the problem and outlined the proposed solution. In the first half of Chapter 2 “Background and Related Work”, we provide information about UML Use Case models and User Stories, as well as information about the Prose to Prototype project and other research projects with objectives similar to ours. In the second half, we present the architecture of the NLP models used for training. Chapter 3 is dedicated to the data set used to train the models. We present the collected requirements texts and user stories and describe the annotation process. In Chapter 4 we show the methods that were used for the information extraction, namely Named Entity Recognition and Relation Extraction with supervised learning for Actors, Systems and Use Cases and Rule-based matching for Triggers, Preconditions and Postconditions. In Chapter 5, we list our experiments with various models and we present the results. Furthermore, we test the models with out-of-sample data and evaluate their performance. In Chapter 6 we describe the integration of the UML Use Case modeller to the ngUML project. In Chapter 7 we discuss the results, limitations and future work. The thesis is concluded with Chapter 8, where we present the key takeaways of this research project.

2 Background and Related Work

This Chapter is dedicated to presenting the basic concepts of UML Use Case models and User Stories, as well as information regarding Prose to Prototype project. We also present the architecture of the machine learning models that we trained to build the Use Case transformer and we provide an overview of the research previously conducted related to the subject.

2.1 UML Use Case models

The first steps of a system development process are the definition and modelling of requirements. During this phase, product requirements are clearly defined, documented and modelled. A Software Requirement Specification document is compiled for the documentation, while standardised UML models are developed by the IT team. UML Use Case models in particular, specify the required functionalities of the system from the user perspective, by showing the relationship between a set of actors and the tasks they perform [6]. A simple example of modelling a textual requirement as a UML Use Case is the following:

***Dinner at a Restaurant:** The customer arrives at the restaurant and the receptionist confirms his/her reservation. Once the customer is seated, a waiter hands him/her the menu. The client reads the menu and gives his/her order to the waiter. The customer can optionally order wine to accompany his/her dinner. The waiter takes the customer's order and informs the chef who cooks the food. When the food is ready, the waiter serves the food to the customer and also his/her wine in case it was ordered. After finishing his/her meal, the customer pays the bill.*

The textual form of this requirement can be represented as a UML Use Case model comprised of the necessary Use Case Elements and their relationships as shown in Figure 1.

The first Use Case element is the Actor, a type of entity that can be a human, an organisation, a device or an external system that interacts with the system [7]. In this example the Actors are the Waiter, the Customer, the Receptionist and the Chef. An Actor can be associated to one or many Use Cases, the second element of a UML Use Case model. Use Cases are verb phrases which specify how the Actors interact with the system. For example, the act of the Chef cooking the food ordered by the Customer is represented by the Use Case “*cook Food*” which is linked with an Interact relationship with the Actor “*Chef*”. Apart from the Actor, the Use Cases can be related to each other through Extension and Inclusion. An extending Use Case is a Use Case that can be optionally “*inserted into the behavior defined in the extended Use Case*” [7]. In the Restaurant example, the “*order wine*” Use Case is extending the “*order food*” relationship, as an optional action of the Customer. On the other hand, the behavior of an included Use Case is part of the behavior of its including Use Case and “*must be available in order to completely describe the included Use Case*” [7]. For example, the customer can read the menu to decide his/her order, but first he/she has to get the menu from the waiter. At the Use Case model this requirement is represented with an including Use Case “*read menu*” that is related to the included Use Case “*get menu*”. The interactions between Actors and Use Cases are set on a defined interface, called System Boundary, in this case the *Restaurant*.

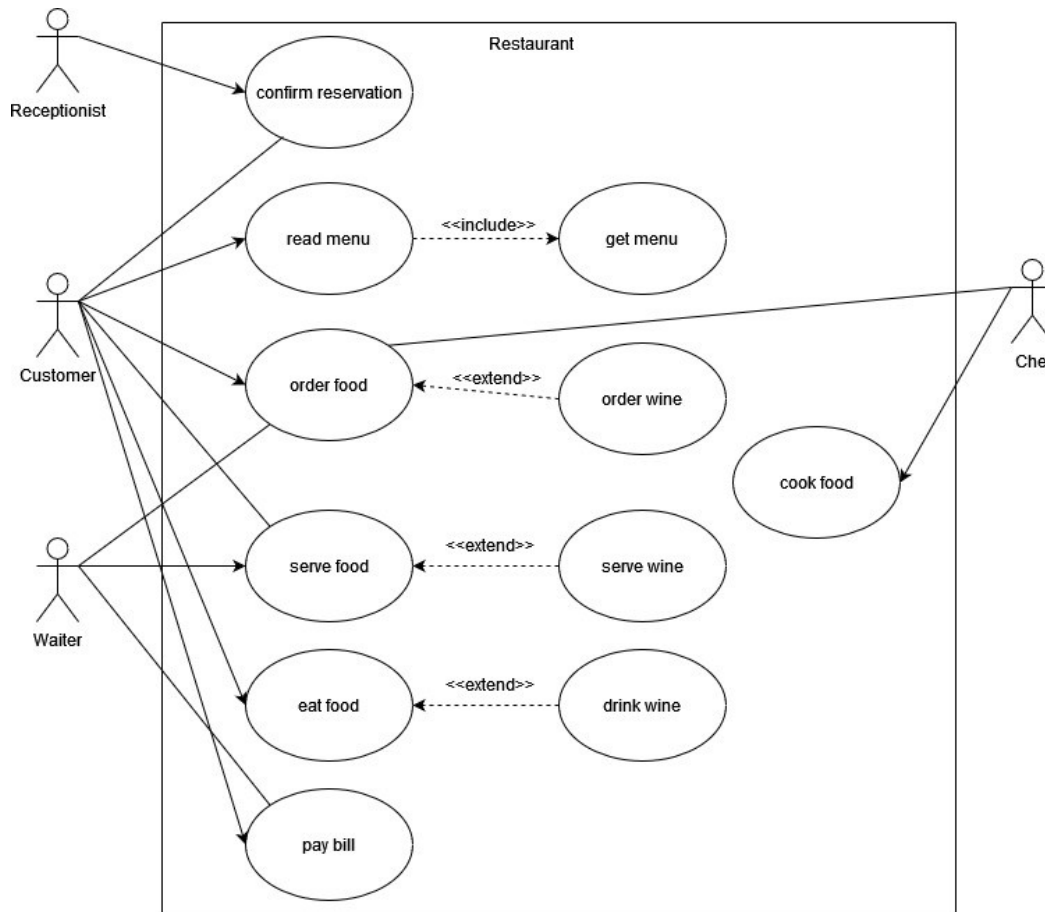


Figure 1 Example of a UML Use Case model

A UML Use Case model often contains additional information derived from the requirements, conditions that indicate the beginning and the end of a Use Case. The preconditions have to be true for a Use Case to begin and the post-conditions describe the state of the Actors and the System when the Use Case ends. Finally, a Trigger describes a time or a change event that triggers the start of the Use Case. These conditions are not part of the Use Case diagram, but are described in a Use Case template that supplements the diagram. For example, a Precondition for the Use Case “confirm reservation” would be, “Customer must make a reservation before visiting the restaurant.” A Trigger for the Use Case “read menu” is “Customer is seated”. Finally, a Post-Condition for the Use Case “serve food” would be, “The waiter has received the cooked food from the Chef and has served it to the Customer.”

A UML Use Case model consists of other components such as the Exceptions and the Alternative Flows but will not be described as, these detailed aspects of the UML Use Case model could not be addressed within the timeline of this research project.

2.2 User Stories

As the elicitation of requirements, when developing a new system, is a challenging task due to the ambiguity of natural language, companies have employed various ways to structure the requirements in a way that will facilitate the software development phase. One of the solutions, especially used in Agile methodology, is to build the requirements in a semi-

structured way, using a User Story template. The User Story template represents a requirement that is focused on the user and his/her interaction with the system [8]. A commonly used User Story template [9] has the following structure: *As a [type of User/Actor], I want to [interaction with the System/Use Case], so that I can [goal of the interaction/Post-Condition]*. A User Story example is: “As a Claims Administrator, I want to have access to the customer’s insurance data, so that I can properly evaluate the claim.”

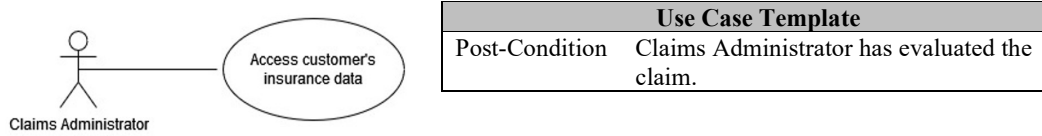


Figure 2 “Connextra” User Story modeled into a UML Use Case

Other User Story templates emphasise the added value of a requirement: “In order to [goal of the interaction/Post-Condition] as a [type of User/Actor], I want to [interaction with the System/Use Case].” [10], or provide a more detailed description of the requirement but utilising the “Five Ws” framework: “As [who][when][where], I [what] because [why].” [11].

2.3 Prose to Prototype project

The objective of this research project was to build a UML Use Case transformer as part of the Prose to Prototype wider project. P2P aspires to be a development tool that “provides automated support for synthesising UML models from requirements text expressed in natural language” [4] by combining state-of-the-art NLP and AI techniques, and a human-in-a-loop approach to tackle the ambiguity of the natural language. The synthesised UML models can be executed as *runnable prototypes* that allow end-users and domain experts to evaluate the system specification.

The development tool consists of several subsystems as shown in Figure 3: The NLP pipeline includes a *Speech to Text* component, which transcribes audio fragment that describe requirements, a *Text Condensation* component, which summarises superfluous textual requirements, a *Text Classification* component, which classifies text fragments into UML sub-model “buckets” and a *Specification Mapping* component that applies Part-of-Speech tagging and keyword analysis to the bucketed input, to generate UML specification models [4]. The UML Use Case transformer is a part of the *Specification Mapping* component which also includes a UML Class and a UML Activity modeller.

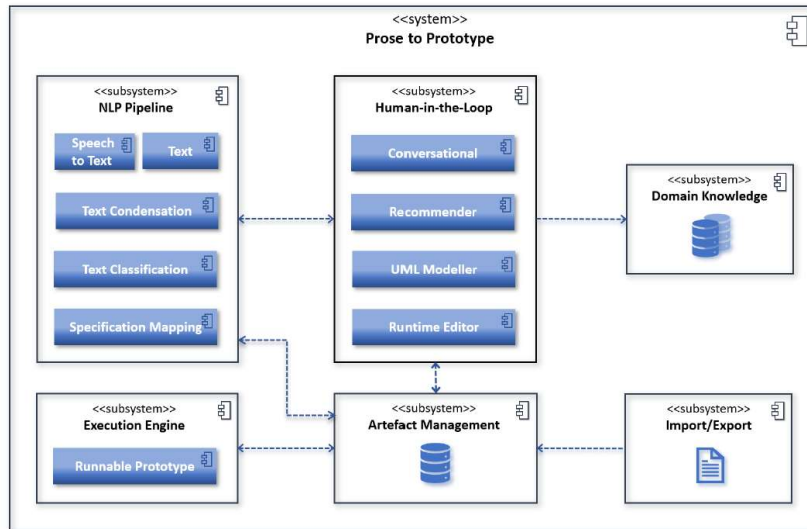


Figure 3 Prose to Prototype architecture

The Human-in-the-Loop subsystem which allows users to interact with the NLP pipeline consists of the following components: a *Conversational* component which enables users to converse with the system, a *Recommender* component which proposes instances that might interest the user, a *UML modeller* which allows the user to visualise and edit the generated models from a visual perspective and a *Runtime Editor* which allows the user to make changes directly to the UML model and consequently the running prototype [4].

Additionally, the *Domain Knowledge* component contains existing UML models and requirements texts organised by business and application domain, the *Artefact Management* component is used to store several artefacts such as UML models, requirement texts and code snippets and the *Import/Export* component allows the user import the specification models into other tools for application implementation [4].

2.4 Related Work

The challenge of reducing time and effort required in the first phases of system development has been the objective of many researchers that specialise in the Information Systems domain. Research focuses mainly on transforming text requirements, usually provided in a semi-structured form, into various analysis models, including UML Use Case models.

In 2009, Deeptimahanti & Babar developed the “*UML Generator from Analysis of Requirements*” (UMGAR) tool. The model builds UML models, such as Use-case diagrams, Analysis class models, Collaboration diagrams and Design class models from requirements texts. The generator is built using a rule-based approach by utilising various natural language processing tools, like the Stanford Parser [12] that was used to generate a parse tree for each requirement to extract UML Class and Use Case elements, Wordnet [13], an English language lexical database to perform morphological analysis and JavaRAP [14] that replaces pronouns with its correct noun form. UMGAR parses the extracted information generating XMI files that can be imported into suitable UML tools for visualising the generated models [15].

In 2012, More & Phalnikar presented a desktop tool called “*Requirement analysis to Provide Instant Diagrams*” (RAPID). RAPID’s architecture is based on UMGAR’s syntactic reconstruction rules for extracting information from requirements documents. OpenNLP POS tagger was used for the lexical parsing, while OpenNLP Chunkier, which chunks a sentence into phrases, was used for the syntactic parsing [16]. Domain ontology is being used to facilitate the performance of concepts identification [17].

In 2017, Narawita et al. proposed the “UML Generator” system, which generates use case and class diagrams from text requirements. The authors highlight the need of auto-generating UML based documentations to achieve cost and time reduction in the requirements analysis phase [18].

Yue et al. composed a systematic review of transformation approaches between user requirements and analysis models. They compare and evaluate 20 primary studies using a conceptual framework that provides common concepts and terminology [19]. Moving in the same direction, Osman et al. focus their research on literature works that use NLP techniques to transform textual requirements into visual models. Their study describes the different tools used in the information extraction process, as well as the issues of each proposed approach [3].

Ramackers et al. presented a vision of an automated development tool that creates UML models from textual requirements with the utilisation of machine learning and more specifically NLP techniques, while enabling human interaction with the system to further improve the generated UML models [4]. One of the first components of this tool was designed and developed by Tang, namely a UML Class Generator that receives functional requirements as text/audio input and transforms them into UML Class metadata [20].

Other studies focus on specific forms of user requirements, for example user stories. In 2017, Lucassen et al. showcased the extraction of conceptual models from User Stories by utilising the Visual Narrator tool, a component of the Grimm method. The Grimm method is used for requirements quality validation, elicitation and analysis. They separate each User Story into three parts: “*role, means and ends*” and use natural language processing heuristics to extract the conceptual models from the requirement texts [21].

In 2018, a process of automatically transforming User Stories into UML Use Case Diagrams using NLP Techniques was proposed by Elallaoui et al. For this purpose, TreeTagger parser was used for applying POS tags and categorising terms. The extraction algorithm creates new actor and use case elements which are then being transformed into a UML Use Case diagram [22].

2.5 NLP models architecture

As the aforementioned research projects rely on NLP heuristic rules to address the problem of transforming natural language requirements into UML models, we experimented with a different approach, namely using supervised learning to perform Named Entity Recognition and Relation Extraction on an annotated data set. By training models to recognise Actors, Systems, Use Cases and their relationships, we managed to create a pipeline that extracts this information from a requirements text and transform it into UML Use Case metadata.

To evaluate the performance of the trained models we compared the scores of three metrics: Precision, which calculates how many of the predictions the model categorised as positive were actually positive, Recall, which shows how many positives the model predicted compared to the actual positives and F1, which shows the harmonic mean between Precision and Recall.

The models that were used for training are part of spaCy’s library and their architecture is presented in the next subsection.

2.5.1 Transition-based Named Entity Recogniser

SpaCy’s overall framework for named entity recognition is rooted on a transition-based approach inspired by shift-reduce parsers, which was presented in the paper “*Neural Architectures for Named Entity Recognition*” by Lample et al. [23]. Instead of having each word as the object of interest and attach a tag to this word, the algorithm starts with having all the words on buffer and two empty stacks, the output stack and the stack that will contain each word in question. It then defines some actions that match the following transitions: a SHIFT transition where a word is moved from the buffer to the stuck, an OUT transition that moves a word from the buffer directly to the output stuck and a REDUCE(y) transition that pops all the items from the stack, labels them with the label y and moves this chunk of labeled words to the output stack [23]. An example of this approach is presented in Figure 4 that is included in the referenced paper.

Transition	Output	Stack	Buffer	Segment
	[]	[]	[Mark, Watney, visited, Mars]	
SHIFT	[]	[Mark]	[Watney, visited, Mars]	
SHIFT	[]	[Mark, Watney]	[visited, Mars]	
REDUCE(PER)	[(Mark Watney)-PER]	[]	[visited, Mars]	(Mark Watney)-PER
OUT	[(Mark Watney)-PER, visited]	[]	[Mars]	
SHIFT	[(Mark Watney)-PER, visited]	[Mars]	[]	
REDUCE(LOC)	[(Mark Watney)-PER, visited, (Mars)-LOC]	[]	[]	(Mars)-LOC

Figure 4 Transition sequence for Mark Watney visited Mars with the Stack-LSTM model [23]

SpaCy’s approach has an action that corresponds to the beginning move and fixes the label at the start of the entity and also their transition system matched the BILUO tagging scheme, because it discriminates better between different classes. The letters in BILUO stand for *Beginning, Inside, Last, Unit* and *Outside* respectively. SpaCy’s transition-based algorithm also assumes that the most important information regarding the identification of entities is close to the initial tokens, making it a bad fit for a task where the entities are long and the decisive tokens are in the middle of the span [24].

The statistical model that is used to predict the transitions is a combination of different neural network techniques that build the “*Embed, Encode, Attend, Predict*” framework [24].

The embedding task of the framework, shown in Figure 5, is to map long, sparse, binary vectors into shorter, dense, continuous vectors in an embedding table using “*one hot*” encoding. Word embeddings are used because it makes it easier to perform similarity operations and feed them forward in a neural network. The vectors are relatively short, ranging from 64 to 300 units long. Word embeddings are used because it makes it easier to perform similarity operations and feed them forward in a neural network.

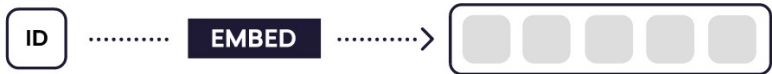


Figure 5 Embedding step [25]

The first step in word embedding is the “*doc2array*” procedure, where four attributes are extracted from each token in a document: an ID for the normalised form of the string, the prefix, the suffix and a word shape feature that replaces all the digits with the letter “*d*”, the lowercase characters with “*w*” and the uppercase characters with “*W*”.

After the feature extraction stage, a matrix with four numerical columns is created where each row is a word in the document. To embed each of these columns into a table a “*hashing trick*” is used that is called “*Bloom embeddings*”. Instead of having a fixed inventory for all the known words in the embedding table and only one out-of-vocabulary vector, each word is represented by the sum of four different hashes, so the vast majority of the words will end up with unique representations.

The result is the embedding table that consists of a separate embedding for each of these features that are then concatenated together using four functions. The concatenated input is fed forward to a multi-layer perceptron that consists of one hidden layer and a maxout activation function. The result is a 128-dimension vector per word that takes into account sub-word features and is able to learn an arbitrarily-sized vocabulary.

The encoding task of the framework (Figure 6) deals with the sequence of vectors, as the linear order of words is very important. To make the word representations context-specific a sentence matrix is being used which consists of the dependent vectors. Each row of the matrix represents the meaning of each word in the context of the rest of the sentence [24].

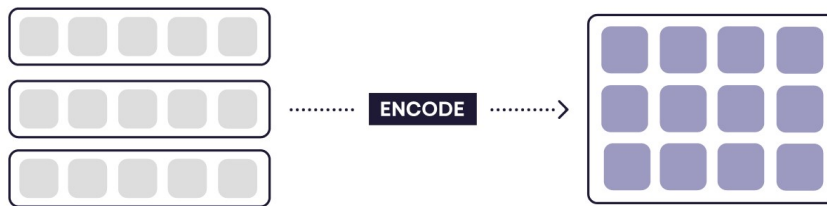


Figure 6 Encoding step [25]

Instead of using the traditional long-short memory recurrent neural network (BiLSTM), spaCy uses a convolutional neural network (CNN) to perform this operation based on the work of Collobert et al. in their publication “*Natural language processing (almost) from scratch*” [26]. The fundamental building block is a trigram CNN layer which takes a window on either side of the word and concatenates them together. As mentioned above, each word is represented by a 128-dimension vector, so after taking into account the neighboring words, a vector with 384 is created. Then, a multi-layer perceptron is used to map that representation into 128 dimensions. The result is an output vector that includes information about the target word and two words, one from each side of the target word, that has the same dimensionality, in order to relearn the meaning of the target word based on its neighbors. By continuing stacking this process, at the fourth layer, information is drawn by potentially four words on either side, thus draw information about the word’s vector based on its surrounding context, without taking into account the whole document. Lastly, residual connections are used to the output of each of these convolutional layers, so that the output space of each of the convolutions is similar to the output of the input, in order to roughly preserve the original input representation [24].

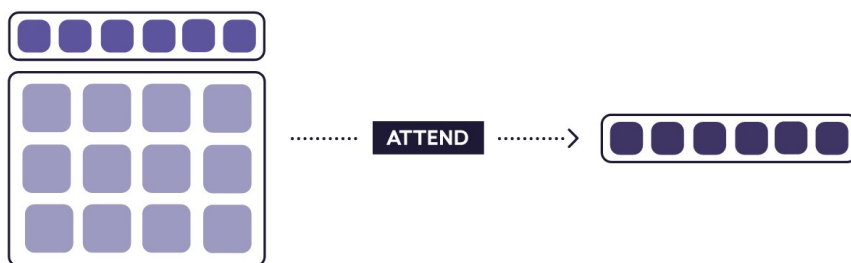


Figure 7 Attention step [25]

The third model is an attention layer, shown in Figure 7, that takes as an input the matrix representation previously produced in the encoding step, and reduces it into single vector, creating a state that will be passed onto a standard feed-forward network for the prediction task. The attention model manually extracts features with a translation layer into the hidden layer. It takes an input query vector for each word in the sentence and learns a weighted summary of the word in the buffer, the word before it, the first and last words of the previous entity and the last word of the entity before that. The features considering the previous entities can be arbitrarily far back in the document, in comparison to a CRF model which is bounded in the number of previous decisions that is conditioned on.

Finally, after calculating the features for the state, a multi-layer perceptron is used to get the action probabilities (Figure 8). Then, a procedure checks which actions are valid given the state and decides with is the best valid action to perform.



Figure 8 Prediction step [25]

A cost, which represents the number of new errors that will occur if this action is taken, is assigned to each action. If the predicted action is not zero-cost, the weights are updated, so that in the future this particular action will cost more and the best zero-cost action will score higher. By making the predictions this way, the algorithm will always choose the action that scores higher when dealing with a particular state. The scores are not scaled, so they cannot reflect the wider parse quality and cannot be used to obtain confidence scores and set thresholds. [24].

After making a prediction, the algorithm then moves to the next state, proceeding forward in the loop, until there are no states left in the buffer. Figure 9 shows the pseudocode of the loop.

```

tensor = trigram_cnn(embed_word(doc))
state_weights = state2vec(tensor)
state = initialize_state(doc)
while not state.is_finished:
    features = get_features(state, state_weights)
    probs = mlp(features)
    action = (probs * valid_actions(state)).argmax()
    state = action(state)

```

Figure 9 Pseudocode of the overall parsing loop [27]

Besides the previously described models, a listener is used as a sublayer to pass the predictions from the Tok2Vec components into the ner component and to communicate the gradients back upstream. The listener works by caching the Tok2Vec output for a given batch of Documents [28].

2.5.2 Named Entity Recogniser with RoBERTa-base transformer

This pipeline uses a transformer model combined with the transition-based Named Entity Recogniser with the use of a TransformerListener layer instead of using the Tok2Vec component and Tok2VecListener sublayer. The advantage of transformers compared to alternatives like CNN or LSTM is that they scale up better when it comes to adding more parameters.

The transformer model was first introduced in the paper “*Attention is All You Need*” by Vaswani et al [29]. The model consists of the same number of encoders and decoders and all encoders are identical in structure.

As shown in Figure 10, each decoder contains two sublayers, a self-attention layer and a feed forward neural network. Each input sequence flows first through the self-attention layer, which is responsible for looking at other words in the sentence, facilitating a better representation of the word. Then, the output of the self-attention layer is fed to the feed-forward neural network. At the bottom encoder the embedding of the input sequence occurs and each embedded word is fed in the layers of the encoder through its own path. To preserve the order of embedded words, the transformer has a vector to each input embedding that follows a pattern the model learns. Additionally, each sub-layer of the encoders has a residual connection and a layer normalisation operation [30].

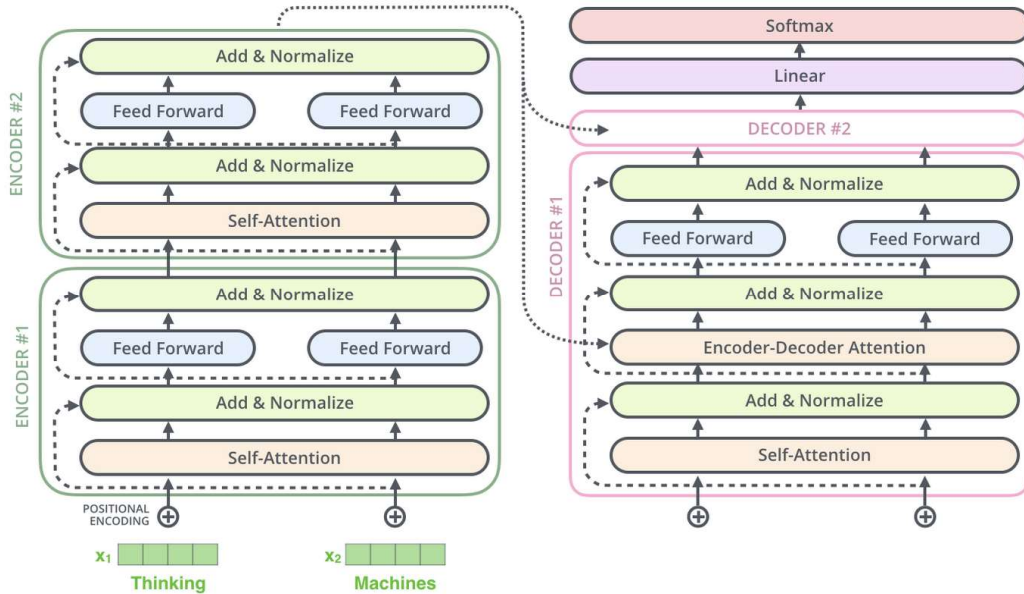


Figure 10 Transformer with two stacked encoders and decoders [30]

The decoder consists of these two sublayers but in addition, between them there is an extra attention layer that assists the decoder to recognise the important information of the input sentence. The output of the top encoder is transformed into attention vectors and is initially fed to the encoder-decoder attention sublayer and then flows to the other sublayers. Finally, a

linear layer which is a fully connected neural network, creates a much larger vector, called logits vector, by projecting the vector produced by the decoders. At the logits vector each cell corresponds to the score of each unique word. Then, the SoftMax function turns the scores into probabilities and the cell with the highest probability is chosen [30].

SpaCy's transformer-based pipelines use the [Hugging face Transformers](#) library and PyTorch. For the English transformer pipeline, the default model used is the RoBERTa-base model published by researchers at Facebook [31]. The model is based on Google's BERT model [32] and modifies key hyperparameters by removing the next-sentence pretraining objective. It is trained with much larger mini-batches and learning rates and is pretrained using self-supervised learning on a large corpus of English data [33]. The model is intended to be fine-tuned on downstream tasks like NER and it works best on tasks that use the whole sentence to make predictions. Before adding the transformer model in the pipeline, spaCy uses their machine learning library Thinc which works as interface layer between spaCy and other machine learning libraries. For example, when using the HuggingFace transformers library, Thinc wraps up their PyTorch models so that they can be plugged into a spaCy component and behave the same as models developed by spaCy.

In general, although transformer-based pipelines have more dependencies and run on GPU which is more expensive and less reliable, they greatly improve the results of various natural language processing tasks.

2.5.3 Span categoriser

The span categoriser is an experimental spaCy component, that was developed to provide better predictions in cases the entities are phrase sentence fragments and not token-based tags or there is a label overlap. Moreover, the named entity recogniser assumes that the most informative words are close to their starting tokens, while the span categoriser uses the full context of a span to learn its task [34].

The span categoriser can be divided into two parts: the suggester and the classifier, as shown in Figure 11. The suggester is a function that extracts span candidates from the input text, that may or not overlap, and feeds them to the classifier. Suggester functions can be written manually and can be completely rule-based depending on annotations from other components or the default built-in n-gram suggester functions can be used, in which the n-gram sizes to get suggested for every extracted span can be defined. By manually writing suggester functions, the model can be biased towards precision or recall, depending on the use case [34].

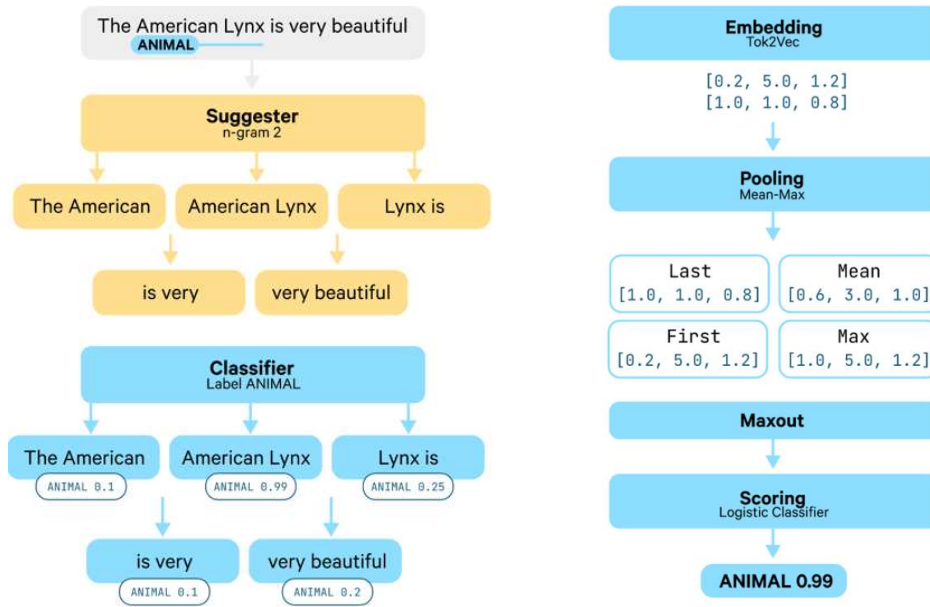


Figure 11 The Spancat architecture and Spancat's classifier [34]

The classifier takes as an input the suggested spans and predicts the probability for each label. It consists of three layers: The embedding layer, where the tok2vec representation of the respective span is obtained, the pooling layer, where the sequences are reduced to make the model robust and the context is encoded using a window encoder, the Scoring layer, where multilabel classification is performed on the pooled spans and model predictions and label probabilities are returned [34].

Unlike the named entity recogniser, the span categorisation model predicts label probabilities over the whole span, allowing access to confidence scores to threshold against.

2.5.4 Relation extractor

The relation extractor is based on a binary relation extraction method that examines two entities in a document and determines if these entities are related and if they are related, the type of relation that links them [35]. The model is built using the machine learning Thinc library and takes a document as input and outputs a two-dimensional matrix of the predicted relations. This model is then used to power a pipeline is implemented that translates the predicted scores into annotations. The architecture of the relation extractor is shown in Figure 12.

The first layer of the model transforms each document into a list of tokens and includes an embedding layer that can either be a Tok2Vec component or a Transformer. A pooling layer summarises the token vectors into entity vectors, as entities can consist of multiple tokens. A method then generates pairs of entities that will be classified as being related or not. The two entities have to be within a predefined maximum distance of each other, in order to be considered for relation classification.

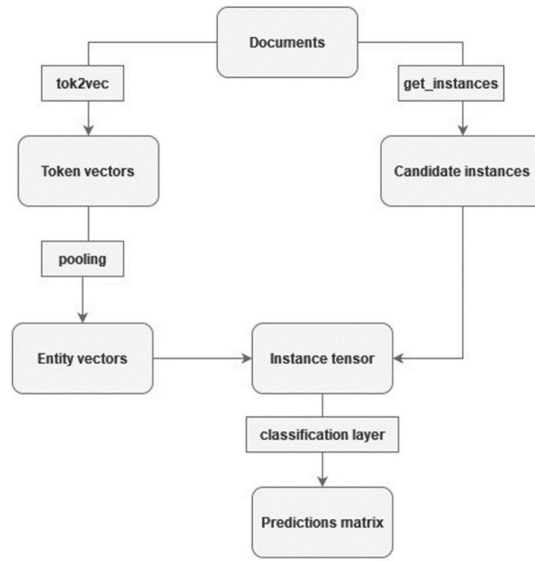


Figure 12 Model architecture of the Relation Extractor

By reducing the maximum distance of two entities the model considers for pairing, fewer instances will be classified, resulting in an increased precision rates and decreased recall rates. As the binary relations between the two entities are directed, two instances are created, one where the first entity is the subject and the second is the object and a second where the first entity is the object and the second is the subject. For each instance, the vectors of the two corresponding entities are concatenated into one larger tensor that will be the input of the classification layer [35].

The Tok2Vec layer can be replaced by a Transformer layer, that will include a pretrained model from the HuggingFace library. The replacement and utilisation of a transformer in the relation extraction, is similar to the process of the named entity recognition task.

The classification layer transforms the instance data to the matrix holding the final predictions for each instance and each relation label. It is a linear layer that is followed by a logistic output activation to ensure that the predictions are within the $[0,1]$ interval [35]. During training the created predictions will be compared to the gold-standard data to calculate the loss and the gradient of loss that will be used to update the weights of the model through backpropagation. Once the model is trained, its performance can be calculated on a set of left-out examples, using the precision, recall and F1 metrics.

3 Data

Chapter 3 is dedicated to presenting the data set that was used to train the models. In the first part, we describe the sources that we used to compile the data set. In the second part, we describe the annotation procedure and provide statistics regarding the annotated samples.

The first step in developing a reliable UML Use Case modeller using supervised learning is to ensure that the input data are representative of the various formats in which user requirements are written. When developing a new system, the process of requirements analysis and elicitation occurs in the early stages of the Software Development Life Cycle. During the requirements analysis phase, business experts and senior stakeholders participate in workshops in order provide information regarding the expected functionalities of the developed software [2]. This information is provided in natural language, in a textual or vocal form. Next, in the requirements definition phase, the collected information is documented, usually in the form of a Software Requirement Specification document (SRS) or a Business Requirement Document (BRD). As many companies nowadays use agile methodology when developing a new system, documented user requirements are transformed into User Stories to fit in multiple short sprints.

As Software Requirement Specification documents, Business Requirement Documents and User Stories are part of a company's strategic planning, they are rarely disclosed publicly, thus obtaining software requirements is a challenging task. For the purposes of this research project, multiple sources of requirements have been utilised: existing requirements data sets, user stories data sets, and real-life samples presented in Software Engineering and UML modelling books. The focus, when gathering the data, was on finding user requirements that describe interactions between users and systems and can be modelled as UML Use Cases.

3.1 Data Collection

In this section the composition of the UML Use Case data set is described, along with information regarding existing requirements repositories.

3.1.1 Public Requirements Data Set (PURE)

That first data set utilised was PURE. PURE (Public Requirements dataset) is a popular dataset, used in requirements engineering for natural language processing tasks [36]. The original dataset consists of 79 publicly available requirements documents, of which 15 requirements documents were included in the UML Use Case dataset. These 15 documents were split in 80 smaller samples to facilitate the annotation procedure. Each of these 80 examples consists of multiple sentences which include use case elements such as use cases, actors, systems and their respective relationships that were annotated and used as input for the named entity recognition (NER) and relation extraction (RE) tasks. Furthermore, many of these 80 samples contain phrases, indicative of triggers, post-conditions and preconditions, which were considered when creating the heuristic rules for these specific use case elements.

3.1.2 User Stories Data Set

The UML Use Case dataset is also composed of 1,618 user stories derived from a requirements dataset compilation published by Dr. Fabian Dalpiaz. The original dataset is comprised of 22 requirements documents and each document contains more than 50 user stories. These requirements documents were either published online or retrieved by software companies [37]. User stories in the dataset are presented in their typical agile structure: “*As a [persona], I [want to], [so that].*” The elements relevant to use cases that were annotated in these user stories are *actors*, *systems* and *use cases*, as well as *interact* and *include* relationships. Also, the goal of the user story, which is expressed in the second part of the sentence can be modeled as a *post-condition* according to the UML Use Case standards.

By semi-automatically modifying the original user stories, 1,600 user requirements were added in the UML Use Case data set. More specifically, we deleted the phrases “*As*” and “*I want to*” and we converted the pronouns from first person to third person with the implementation of a python script. We then checked the modified sentences using Grammarly, to identify and correct the verbs. Below is an example of how the changes were implemented:

Original User Story: “*As a dataset developer, I want to have an archetype that helps me package my dataset type properly.*”

Edited user requirement: “*A dataset developer has an archetype that helps him or her package his or her dataset type properly.*”

3.1.3 Tera-PROMISE

OpenScience’s Tera-PROMISE is another popular software engineering research data repository that includes data sets regarding functional and non-functional requirements, source code analysis and metrics, refactoring, and effort estimation [38]. UML Use Case data set includes 131 examples from the Tera-PROMISE repository. Each example is 1 to 3 sentences long and consists mostly of actor, system and use case entities suitable for the NER and RE tasks.

To further enrich the UML Use Case data set, we added 75 requirements documents that were retrieved from various Software Engineering books and online sources. Table 1 lists these sources.

Author	Title
Alistair Cockburn	Writing Effective Use Cases [39]
Ghinwa Jalloul	UML by Example [40]
Kurt Bittner & Ian Spence	Use Case Modeling [41]
Petra J. Papagiorgji, Panos M. Pardalos	Software Engineering Techniques Applied to Agricultural Systems - An Object-Oriented and UML Approach [42]
Timoth Lethbridge & Robert Laganier	Object-Oriented Software Engineering - Practical Software Development using UML and Java [43]

Bernd Bruegge & Allen H. Dutoit	Object-Oriented Software Engineering using UML, Patterns and Java [44]
Frank Armour & Granville Miller	Advanced Use Case Modeling, Volume One - Software Systems [45]
INSPIRE Knowledge Base	Use case “INSPIRE Harmonisation of Energy Performance Certificates (EPC) datasets [46]
Zahra Abdulkarim Hamza & Mustafa Hammad	Generating UML Use Case Models from Software Requirements Using Natural Language Processing [47] Title of the original document: Mental Health Care Patient Management System

Table 1 Software Requirements Documents

The collected data were added in a JSONL file, with each line representing a document and the end of a line in a certain document being indicated with the character “\n”.

3.2 Data Annotation

3.2.1 Annotation tools

For the purposes of the research project, the annotation tool to be used should provide the following functionalities:

- Named Entity Annotation
- Relationship Annotation
- Export IOB/BILUO Labels
- Allow nested and overlapping labeling

Doccano [48] is a user friendly, open-source annotation tool, mostly used for Named Entity Annotation. Although overlapping annotation is possible with Doccano, Relation annotation is work in progress and installation of Doccano transformer is needed for exporting IOB labels. Label Studio [49] is another free-to-use annotation tool with many functionalities, which although allowed annotating relationship between entities, the relations were not visible in the exported files. Other annotation tools, like Universal Data Tool [50], Brat [51], TagTog [52] were also tested but failed to meet some the forementioned prerequisites. Table 2 shows information regarding the annotation tools.

Doccano	Label Studio	Universal Data Tool	Prodigy
Requires installation	Requires installation	Online → requires setup every time or can be installed	Requires installation
Very user friendly, no data or programming skills needed	Easy to use, no data or programming skills needed	Not so easy to use, some programming skills needed	Scriptable, programming skills required

Open Source-Free to use	Open Source-Free to use	Free to use	Need to purchase lifetime license
Overlapping annotation possible	Overlapping annotation not possible	Overlapping annotation possible	Overlapping annotation possible with span categorisation
Relation annotation not available yet	Relation annotation is possible, but does not appear in the exported files	Relation annotation possible	Relation annotation possible Co-Reference annotation possible
Exports only in JSONL, needs doccano transformer for IOB scheme	Exports in multiple formats, including Conll	Exports in its own UDT format – requires transformation	Imports and Exports in JSONL format IOB can be retrieved

Table 2 Review of four annotation tools

Prodigy [53] is a scriptable annotation tool that offers functionality for annotating Named Entities, Relations, Text Classification and Image labels. Overlaps are not allowed in Named Entity annotation, but are feasible in Span Categorisation. Moreover, as Prodigy is developed by the same team that created spaCy [24], a free open-source library for Natural Language Processing, it provides various functions that help with the model training. For example, prodigy has a build-in function that shuffles and splits the data into training and validation sets, while at the same time transform the data into the binary spaCy format, required for training. Annotated datasets are being stored in the database using SQLite and can be directly imported for training, or they can be exported as JSONL files, in case further processing is needed. Each annotation example, is a dictionary that contains information about the samples like the text, the entity spans and the labels.

3.2.2 Annotation guidelines

Detailed and case specific guidelines were carefully drafted with the guidance of a UML and an NLP expert, to ensure the credibility of the annotation procedure. In addition, the annotation of complex or ambiguous documents was examined separately together with the UML expert to increase reliability. Table 3 lists the labels used for NER and RE tasks, as well as their definitions.

	Label	Definition
Named Entity Recognition	ACTOR	Type of role played by an entity that interacts with the system, described with a noun or noun phrase. It can be a person, an organisation or another system that exists out of the system boundary.
	SYSTEM	Name of the system the actors interact with. In many cases the annotated word is the word “ <i>system</i> ”.
	USECASE	A verb phrase that specifies how the Actors interact with the system.

	INTERACT	Association between the Actor and the Use Case. The relationship arrow begins from the Actor and points to the Use Case. In case, there is a System and a Use Case, the arrow has the opposite direction.
Relation Extraction	INCLUDE	Relationship among Use Cases, when a base Use Case is not complete in itself but dependent on the included use case to be meaningful and complete. The arrow starts from the base use case and points to the including use case.
	EXTEND	Relationship among Use Cases, when there is some additional behavior that should be added, possibly conditionally, to the behavior defined in one or more base Use Cases. The arrow starts from the extending use case and points to the extended use case.

Table 3 Labels and Definitions for NER and RE

We built the general annotation guidelines based on the following rules:

- Read the document as a whole before labeling in order to comprehend the category of each entity, because the same phrase/word could be interpreted differently based on the context. For example, “*Bank System*” can be an external Actor that interacts with the System or the System itself.
- The annotator will not proceed in any spelling/grammar error corrections, because the annotated documents must resemble the actual data that are bound to have errors.

Furthermore, we specified a series of Named Entities Annotation, Span Categorisation and Relation Extraction Rules to ensure consistency while labeling. A detailed list of the annotation guidelines can be found in the Appendix A.

3.2.3 Annotation process

The annotation task was performed by one person and the collected data were labeled with the use of three prodigy recipes: ner for the Actors and the Systems, spancat for the Use Cases and rel for the Interact, Include and Extend Relationships.

To start with NER annotation, we needed to indicate the prodigy recipe, load a spaCy pipeline for tokenisation, create a new name for the dataset, write the path to the file that has the input text and provide the labels [53]. After typing the command, prodigy creates the two labels, and the new dataset to database SQLite and starts the web server at the local host. The interface of the app is presented in the Figure 13.

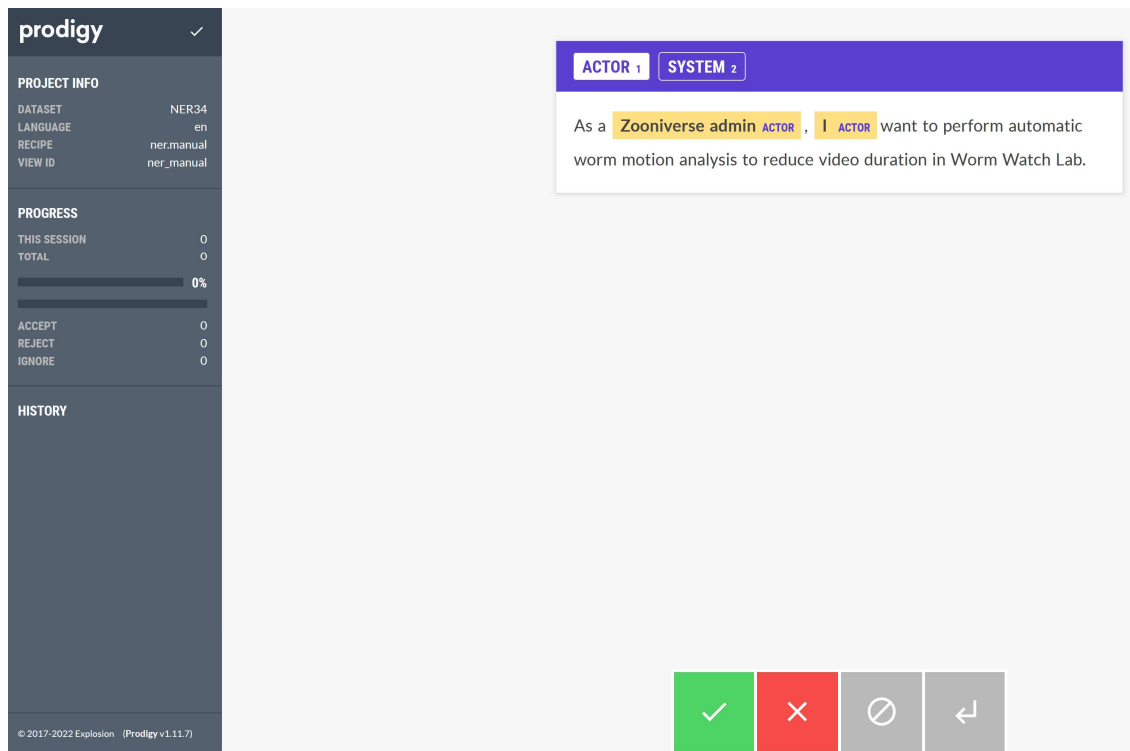


Figure 13 prodigy interface

As Use Cases have less consistent boundaries and mixed lengths, prodigy suggested using the span categoriser component for annotating and training, instead of named entity recognition. Moreover, span categorisation allows overlapping, a useful feature, as in many cases an Actor or a System are also included in the Use Case. Figure 14 summarises the differences between the Named Entity Recognition and the Span Categorisation components.

Named Entity Recognition	Span Categorization
spans are non-overlapping syntactic units like proper nouns (e.g. persons, organizations, products)	spans are potentially overlapping units like noun phrases or sentence fragments
model predicts single token-based tags like B-PERSON with one tag per token	model predicts scores and labels for suggested spans
takes advantage of clear token boundaries	less sensitive to exact token boundaries

Figure 14 NER vs Span Categorisation [53]

In this annotation scheme Actors and Systems which were imported from the NER data set are represented as spans as shown in Figure 15.

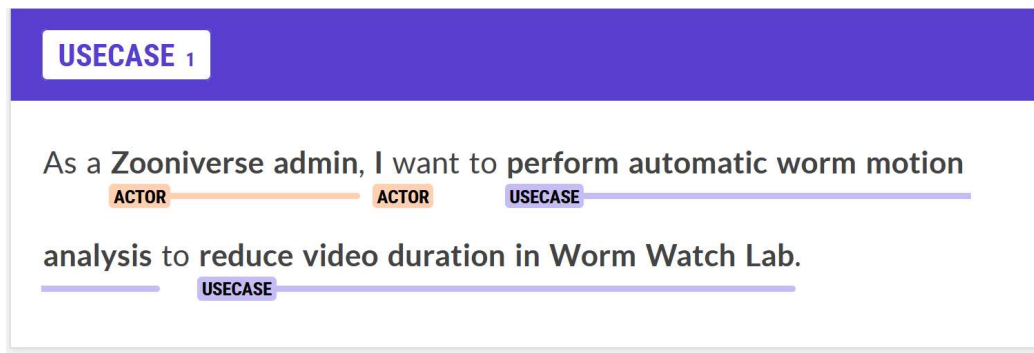


Figure 15 Span Categorisation annotation

Finally, for the Relation Extraction task, the rel.manual recipe was called and three new labels were created. For this task we used the span data set that includes Actor, System and Use Cases labels. By clicking on the first entity, the head is indicated and by clicking on the second entity the child is indicated. Head and child are recognised based on the direction of the arrow.

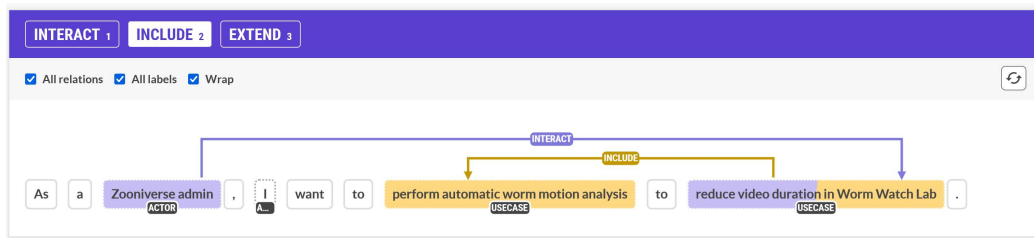


Figure 16 Relation extraction annotation

The annotated dataset can be exported in a JSONL file. Each document is written in a new line that includes the following information:

First, the original text is presented.

```
{"text": "As a Zooniverse admin, I want to perform automatic worm motion analysis to reduce video duration in Worm Watch Lab.", "_input_hash": 1088849660, "_task_hash": 955379237, "_is_binary": false,
```

Then, each token and their position in the text.

For example, the phrase “*As a Zooniverse admin*” receives the following annotations:

“tokens”:

```
[{"text": "As", "start": 0, "end": 2, "id": 0, "ws": true, "disabled": false}, {"text": "a", "start": 3, "end": 4, "id": 1, "ws": true, "disabled": false}, {"text": "Zooniverse", "start": 5, "end": 15, "id": 2, "ws": true, "disabled": false}, {"text": "admin", "start": 16, "end": 21, "id": 3, "ws": false, "disabled": false}
```

After that, only the tokens that were labeled and, their token position in the text and the position of the first and last character of the spans are shown.

```
"spans": [{"text": "Zooniverse admin", "start": 5, "token_start": 2, "token_end": 3, "end": 21, "type": "span", "label": "ACTOR"}, {"text": "I", "start": 23, "token_start": 5, "token_end": 5, "end": 24, "type": "span", "label": "ACTOR"}, {"text": "perform automatic worm motion",
```

analysis", "start": 33, "token_start": 8, "token_end": 12, "end": 71, "type": "span", "label": "USECAS E"}, {"text": "reduce video duration in Worm Watch Lab", "start": 75, "token_start": 14, "token_end": 20, "end": 114, "type": "span", "label": "USECASE"}], "answer": "accept", "_timestamp": 1652366354,

Last, the relations between the entities, with the span and character positions of the head and the child are indicated.

"relations": [{"head": 3, "child": 20, "head_span": {"start": 5, "end": 21, "token_start": 2, "token_end": 3, "label": "ACTOR"}, "child_span": {"start": 75, "end": 114, "token_start": 14, "token_end": 20, "label": "USECASE"}, "color": "#c5bdf4", "label": "INTERACT"}, {"head": 20, "child": 12, "head_span": {"start": 75, "end": 114, "token_start": 14, "token_end": 20, "label": "USECASE"}, "child_span": {"start": 33, "end": 71, "token_start": 8, "token_end": 12, "label": "USECASE"}, "color": "#ffd882", "label": "INCLUDE"}]}

3.2.4 Annotation statistics

The tables in the next page present information regarding the data set and its annotations. As the word count was calculated using the JSONL files, the word “*text*” has been subtracted from the word count. Also, in the character count we excluded white spaces and the characters {“*text*”}.

Each row of the first table shows the number of documents, sentences, words and characters for each source. Although the PURE data set consists only of 80 documents, it has more sentences and more words than the user stories data sets. The reason is that each document of the PURE set provides a complete user specification regarding a system, while each document in the user stories data sets is just one user requirement, hence one sentence. Also, as it can be observed in the second table, PURE data set offers more valuable information regarding UML Use Cases and includes more complex relations between the UML Use Case elements.

Finally, by observing the data in Table 4, it can be safely assumed based on the size of EXTEND labels that the model will not be able to learn this label.

Textual Information				
Source	Documents	Sentences	Words	Characters
PURE	80	3,006	44,161	228,841
Original User Stories	1,618	1,618	39,199	181,044
Transformed User Stories	1,638	1,638	36,179	174,288
Tera-PROMISE	131	167	2,971	15,877
Miscellaneous	75	660	11,084	55,778
Total	3,542	7,027	133,594	655,828

Annotation Statistics						
Source	ACTOR	SYSTEM	USECASE	INTERACT	INCLUDE	EXTEND
PURE	1,994	1,074	1,948	1,816	85	5
Original User Stories	4,680	229	1,784	1,850	29	1
Transformed User Stories	3,106	242	1,820	1,940	37	2
Tera-PROMISE	227	92	196	272	4	0
Miscellaneous	772	376	712	793	5	1
Total	10,779	2,013	6,460	6,671	160	9

Table 4 Textual Information and Annotation Statistics

4 Methods

The main objective of this research project was to develop a UML Use Case transformer that receives a requirements text as input and extracts information that can be used to create a UML Use Case model. In this chapter various natural language processing methods and the libraries used for developing the UML Use Case transformer are presented. In the first section we describe the implementation of information extraction techniques regarding Use Case elements and their relationships, by training models using supervised learning. In the second section we present the methodology of extracting information relevant to the starting and ending conditions of a Use Case based on heuristic rules. Figure 17 shows the various components and methods that were used to build the UML Use Case Transformer.

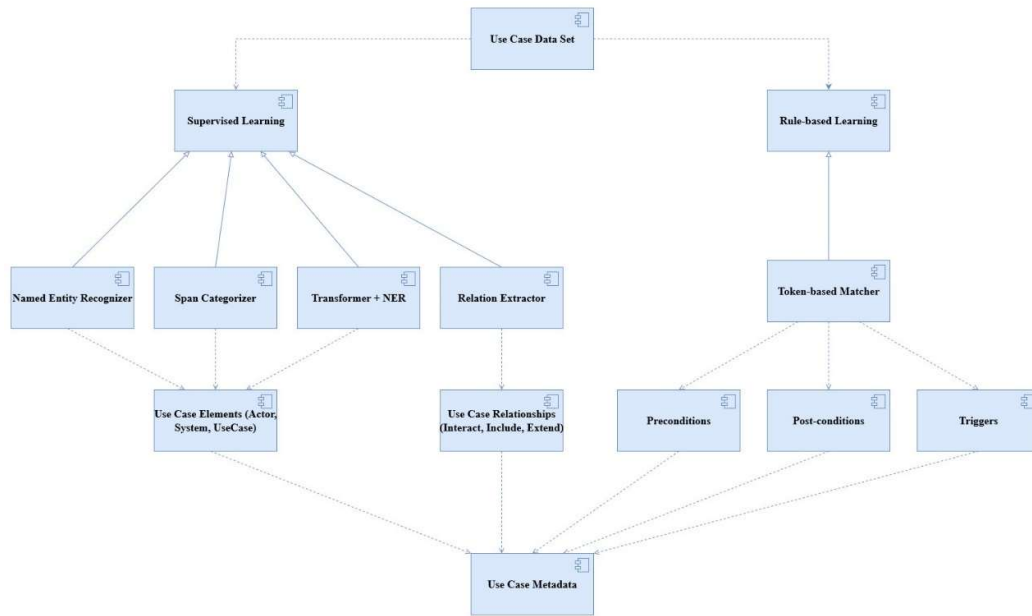


Figure 17 Overall scheme of the methods used to build a UML Use Case Transformer

4.1 NLP Supervised Learning techniques

As most of the work related to developing UML models using NLP techniques is focused on extracting information with heuristic rules, we focused our efforts on extracting the Use Case elements *Actor*, *System* and *Use Case* and their relationships using supervised learning based on the annotated data sets we created. Models that yielded the best evaluation scores with this data set were used to assemble a pipeline that produces UML Use Case metadata from requirements texts. To test the overall performance of this pipeline we performed an extra evaluation using out-of-sample data, namely requirement example documents randomly selected from various sources.

For the extraction of the Use Case elements, we trained several transition-based named entity recognisers with different sets of data and compared their performance. More specifically, we trained models to predict Actor/System and Use Case labels separately and together, to check the impact in the performance of words that should be identified as Actors but were included in the UseCase labels. We also performed two different splits of the data set, 70% and 80%

for training data and compared the results. Finally, we trained models using only specific documents of the annotated set to check if there is an increase in performance. The architecture of the transition-based named entity recognisers is described in Chapter 2.

To improve the extraction of the Use Case element, as the Use Case elements tend to consist of more than three tokens, we also trained a span categoriser that can handle longer phrases in comparison to a named entity recogniser.

Additionally, we experimented with a pretrained transformer, which replaces the traditional embedding layer for transforming the words into vectors, and is used in conjunction with the NER and Span Categorisation components. While training these models, we used the same methods as the ones when training the transition-based named entity recognisers: train with Actor/System and Use Case labels separately and together and use specific chunks of the data set.

For the extraction of the relationships between the Use Case elements, we trained a relation extractor with supervised learning using the annotated relations of the Use Case data set. To train the relation extractor, the named entities must be known to the model, as it only predicts relations between these named entities. The named entity recogniser included a transformer component, to maximise the prediction accuracy of the named entities. As the relation extractor we utilised assigns a head and tail label to the entities, suggesting direction, we included specific rules when creating the annotation guidelines to ensure that the relation labels are representative of the Use Case elements relations.

To evaluate the performance of the trained models, we calculated precision, recall and F1 scores overall and individually for each label. We conducted a separate evaluation of partially successful predictions, in order to take into consideration cases where the prediction was partially correct. The reason for performing this evaluation, particularly for the Use Case label, was that the starting or the ending tokens were likely to not be predicted correctly by the model, leading the algorithm to treat the whole entity as a missed label, while in reality the important information that characterises a Use Case was correctly predicted.

4.1.1 Implementation and Libraries

The Use Case information extraction pipeline was implemented with the spaCy library. SpaCy is a free open-source library, that provides many natural language processing tools and build-in models like a Tokeniser, a Named Entity Recogniser and a Text Classifier [24]. This library was chosen because it is considered to be faster than other libraries and more efficient in handling large amounts of text data as it is written in Cython. It is also compatible with the annotation tool used to label the Use Case dataset, as it is developed by the same team.

For training and evaluating the models, the documents of the Use Case data set were shuffled and split in training and test sets using prodigy’s functionality, which additionally transformed the JSONL data files into spaCy’s binary format. This format serialises a DocBin, which contains a collection of Doc objects. Each Doc object is a sequence of Token objects and their annotations. By transforming the input data into their binary format, spaCy pipelines are trained using the same format they output. Moreover, DocBins produce small data sizes, which can be more efficiently stored and decrease training time [54].

At the beginning of each training process, a configuration file was created, the single source of truth for training. The config unifies several workflows and it includes all settings and hyperparameters needed to train a pipeline. [spaCy](#) provides a widget that generates an initial configuration with the recommended settings based on the components and the hardware that will be used, as well as an optimisation option for efficiency or accuracy, as shown in Figure 18. For our experiments, we optimised for accuracy, to ensure a better performance with the trade-off a larger and slower model. Regarding the hardware, we used CPU when training the transition-based recognisers and GPU when training the transformers and the relation extractor.

The screenshot shows a web-based configuration interface. It has a light gray background with white sections. The first section is 'Language' with a dropdown menu set to 'English'. The second section is 'Components' with a question mark icon and seven checkboxes: 'tagger', 'morphologizer', 'trainable_lemmatizer', 'parser', 'ner', 'spancat', and 'textcat'. The third section is 'Hardware' with two buttons: 'CPU' (highlighted in blue) and 'GPU (transformer)'. The fourth section is 'Optimize for' with a question mark icon and two buttons: 'efficiency' (highlighted in blue) and 'accuracy'.

Figure 18 QuickStart widget for generating a starter config

The initial config file can then be edited to tune the parameters and fit the specific requirements of a training session. The config system supports registered functions, which are retrieved from an extensible table, called the registry. In Appendix B we present an example of the configuration file.

We performed the training sessions in various environments, more specifically, we used Jupyter Notebook, Google Colab and the desktop’s terminal, depending on each model’s prerequisites. For each model we saved the best and the last performer and compared their predictions.

4.2 Rule-based matching

To extract information relevant to Preconditions, Post-Conditions and Triggers, we used rule-based matching. Key words that indicate the existence of these conditions were identified and were used to define token-based patterns, to match these words in a requirements text and extract the relevant phrases. These rules have been created with the use of spaCy’s rule-matching engine called the Matcher. This tool matches sequences of tokens in a document, based on pattern rules. Each pattern that is created and added to the Matcher, consists of a list of dictionaries and each dictionary describes one token and its attributes [55].

When the algorithm runs, it searches to match a specific phrase in the document with the key phrase. If it finds a match, it returns the key phrase and the part of the sentence that exists after this phrase.

For the purposes of this project, we created three distinct Matcher objects. The first one is the Precondition Matcher, that matches key phrases related to phrases that indicate that specific conditions exist for a Use Case to begin. Examples of such key phrases are: *“criteria must”*, *“before”*, *“conditions needed”*, *“precondition”* and *“if”*. The second is the Trigger Matcher, which matches phrases that imply an action exists that triggers the start of the Use Case. Example trigger phrases are: *use case begins*, *“triggered”*, *“scenario starts”*, and *“when”*.

The last rule is the Post-Condition Matcher, that matches phrases indicating the end of a Use Case, like for example: “*resulting*”, “*use cases terminates*”, “*process is completed*”, and “*scenario concludes*”.

In Figure 19, a rule created to identify a triggering action is presented. The “TEXT” attribute indicates that the token is a string. The regular expression that is used twice in the rule matches any of the words in the parentheses that could either start with a lowercase or an uppercase letter. It starts with the “i” ignorecase option that allows for case-insensitive matching. The “^” symbol indicates the start of the string and the key words are separated by an “OR” operator. The regular expression ends with the symbol “\$” that signals the end of the string with an optional “\n”. Between these two strings the “*” operator indicates that zero or more words can exist between these words. Following the last word that was part of the second regular expression, is a part of speech attribute matching and adverb followed by zero or more words. The last part of speech attribute matches a verb phrase, that can be followed by zero or more words.

```
ScenarioStartpattern = [{ 'TEXT': {'REGEX': '(?i)^(?:scenario|process|flow|action)$'}, {'OP': '*'},  
  {'TEXT': {'REGEX': '(?i)^(?:starts|begins|initiates|start|begin|initiate|began|initiated|started)$'},  
    {'POS': 'ADV', 'OP': '*'},  
    {'OP': '*'},  
    {'POS': 'VERB', 'OP': '+'}, {'OP': '*'} ]
```

Figure 19 Rule for matching a trigger condition in a document

The final pipeline used to build the Use Case Transformer is constructed with the models that yielded the best results in their respective tasks in combination with the rules defined with the Matcher.

5 Experiments and Results

This chapter is dedicated to describing the various experiments conducted while training the described machine learning models and their respective results. In the first section we present the training of a named entity recognition model with a Tok2Vec layer, a named entity recognition model with a Transformer layer and a span categorisation model. In the second section the training of relation extraction models with a Tok2Vec layer and a Transformer layer is presented.

5.1 Actor, System and Use Case extraction

The specifications of the machine used for the training were: an AMD Ryzen 5 2600 Six-Core Processor, 16GB RAM, NVIDIA GeForce RTX 2060 GPU and Windows 10 Home OS. The code was written in Python 3.9.1 using the command line or Jupyter Notebook, while the Transformer models were trained using the GPU provided by Google Colab Pro. The library used for training was spaCy and its various functions which will be listed in the next sessions.

5.1.1 NER with Tok2Vec component and Span Categorisation

In this section the training of NER and Span Categorisation models is presented. Before starting the training, the train-curve functionality by prodigy was used to determine the quality of the annotations, as well as if more training examples were necessary to improve the accuracy. The model is trained four times with different portions of the training examples, 25%, 50%, 75% and 100% of the data, and prints the accuracy scores with more data [56]. As shown in Table 5, the accuracy does not improve within the last training, indicating that the number of samples provided for training a NER component is sufficient.

% of training examples	Score
0%	0.00
25%	0.89
50%	0.90
75%	0.91
10%	0.91

Table 5 Train curve for NER

The NER model with a Tok2Vec component for predicting Actors and Systems was trained twice with two data split variations: 70% train data, 30% validation data and 80% train data, 20% validation data. The data were first shuffled on the document level, then split and transformed into spacy format. In parallel, the configuration file that includes all the necessary information for training the model was created.

The 70/30 split produced 2473 training samples and 1,055 validation samples, while the 80/20 split produced 2,824 training samples and 704 validation samples.

The initial learning rate was set at 0.001 and an Adam optimiser was used to adapt the learning rate of the weights after the first estimations. To handle overfitting and improve the model's generalisation, the dropout rate, namely the rate of zeroing out a random fraction of neurons at each training step, was set to 0.1, which was the default value in the configuration file. The number of epochs for each training was automatically decided based on the status of the evaluation score: The evaluation frequency, namely the rate of evaluating the model after certain steps, was set to 200. Patience was set to 1,600 steps, meaning that the training would stop if the evaluation score did not improve after 8 evaluations.

In Table 6, the overall and per label scores of the best model are shown for the two splits, after training the models for 21 epochs.

	70/30 split			80/20 split		
	Precision	Recall	F1	Precision	Recall	F1
Actor	0.908	0.922	0.915	0.929	0.930	0.930
System	0.864	0.825	0.844	0.873	0.818	0.845
Overall	0.901	0.906	0.903	0.918	0.922	0.913

Table 6 Evaluation of the NER models for Actor and System

As expected, in most cases the best model that was trained with 80% of the data has achieved better scores than the model trained with 70% of the data, although the results are not directly comparable because the test sets are different. Also, the Actor label scores better than the System label. This can be attributed to the fact that in many documents a system was labeled as an Actor because it was a system external to the Use Case.

A span categorisation model was trained to predict only the UseCase labels as spans instead of named entities. In addition to the parameters that were set at the same values as in the previous experiments, the n-gram suggester was set to suggest spans with size 1 to 46, an interval automatically inferred from the labeled data.

In Table 7 the overall and per label scores of the best model are shown for the two splits, after training the models for 23 epochs. The 70/30 split resulted in 2,453 train data and 1,049 validation data, while the 80/20 produced 2,802 train data and 700 validation data.

	70/30 split			80/20 split		
	Precision	Recall	F1	Precision	Recall	F1
Use Case (spancat)	0.736	0.634	0.681	0.780	0.679	0.726
Use Case (NER)	0.679	0.655	0.667	0.730	0.648	0.687

Table 7 Evaluation of NER and Span models only for the UseCase label

Compared to the models predicting Actors and Systems, models that predict UseCases performed worse and this is reasonable as UseCase entities consist of many tokens and have

more complex syntax and less clear boundaries. The span categoriser performed better than the named entity recogniser, although both models score low on recall, meaning that the models cannot recognise well which phrases should be classified as UseCases.

Table 8 presents the scores of the NER and span categorisation models trained with all three labels (ACTOR, SYSTEM, USECASE). The reason for training the models separately is that in many cases Actor and System tokens were in between a UseCase entity, so for the named entity recognition task, where overlapping was not allowed during labeling, these tokens were labeled as part of the UseCase, while for the span categorisation task, these tokens were labeled both as part of the UseCase and as Actor/System.

To train the NER model we used 80% of the data set, a total of 2,828 training samples while the rest 20%, namely 707 samples, were used for validation. For the span categorisation model, we used 2,802 training samples and 700 validation samples.

	NER			Span Categorisation		
	Precision	Recall	F1	Precision	Recall	F1
Actor	0.936	0.934	0.935	-	-	-
System	0.793	0.807	0.800	-	-	-
UseCase	0.762	0.686	0.722	-	-	-
Overall	0.869	0.838	0.853	0.882	0.825	0.853

Table 8 Evaluation of NER and Span Categorisation models for all labels

The span categorisation component provides evaluation scores only for the model's overall performance and not per entity type, but as the results of the two models are almost the same, it can be assumed that the models had similar performance also per entity.

	NER (one model: Actor/System/UseCase)			NER (two models: 1. Actor/System – 2. UseCase)		
	Precision	Recall	F1	Precision	Recall	F1
Actor	0.936	0.934	0.935	0.929	0.930	0.930
System	0.793	0.807	0.800	0.873	0.818	0.845
UseCase	0.762	0.686	0.722	0.730	0.648	0.687

Table 9 Comparison between NER model trained with all 3 labels, NER model trained with Actor/System and NER model trained with UseCase

At Table 9 we compare the NER model that was trained with all three labels simultaneously with the two NER models that were trained separately: the first one was trained with Actor and System labels and the second one was trained with UseCase labels. Although the two methods cannot be directly comparable because the models were trained and evaluated on different datasets, we can infer that the overlapping labels do not impact the model's performance greatly.

5.1.2 NER with Transformer component

In this section the experiments of using a Transformer model in the NER pipeline instead of a Tok2Vec component and their results are presented. The transformer that was used was RoBERTa-base from HuggingFace library. To work with transformer models, the use of a NVIDIA GPU with at least 10GB of memory was strongly recommended, as well as the installation of CUDA v9+ and PyTorch libraries was required [57].

On a Google Colab notebook we set up the English transformer pipeline, that includes the following components: transformer (RoBERTa-base), tagger, parser, ner, attribute ruler and lemmatiser.

For training and evaluating the NER model to predict Actors and Systems, we used 80% of the data set in training, a total of 2,818 documents and 20% of the data, which translates into 703 documents were used for validation. With the same split 2,821 training examples and 707 validation examples were used for the NER model trained on UseCases, and 2,825 training data and 706 validation data were used to train and evaluate the ner pipeline for Actors, Systems and Use Cases.

Table 10 shows the scores of two models: the first model was trained to predict Actors and Systems, while the second was trained to predict only UseCases. Table 11 presents the evaluation scores of the best model that was trained with all three categories. The results between the two training methods are similar, the model trained with all three labels scores slightly better, probably because of how the documents were shuffled and split.

	Precision	Recall	F1
Overall (Actor/System)	0.912	0.943	0.927
Actor	0.941	0.927	0.934
System	0.840	0.875	0.858
UseCase	0.732	0.749	0.740

Table 10 Performance of two separate models: the first for predicting Actors/Systems and the second for predicting UseCases

	Precision	Recall	F1
Overall	0.868	0.875	0.871
Actor	0.927	0.942	0.935
System	0.836	0.864	0.850
UseCase	0.785	0.777	0.781

Table 11 Performance of a model trained on all three labels

Compared to the NER models that had the traditional Tok2Vec component and the Span Categorisation model, the Transformer model yielded the best F1 scores. The most notable

improvement was recall, especially in predicting UseCases: the recall score of UseCases for the traditional NER model was 0.686 and the overall recall score was 0.838, while in the Transformer NER model the recall score of UseCases was 0.780 and the overall recall score 0.875 respectively.

	Precision	Recall	F1
Transition-based NER	0.869	0.838	0.853
Span Categorisation	0.882	0.825	0.853
Transformer NER	0.868	0.875	0.871

Table 12 Performance comparison between the different models

By taking into consideration that the UML Use Case Transformer is implemented with a human-in-the-loop approach, which favors recall over precision regarding performance, the Transformer-NER models outperformed the rest of the trained models and will be part of the UML Use Case Transformer pipeline for recognising Actors, Systems and Use Cases in a requirements text.

5.1.3 Token-Level evaluation of the Transformer – NER model

As spaCy’s NER objective is to “*predict the correct sequence of BILUO tags over a sequence of tokens*” [58], evaluation metrics don’t consider partial results. But, especially regarding the UseCase elements, the entity boundaries are not always clear, a UML expert can decide that a predicted UseCase is correct, even if the first or the last token was missed. A large number of correct token-level predictions could also mean that imposing stricter guidelines during the annotation process, can improve the model’s overall performance.

To score the model considering token-level prediction, the following process was implemented. Initially, the BILUO labels and the entity labels were extracted from the validation set along with their predicted counterparts. The validation set consists of 703 documents and 28,598 tokens. Based on the BILUO labels, precision, recall and F1 scores were calculated for all three labels.

	Bs	Is	Ls	Us	Os
Actual BILUO tag	1,909	8,239	1,909	1,505	15,036
Predicted BILUO tag	1,848	8,334	1,848	1,526	15,042
Correctly identified	1,741	7,460	1,728	1,406	13,836

Table 13 Count of BILUO labels in the validation set for all entities

$True\ Positives = Correctly\ Identified\ Bs + Correctly\ Identified\ Is + Correctly\ Identified\ Ls + Correctly\ Identified\ Us$

$True\ Negatives = Correctly\ Identified\ Os$

$False\ Positives = |(Predicted\ Bs + Predicted\ Is + Predicted\ Ls + Predicted\ Us) - (Correctly\ Identified\ Bs + Correctly\ Identified\ Is + Correctly\ Identified\ Ls + Correctly\ Identified\ Us)|$

$False\ Negatives = |Predicted\ Os - Correctly\ Identified\ Os|$

Calculation of Precision, Recall and F1:

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{1,741 + 7,460 + 1,728 + 1,406}{(12,335) + [(13,556) - (12,335)]} = \frac{12,335}{13,556} = 0.910$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{12,335}{12,335 + [15,042 - 13,836]} = \frac{12,335}{13,541} = 0.911$$

$$\text{F1} = 2 * \frac{(\text{Precision} * \text{Recall})}{(\text{Precision} + \text{Recall})} = 2 * \frac{0.910 * 0.911}{0.910 + 0.911} = 0.910$$

This score includes both the fully correctly predicted, as well as the partially predicted labels, with some margin of error in case an Actor or a System were misclassified. Alternatively, token-level predictions could receive a 0.5 weight when summed with the actual predictions, to normalise the results and reflect the model's performance better. Nonetheless, the results show that with adjustments in the labeled data the model can learn to predict all three labels better.

5.2 Relation Extraction

To predict the relationships between Actors, Systems and UseCases, spaCy's relation extraction component was trained using the NER and relations data sets. The data set was split in three parts: 70% of the samples, namely 2,420 documents were used for training, 15% of the samples, 518 documents were used for validation and another 15% was used for testing. The whole data set contained 6,376 Interact labels, 157 Include labels and 9 Extend labels. As the Include and Extend label size is too small, it was expected for the modeller to learn only the Interact relation.

The model also has a NER component, because it predicts relationships between two named entities. For the NER component the transformer-based approach was used, as it performed better.

Apart from the parameters mentioned in the previous experiments, for this training, the default parameter max length, which indicates the maximum distance two entities can have to be considered for relation prediction, was changed from 100 to 40.

	Precision	Recall	F1 Score
Overall	0.839	0.694	0.759

Table 14 Evaluation scores for the Relation Extractor

Table 14 shows the evaluation scores of the relation extraction model with the threshold set at 50%. The component only calculates the overall performance of the model, but in this case as the include and extend labels contribution in the learning process was very limited, the overall scores could be considered to be the scores for predicting an Interact relationship.

To further examine the confidence of the model of predicting a relationship, the test set was used to evaluate the predictions at certain thresholds. The default threshold was set at 50%, but when used in the UML Use Case transformer pipeline, the threshold is set at 10%, as the objective is to receive more predictions and manually apply corrections. Figure 20 is a graph that shows the evaluation scores based on the threshold value.

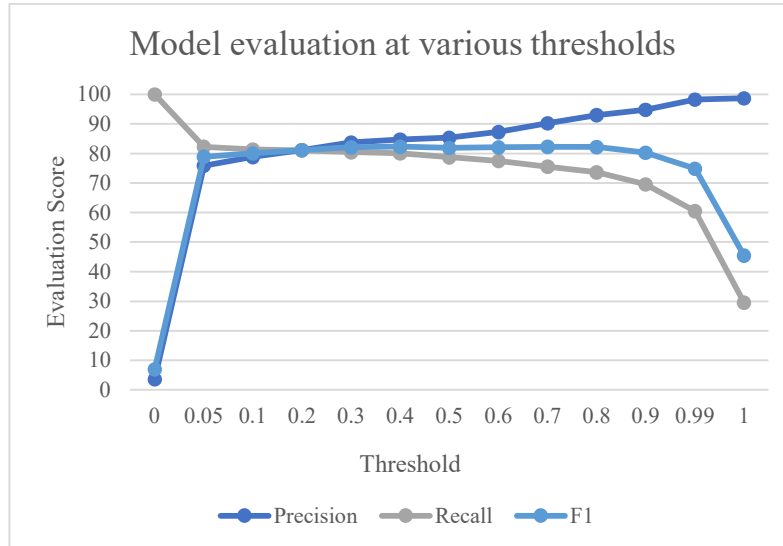


Figure 20 Graph showing the model's evaluation at various thresholds

5.3 Rule-based Matcher

To create rules that match of the three conditions, we reviewed the Use Case dataset to identify the key phrases in requirements documents that imply the existence of these conditions in the text. These key phrases were used as a core around which we created the patterns, using spaCy's Matcher library, as shown in the previous chapter.

Preconditions	Triggers	Postconditions
preconditions precondition	trigger triggers triggered triggering	postconditions postcondition post
must + have be	When Once +, when before	so + that
criteria conditions condition	scenario process flow action	scenario process flow action
+	+	+
must need have has needs	starts begins initiates start begin initiate began initiated started	ends finishes terminates concludes completes completed finished terminated concluded ended over
have has	use + case	use + case
+	+	+
to	starts begins initiates start begin initiate began initiated started	ends finishes terminates concludes completes completed finished terminated concluded ended over
if first firstly + ,		end + of
		resulting

Table 15 Words used to create rules for each of the conditions

Each of these patterns was added to its respective matcher instance. For example, the pattern that matches the phrase “so that” and identifies a post-condition is added to the matcher by typing:

```
PostConditionMatcher.add('so_that', [SoThatpattern])
```

To extract the matched condition from a document, the input text is split into sentences. Each sentence runs through the matcher and if one of the patterns is recognised, the span that consists of the pattern is returned.

As shown in Figure 21 the output of the example sentence “As an Investor, I need to see a summary of my investment accounts, so that I can decide where to focus my attention.” when using a pipeline that consists of the NER and relation extraction models and the Matcher would be:

```
As an Investor ACTOR , I ACTOR need to see a summary of my investment accounts USECASE , so that I ACTOR can decide where to focus my attention.

spans: [(2, 'Investor', 'ACTOR'), (4, 'I', 'ACTOR'), (7, 'see a summary of my investment accounts', 'USECASE'), (17, 'I', 'ACTOR')]
entities: ('Investor', 'see a summary of my investment accounts') --> predicted relation: {'INTERACT': 0.9997664, 'INCLUDE': 7.1274353e-06, 'EXTEND': 0.00060946407}
Post-conditions: [so that I can decide where to focus my attention.]
```

Figure 21 Example output of the UML UseCase pipeline

The first two outputs indicate the Actors, Systems and UseCases, the third output shows the predicted relationship, while the last output prints the matched condition.

5.4 Evaluation with out-of-sample data

In this section we will use out-of-sample data to test the performance of the pipeline. We present the requirement text that was used as an input the pipeline, we show the returned results and we comment on the model’s performance. Some requirements texts, use cases and user stories were selected from online sources and the rest were provided by Mark Kramer and Önder Babur from Information Technology Group, WUR.

Input	Output
<i>The user browses restaurant options. Once the preferred restaurant is selected, they place an order through the application. The user pays online or verifies they will pay in person. The order is sent from the app to the restaurant's internal system. The restaurant worker receives and processes the electronic order. [62]</i>	<p>'actors': ['User', 'Restaurant', 'Restaurant worker']</p> <p>'usecases': ['browses restaurant options', 'place an order', 'pays online', 'verifies they will pay in person', 'order is sent', 'receives and processes the electronic order']</p> <p>'system': ['Application'],</p> <p>'relationships': {'User' → 'browses restaurant options', 'User' → 'pays online', 'User' → 'verifies they will pay in person', 'Restaurant worker' → 'receives and processes the electronic order'}, 'RelationshipType': ['INTERACT']}</p> <p>'triggers': {'Trigger': ['the preferred restaurant is selected'], 'UseCase': ['place an order'],}}</p>

Evaluation: The pipeline identifies correctly the 'User' and the 'Restaurant worker' as *Actors*, but it also classifies as *Actor* the 'Restaurant'. The correct *Actor* in this case would be the 'Restaurant's internal system'. All Use Cases were identified. The model correctly identified the two *Use Cases* divided by the “or”. The last sentence contains two Use Cases: “receives the electronic order” and “processes the electronic order”, but the model could only classify them as one. The *System* is correctly identified. Out of six *interactions*, four were identified. The *Trigger* condition for *placing an order* was successfully matched.

UpCloud Airways software engineers design a branded and refreshed fare booking page, complete with tiered fare selection, add-on options like lounge access, free flight change or cancel abilities and complimentary checked bags. It also allows account holders to pay in credit, debit, online payment platforms or by UpCloud loyalty program miles. The software engineers conduct several use cases to establish how the booking flow works and identify potential concerns. They run cases that include: A customer browsing flight schedules and prices, A customer selecting a flight date and time, A customer adding on lounge access and free checked bags, A customer paying with a personal credit card, A customer paying with UpCloud loyalty miles, Through the various use cases, the engineering team identifies a malfunction with the optional add-ons prompting unless the user has a previously established account. The team rectifies the issue before launching the refreshed booking system. [63]

'actors': ['Upcloud airways software engineers', 'Account holders', 'Software engineers', 'Customer', 'User', 'Team'],

'usecases': ['design a branded and refreshed fare booking page', 'pay in credit, debit, online payment platforms or by UpCloud loyalty program miles', 'conduct several use cases', 'establish how the booking flow works', 'identify potential concerns', 'browsing flight schedules and prices', 'selecting a flight date and time', 'adding on lounge access and free checked bags', 'paying with a personal credit card', 'paying with UpCloud loyalty miles', 'identifies a malfunction with the optional add-ons prompting unless', 'rectifies the issue'],

'system': ['System'],

'relationships': {'Upcloud airways software engineers' → 'design a branded and refreshed fare booking page', 'Account holders' → 'pay in credit, debit, online payment platforms or by UpCloud loyalty program miles', 'Software engineers' → 'establish how the booking flow works', 'Software engineers' → 'identify potential concerns', 'Software engineers' → 'conduct several use cases', 'establish how the booking flow works' → 'conduct several use cases', 'identify potential concerns' → 'conduct several use cases', 'Customer' → 'selecting a flight date and time', 'Customer' → 'paying with a personal credit card', 'Team' → 'rectifies the issue', 'RelationshipType': ['INTERACT']}

'triggers': {'Trigger': ['launching the refreshed booking system', 'UseCase': ['rectifies the issue']}]

Evaluation: In this text, the author has used five different ways to refer to the same *Actor*: “UpCloud Airways software engineers, software engineers, They, engineering team, team”. While the pipeline, impressively, extracted a four-word *Actor*, it classified each reference as a different *Actor*. The *System* was not identified, so the pipeline returned the default word “System”. The sentence “The software engineers conduct several use cases to establish how the booking flow works and identify potential concerns.” contains three relationships: An “interact” relationship between the *Actor* “software engineers” and the *Use Case* “conduct several use cases” and two “inclusion” relationships from the including *Use Case* “conduct several use cases”, to the included *Use Cases* “identify potential concerns” and “establish how the booking flow works”. The pipeline identifies five relationships: it relates each *Use Case* with the *Actor* and identifies the relationship between the two *use cases* but as *interaction* instead of *inclusion*. In this part of the text: “...unless the user has a previously established account. The team rectifies the issue before launching the refreshed booking system.”, the condition described with the word “unless”, is a *precondition* of the *use case* “rectifies the issue” and the phrase “before launching the refreshed booking system” is its *trigger*. The pipeline only extracts the *trigger* because the *precondition* is mentioned in the previous sentence.

... Envisage an environmental scientist in Cambodia, researching the impact of deforestation in Vietnam as part of investigating the regional impacts of climate change. She submits her search keywords, in Cambodian, and receives responses indicating there is some data from the 1950s, printed in a 1960 pamphlet, in the Bibliothèque Nationale, a library in Paris, France, in French. She receives an abstract of some form that enables her to decide that the data are worth accessing, and initiates a request for a digital copy to be sent. She receives the pamphlet as a scanned image of each page, and she decides that the quantitative information in the paper is useful, so she arranges transcription of the tabular numerical data and their summary values into a digital form and publishes the dataset, with a persistent identifier, and links it to a detailed coverage extent, the original paper source, the scanned pages and her paper when it is published. She also incorporates scanned charts and graphs from the original pamphlet into her paper. Her organization creates a catalog record for her research paper dataset and publishes it in the WIS global catalog, which makes it also visible to the GEO System of Systems broker portal. [64]

{**actors**': ['Environmental scientist', 'Organization', 'User'],

'usecases': ['submits her search keywords, in Cambodian,', 'receives responses indicating', 'receives an abstract of some form', 'decide that the data are worth accessing', 'initiates a request for a digital copy to be sent', 'receives the pamphlet as a scanned image of each page', 'decides that the quantitative information in the paper is useful', 'arranges transcription of the tabular numerical data and their summary values into a digital form', 'publishes the dataset', 'links it to a detailed coverage extent, the original paper source', 'incorporates scanned charts and graphs from the original pamphlet into her paper', 'creates a catalog record for her research paper dataset', 'publishes it in the WIS global catalog'],

'system': ['Geo system'],

'relationships': {'User' → 'submits her search keywords, in Cambodian,', 'User' → 'receives an abstract of some form', 'User' → 'decide that the data are worth accessing', 'User' → 'initiates a request for a digital copy to be sent', 'User' → 'decides that the quantitative information in the paper is useful', 'Organization' → 'creates a catalog record for her research paper dataset'}, 'RelationshipType': ['INTERACT']}

'postconditions': {'Postcondition': ['she arranges transcription of tabular numerical data and their summary values into a digital form and publishes dataset with a persistent identifier and links it to a detailed coverage extent original paper source scanned pages and her paper when it is published'],

'UseCase': ['receives the pamphlet as a scanned image of each page', 'decides that the quantitative information in the paper is useful', 'arranges transcription of the tabular numerical data and their summary values into a digital form', 'publishes the dataset', 'links it to a detailed coverage extent, the original paper source'],

Evaluation: 'Geo system' is an Actor in this use case and the System is 'WIS'. There is a missed use case "makes it also visible". There are several missed interactions, even though their syntax was similar to those correctly identified. The word "She" has been replaced in the interaction with the default word "User". The sentence "She receives the pamphlet as a scanned image of each page, and she decides that the quantitative information in the paper is useful, so she arranges transcription of the tabular numerical data and their summary values into a digital form and publishes the dataset, with a persistent identifier, and links it to a detailed coverage extent, the original paper source, the scanned pages and her paper when it is published." contains a sequence of use cases and there are no conditions, but because the word "so" is used to match a post-condition, the pipeline returns this phrase as a post-condition to all the identified Use Cases.

Triggers: The user indicates that she wants to purchase items that she has selected.

{**actors**': ['User', 'Billing system'],

Preconditions: User has selected the items to be purchased.

Post-conditions: The order will be placed in the system. The user will have a tracking ID for the order. The user will know the estimated delivery date for the order.

Normal Flow: The user will indicate that she wants to order the items that have already been selected. The system will present the billing and shipping information that the user previously stored. The user will confirm that the existing billing and shipping information should be used for this order. The system will present the amount that the order will cost, including applicable taxes and shipping charges. The user will confirm that the order information is accurate. The system will provide the user with a tracking ID for the order. The system will submit the order to the fulfillment system for evaluation. The fulfillment system will provide the system with an estimated delivery date. The system will present the estimated delivery date to the user. The user will indicate that the order should be placed. The system will request that the billing system should charge the user for the order. The billing system will confirm that the charge has been placed for the order. The system will submit the order to the fulfillment system for processing. The fulfillment system will confirm that the order is being processed. The system will indicate to the user that the user has been charged for the order. The system will indicate to the user that the order has been placed. The user will exit the system.

[65]

'usecases': ['indicates that she wants to purchase items that she has selected', 'have a tracking ID for the order', 'know the estimated delivery date for the order', 'indicate that she wants to order the items that have already been selected', 'present the billing and shipping information that the user previously stored', 'confirm that the existing billing and shipping information should be used for this order', 'present the amount that the order will cost, including applicable taxes and shipping charges', 'confirm that the order information is accurate', 'provide the user with a tracking ID for the order', 'submit the order to the fulfillment system for evaluation', 'provide the system with an estimated delivery date', 'present the estimated delivery date', 'indicate that the order should be placed', 'request that the billing system should charge the user for the order', 'confirm that the charge has been placed for the order', 'submit the order to the fulfillment system for processing', 'confirm that the order is being processed', 'indicate to the user that the user has been charged for the order', 'indicate to the user that the order has been placed', 'exit', 'unknown usecase'],

'system': ['System'],

'relationships': {'User' → 'indicates that she wants to purchase items that she has selected', 'User' → 'know the estimated delivery date for the order', 'User' → 'confirm that the existing billing and shipping information should be used for this order', 'User' → 'confirm that the order information is accurate', 'User' → 'present the estimated delivery date', 'User' → 'indicate that the order should be placed', 'User' → 'exit'], 'RelationshipType': ['INTERACT']},

'postconditions': {'Postcondition': ['conditions order will be placed in system', 'UseCase': ['unknown usecase']]},

'preconditions': {'Precondition': ['User selected the items to be purchased'], 'UseCase': ['unknown usecase']},

'triggers': {'Trigger': ['The user indicates that she wants to purchase items that she has selected'], 'UseCase': ['indicates that she wants to purchase items that she has selected']}}}

Evaluation: The pipeline correctly identified “Billing system” as *Actor*, but failed to extract “Fulfillment system”. Some conditions were wrongly identified as *Use Cases*, but they were also correctly identified as conditions. In this example, the conditions are mentioned in the beginning of the text, they are not mapped to any of the listed use cases. So, the pipeline has created an “unknown usecase” and linked the conditions to this use case. Many of the interactions with the system were missed due to the fact that the subject in many cases was the system instead of the user. The fact that the trigger phrase ‘indicates that she wants to purchase items that she has selected’ was wrongly identified as use case, led to be also extracted as *interaction* and as *Use Case* in its own trigger condition.

As a user, I want to look at the event schedule, so that the system will show an organised calendar with upcoming events.	<p>'actors': ['User'],</p> <p>'usecases': ['look at the event schedule'],</p> <p>'system': ['System'],</p> <p>'relationships': {'[User]' → ['look at the event schedule'], 'RelationshipType': ['INTERACT']},</p> <p>'postconditions': {'Postcondition': ['system will show an organised calendar with upcoming events'], 'UseCase': ['look at the event schedule']}</p>
Evaluation: The pipeline successfully recognised the elements, the relationship and the post-condition.	
As a member, I want to view the message board when I log in to the application, so that I can see the messages posted by other members.	<p>'actors': ['Member'],</p> <p>'usecases': ['view the message board'],</p> <p>'system': ['System'],</p> <p>'relationships': {'[Member]' → ['view the message board'], 'RelationshipType': ['INTERACT']},</p> <p>'postconditions': {'Postcondition': ['see messages posted by other members'], 'UseCase': ['view the message board']},</p> <p>'triggers': {'Trigger': ['log in to the application'], 'UseCase': ['view the message board']}</p>
Evaluation: The pipeline successfully recognised the elements, the relationship, the trigger and the post-condition.	
<p>I walk through the game and I meet an NPC who can give me a quest.</p> <p>If I complete the quest I will be rewarded with money and/or items and with XP.</p> <p>A quest can be finding an item, or defeating a certain monster.</p>	<p>'actors': ['Npc'],</p> <p>'usecases': ['walk through the game', 'meet', 'give me a quest', 'be rewarded with money and/or items and with XP'],</p> <p>'system': ['System'],</p> <p>'relationships': {'[Npc]' → ['walk through the game'], 'Npc' → 'meet', 'Npc' → 'give me a quest'], 'RelationshipType': ['INTERACT']}</p> <p>'preconditions': {'UseCase': ['be rewarded with money and/or items and with XP'], 'Precondition': ['I complete the quest']}</p>
Evaluation: As this user story is written in the first person, the main actor is not recognised. Instead, only the “NPC” is identified but the acronym incorrectly changes to 'Npc'. Also, the pipeline considers that “P” refers to the “NPC” in the text and relates the “NPC” with the Use Case 'walk through the game'.	
<p>As a customer, I want shopping cart feature so that I can easily purchase items online.</p> <p>As a user, I want to back up my entire hard drive.</p>	<p>'actors': ['Customer', 'User'],</p> <p>'usecases': ['want shopping cart feature', 'back up my entire hard drive'],</p> <p>'system': ['System'],</p>

	<p>'relationships': {'Customer'] → ['want shopping cart feature', ['User'] → ['backup my entire hard drive'], 'RelationshipType': ['INTERACT']},</p> <p>'postconditions': {'Postcondition': ['easily purchase items online'], 'UseCase': ['want shopping cart feature']}</p>
<p>Evaluation: When the action in the user story starts with “<i>I want to...</i>”, the pipeline correctly dismisses this phrase and only extracts the action, like in the second example. In the first example, the only verb in the action is the verb “<i>want</i>”, so the pipeline extracts the whole phrase. So, for these two user stories all the elements, relationships and conditions were correctly identified.</p>	

Table 16 Input, Output and Evaluation of the pipeline

6 Integration into Prose to Prototype

In this chapter we discuss the construction of a pipeline that consists of the best performing trained models, the rule-based matcher and various post-processing methods that builds into a UML Use Case transformer. The pipeline was integrated into the Prose to Prototype project, as a part of a system that aspires to handle the requirement lifecycle, from elicitation to acceptance.

6.1 System Design

The P2P system is currently run in Docker to work independently from the user's machine specifications. As it is mainly built to run in GNU/Linux systems, it requires using a Linux subsystem, like *Lima* or *WSL2*, to run in Windows. The P2P project follows the structure of the Python-based *Django* framework with multiple applications and a model-template-view architecture [59] .

P2P is divided into two main subsystems: the ngUML backend and the ngUML editor. The ngUML backend consists of the *Model-application*, that contains the ORM-related object to store UML metadata and provides endpoints to the ngUML editor to post and edit UML diagrams. The database used to store the data is *PostgreSQL* and *Redis* is used to handle inserts and improve performance. It also contains the *Extraction-application* that adds the NLP-tasks with metadata to the queue and converts the output from the NLP pipeline to ORM-objects. Besides the UML Use Case modeller, the ngUML backend also hosts a UML Class modeller that was built by Tiantian Tang [60] and a UML Activity modeller, built by Pepijn Griffioen [61] . In the future, other UML models, like the Component model, will be added in the P2P system. Finally, the Runtime-application contains the logic to generate applications from the objects stored in Model.

The ngUML editor, functions as the *Presentation tier* and is tasked with gathering the requirements as well as presenting visualisations of the respective UML models. These two systems communicate with each other through *REST API*.

6.2 Use Case model specification

The Use Case model consists of the use case model generation pipeline, a use case metamodel that facilitates storage and retrieval of the use case models and API endpoints that allow the creation, update, delete and retrieval of the use case models.

6.2.1 UML Use Case metadata generation pipeline

The pipeline consists of the trained transformer-based named entity recogniser and relation extractor in conjunction with post-processing functions and the rule-based matcher.

We first load the named entity recogniser to extract the *Actors*, the *Use Cases* and the *Systems* from the given text. Before adding the extracted objects to a list, we first perform some postprocessing tasks, to ensure that each object will be unique in the database and will have the appropriate use case format. For example, as seen in Figure 22, we use *Wordnet*'s lemmatiser for each *Actor*, to retrieve its lemma, because we want to store the singular form

of the word. We then change the first letter of each actor to uppercase, to adhere with the UML Use Case standards and then we check if the specific *Actor* already exists in the list. Finally, we remove personal pronouns from the list.

```
# Extract Actors
for e in doc.ents:
    if e.label_ == "ACTOR":
        actors = e.text
        singles = lemmatizer.lemmatize(actors) # Convert plural to singular
        singles = singles.capitalize() # Change first letter of each actor to uppercase
        if singles not in actor_list: # Add only Actors that don't already exist in the list
            actor_list.append(singles)
keyWords = 'We,Our,I,She,They,He,It,Their,You,Your,my,you,our,he,she,he/she,she/he,She/He,She/he,He/She,He/she'
for word in list(actor_list): # Remove words from actor list
    if word in keyWords:
        actor_list.remove(word)
```

Figure 22 UML generation of Actors using NLP

We follow the almost same process for the *System*, with the difference that we only return one *System*, as we assume that the requirement is about the functionality of a single system and any other systems found in the text are references to the same system. As in some texts the system is not explicitly mentioned in the text, in case we were not able to extract this information, we return a default “*System*”, because this information is needed for the creation of the system boundary.

As for the *Use Cases*, we change the first letter of the first word to lowercase and we check if a *Use Case* with the same name already exists in the list before adding it.

The next component in the pipeline is the relation extractor. We use a dictionary to store the results, as we need to store the relationship type and the two endpoints.

```
for value, rel_dict in doc._.rel.items():
    for sent in doc.sents:
        for e in sent.ents:
            for b in sent.ents:
                if e.start == value[0] and b.start == value[1]:
                    if (
                        rel_dict['INTERACT'] >= 0.1
                    ): # Change the names of the Actors and UseCases to match NER extractions
                        actor = e.text
                        keyWords = 'We,Our,I,She,They,He,It,Their,You,Your,my,you,our,he,she, he/she, she/he, She/He, She/he, He/She, He/she'
                        if actor in keyWords:
                            actor = "User"
                        lemmatizer = WordNetLemmatizer()
                        single_actor = lemmatizer.lemmatize(actor)
                        single_actor = single_actor.capitalize()
                        if single_actor not in actor_list:
                            actor_list.append(single_actor)
                        single_usecase = b.text
                        single_usecase = single_usecase[0].lower() + single_usecase[1:]
                        interaction = list(rel_dict.keys())[0]
                        relationship_dict["From_Classifier"].append(single_actor)
                        relationship_dict["To_Classifier"].append(single_usecase)
                        relationship_dict["RelationshipType"].append(interaction)
```

Figure 23 UML generation of Relations using NLP

As presented in Figure 23 we break the text into sentences, assuming that the relation between two elements exists in the same sentence and we process the identified “*Actors*” and “*Use Cases*” to match the extracted objects from the previous task. To handle personal pronouns, we replace such actor objects with the default word “*User*”. The threshold for classifying relationships is set at 0.1, resulting in decreased precision but increased recall. The reason we prefer recall over precision is because of our human-in-the-loop approach, we want to extract all the possible relationships and then manually delete or edit the false positives. In the

pipeline we have also included the “*EXTEND*” and “*INCLUDE*” options, as in the future we might be able to find enough data to train the relation extractor on these labels.

To extract *post-conditions*, *preconditions* and *triggers* from the text we use the rule-based matcher. Figure 24 shows the code for preconditions and the same concept is applied to the rest of the conditions. For each of the conditions, we split the text into sentences and we first search for a phrase that matches one of the respective patterns. Then, we remove the keyword or key phrase from the matched phrase. For example, if we have extracted the phrase “*so that the user can access the server.*” as a post-condition, we remove the key phrase “*so that*” and the punctuation.

We then need to pair each of the extracted conditions with a use case. We check whether the use case list contains a use case that appears in the same sentence as the condition. If yes, we add the use case and the condition in the dictionary. If not, we check if there is a use case in the use case list with the name “*unknown_usecase*” and add it to the dictionary. Else, we create a new use case with the name “*unknown_usecase*” and add it to the dictionary.

```
re_key_words = re.compile(r"\b(" + "|".join(keyword) + ")\W", re.I) # Remove precondition keywords
toPrint = ''.join(str(e) for e in preconditions)
Preconditions = re_key_words.sub("", toPrint)
Precondition = re.sub(r'^\w\s', '', Preconditions)

# Match precondition with a usecase
for x in usecase_list:
    if x in sent.text:
        precondition_dict["UseCase"].append(x)
        precondition_dict["Precondition"].append(Precondition)

if not any(x in sent.text for x in usecase_list):
    unknown_usecase = "unknown_usecase"
    if unknown_usecase in usecase_list:
        precondition_dict["UseCase"].append(unknown_usecase)
        precondition_dict["Precondition"].append(Precondition)
    else:
        usecase_list.append(unknown_usecase)
        precondition_dict["UseCase"].append(unknown_usecase)
        precondition_dict["Precondition"].append(Trigger)
```

Figure 24 UML generation of conditions using NLP

6.2.2 Use Case metamodel

The extracted use case elements, relationships and conditions are used to map and create objects to the database, based on their respective class, as shown in Figure 25.

```
rel_n = len(extraction.result['relationships']['RelationshipType'])
for i in range(0, rel_n):
    rel_from = objects[extraction.result['relationships']['From_Classifier'][i]]
    rel_to = objects[extraction.result['relationships']['To_Classifier'][i]]
    rel_type = extraction.result['relationships']['RelationshipType'][i]
    if rel_type == 'INTERACT':
        Interaction.objects.create(
            from_classifier=rel_from,
            to_classifier=rel_to,
            usecase=model,
        )
```

Figure 25 Mapping and Creation of "Interaction" objects in the database

For the generation of the Use Case model, we created only selected classes as shown in Figure 26, as we focused on the elements that we retrieve from the pipeline and other information like “*Scenario*” that can be easily added by the user in the ngUML editor. *UseCaseClassifier* and its subtypes are used to create *UseCase*, *Actor* and *UseCaseSystem* objects. *Relationship* and its subtypes are used to create *Interaction*, *Inclusion* and *Extension* objects. *Postcondition*, *Precondition*, *Trigger* and *Scenario* are used to create their respective object.

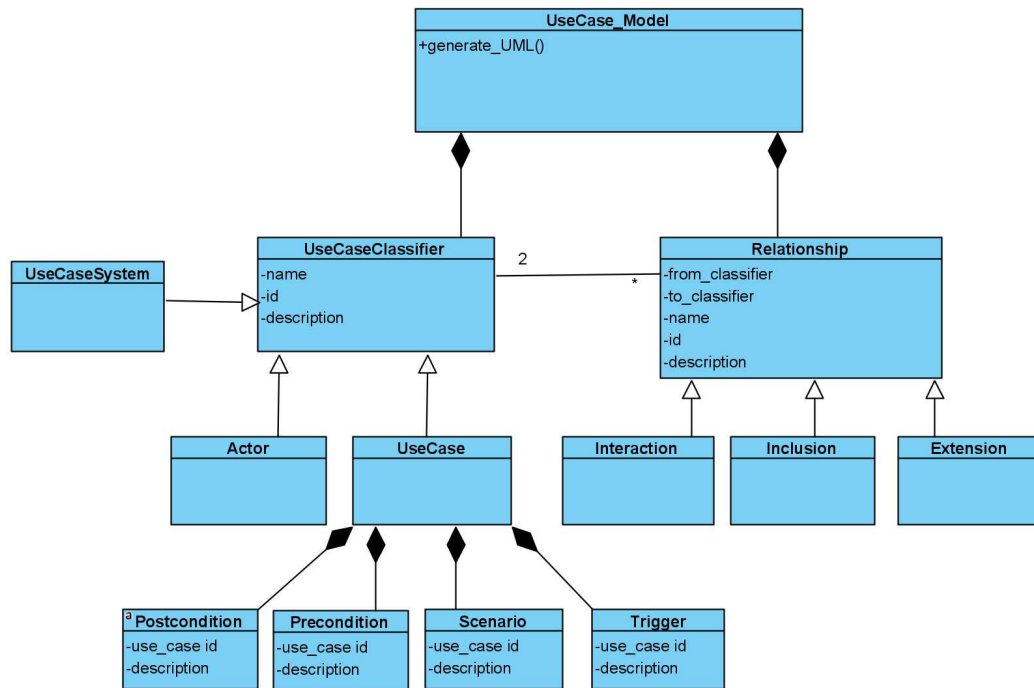


Figure 26 Class diagram of the Use Case model

6.2.3 Use Case model methods

To create, update, get and delete Use Case models we use internal APIs that work with specific methods. These methods help us find elements by their name or id, populate classifiers and relationships and make model changes in the frontend. The basic operations that apply to all the objects are creation and deletion. Besides these actions, *Actor*, *UseCase* and *UseCaseSystem* objects, can be repositioned in the Use Case diagram. The name of *Postconditions*, *Preconditions*, *Triggers* and *Scenarios* can be retyped. For example, a phrase might have been stored as a *Precondition*, but it can be retyped as *Trigger*. Finally, the endpoints of a relationship can change. For example, there might be an “*interact*” relationship between a certain actor and a use case and we can choose to relate the use case with another actor. There is a condition that does not allow “*interact*” relationships between two use cases and “*extend*” or “*include*” relationships between an actor and a use case.

To communicate all the metadata in the ngUML editor and create a UML Use Case diagram, we use external APIs and we define a specific project and system to place the model.

7 Discussion

We started working on this research project with the ambition to build an application that facilitates communication between business and IT stakeholders. We focused on the user requirements and the development of a UML Use Case model that receives requirements texts, use cases and user stories as input and produces UML Use Case metadata, that can be used to create Use Case diagrams.

By studying the related work, we concluded that the existing UML Use Case models (e.g., UMGAR, RAPID) have been built on rule-based NLP techniques, like syntactic parsing. So, we decided to experiment with supervised learning in combination with rule-based matches. To train the models we needed an annotated data set, but as it was not available. We compiled a new requirements data set we gathered requirements texts and user stories from various data sets and independent sources. We then experimented with various annotation tools, as most of them did not support relation extraction and in the end, we decided to use “*prodigy*”. To annotate the data set, we carefully studied our samples and managed to create guidelines that cover the ambiguities of natural language that occurred.

We used the annotated data set to train transition-based and transformer-based named entity recognisers and a transformer-based relation extractor, to extract “*Use Case*”, “*Actor*” and “*System*” elements. For the extraction of “*Preconditions*”, “*Triggers*” and “*Post-conditions*”, we used a rule-based matcher.

Finally, we created a pipeline, comprised of the trained models, the matcher and various post-processing NLP tasks and integrated it into the *Prose to Prototype* project. We tested the performance of the pipeline using out-of-sample data and we reached the following conclusions:

Actors: The pipeline manages to identify the majority of actor elements in a text. In some cases, it fails to identify internal systems or misclassifies external systems.

Use Cases: The pipeline manages to identify the majority of use case elements in a text. In some cases, it misclassifies conditions as use cases.

Systems: The pipeline manages to identify the majority of system elements in a text. If a system was not identified, the pipeline returns the word “System”. It sometimes misclassifies internal systems as systems.

Interact relationship: The pipeline identifies all relationships in User Stories. It performs moderately in larger unstructured texts, for reasons that are not always clear.

Conditions: The pipeline performs very well in identifying conditions in a text. In some cases, it falsely considers a word as a key word and classifies a phrase as a certain condition.

In comparison with the UMGAR tool [15], the RAPID tool [17] and the UML Generator [18], which identify Actors, Use Cases and the relationships between them, our pipeline can additionally identify Systems, Preconditions, Post-conditions and Triggers.

In their research, Lucassen et al. [21] and Elallaoui et al. [22] focus on extracting conceptual models from User Stories. Their models perform very well, when the input User Story follows

the structure: “*As ... , I want to ... , so that ...*”. In comparison, our model performs equally well and can additionally handle User Stories with different structures.

7.1 Limitations

The first limitation we encountered in this project was the lack of requirements documents, especially documents that describe the requirements of the system in detail. The reason is that companies are not willing to publish online the specifications of their software systems due to security risks and competition. As we explained in Chapter 3, the most valuable documents for training were from the PURE dataset, because they provide wholesome requirements and use cases in a semi-structured or unstructured way and include most of the use case elements. The request for more examples of such documents would basically further improve the models’ performance, as the existing data set with the addition of User Stories was already sufficient for the training task.

Another consequence of the limited access to requirements documents, is the lack of data that indicate an *include* or an *extend* relationship between two use cases. Our data set consists of only 160 *include* and 9 *extend* relationship samples, insufficient to properly train the relation extraction model.

The second limitation was the nature of the project itself, as this was a master’s research project with limited resources. In terms of human resources, if a second annotator was available, it would increase data integrity. To compensate for the lack of the second annotator, we dedicated a large amount of time, to carefully study the data set, create annotation guidelines and discuss with experts how to handle different cases. The evaluation scores show that the annotation process was successful, as it yielded high quality labeled samples. Time constraints led to experimentation with specific models instead of training, evaluation and comparison of various models. Instead, we focused on studying these specific models and run multiple experiments to ensure that the final pipeline can handle diverse requirements documents and successfully extract all the necessary information.

7.2 Future work

The creation of a UML Use Case modeller using supervised learning proved to be a successful research experiment. As this was the first time that this method was used to build such a modeller and due to the limitations mentioned in the previous section, it is natural that there is a lot of space for improvements.

For researchers, who are interested expanding the model’s capabilities, the first suggestion would be to annotate *preconditions*, *triggers* and *post-conditions* and replace rule-based matching with supervised learning. The existing data set provides sufficient examples for these labels and it could be further expanded for better representation.

Relation extraction is a challenging task, as it is heavily dependent on precise tagging the relations before training [66]. It also depends on the correct tagging of the named entities that are related and on the distance between the related named entities. For this research project we trained spaCy’s experimental relation extractor with satisfying results. In the future, more robust models can be trained to improve the modeller’s performance.

The relation extractor was trained to only identify “*interact*” relationships from the text, due to the lack of samples that show “*include*” and “*extend*” relationships. In the future, it would be a useful addition to the project to find such sufficient examples and train the relation extractor. In case this is not possible, the future researchers could create rules to match phrases from the text that indicate these relationships between use cases and add them to the pipeline.

A natural language processing task that could improve relation extraction and the modeller’s overall performance is coreference resolution. Coreference resolution facilitates information extraction by indicating “*expressions that refer to the same entity in a text*” [67]. As many structures of requirements texts, follow specific syntactic rules, like for example User Stories and Use Cases, it is suggested to train a coreference resolution modeller with data labeled specifically for UML Use Case information extraction. The optimal position of the coreference resolution task in the pipeline would be as a post-processing step for the relationship extractions. The model should parse the relationship dictionary, the actor list and the text and match the extracted personal pronouns with the “*Actors*” extracted by the named entity recogniser.

For this research project we focused on extracting the most important elements that compose a UML Use Case model. Other characteristics like *Actor* and *Use Case Generalisation*, *Scenarios*, *Alternative Flows* and *Exceptions* that capture important information were not included, due to time constraints. In the future, these elements can be added, to build a fully-featured modeller.

The system under development is one of the entities that we extract from a requirements text and we use it to indicate the system boundary of the respective use case. We assume that each requirements document refers to only one system, so we include all the use cases extracted from the text into one system boundary. It is possible though, that two or more systems and their interactions with the users are referenced in a text. As future work, identifying different systems and build separate use case models for each system, could be a useful functionality.

8 Conclusion

Our research presents a novel approach to extracting UML Use Case metadata from user requirements texts, with satisfying results. Instead of using traditional NLP techniques like POS tagging and syntactic parsing, we used supervised learning to train transformer-based named entity recognition and relation extraction models. These models were used to extract “Actors”, “Use Cases”, “Systems” and their relationships from various forms of user requirements texts. Additionally, we created heuristic rules, using spaCy’s Matcher, to extract “Preconditions”, “Triggers” and “Post-conditions”. We synthesised a pipeline with these models in combination with NLP post-processing tasks, to improve the output.

To train the models we compiled a data set, consisting of 3,542 samples documents from various sources: we used 80 documents from the PURE data set [36] and 75 documents from UML themed books. We enriched the data set with 1,618 User Stories [37] and 1,638 transformed User Stories. We also included 131 textual requirements from the Tera-PROMISE database [38]. We annotated the data set for named entity recognition, span categorisation and relation extraction, using *prodigy*. We conducted the annotation based on specific guidelines, reviewed by a UML expert and an NLP expert. The annotation process yielded 10,779 “Actor” labels, 2013 “System” labels, 6,460 “Use Case” labels, 6,671 “Interact” labels, 160 “Include” labels and 9 “Extend” labels.

For the extraction task, we experimented with spaCy’s transition-based named entity recogniser, experimental transformer-based span categorizer, transformer-based named entity recogniser and experimental transformer-based relation extractor. The best performing model was the transformer-based named entity recogniser with 0.871 F1, 0.875 Recall and 0.868 Precision scores, on average, for all labels. The relation extractor was not able to learn the “Include” and “Extend” labels, because of the limited number of samples. The model was able to predict the “Interact” label, with the confidence threshold set at 50%, with 0.839 precision, 0.694 recall and 0.759 F1.

We built a pipeline using the best performing models and rule-based matcher. We integrated the pipeline into the P2P project, a system that transforms user requirements into UML models and runnable prototypes. We tested the pipeline using out-of-sample data and we received very good results. We have commented on the limitations and proposed solutions for future improvements, as well as ideas for future enhancements.

In conclusion, the UML Use Case pipeline in combination with a human-in-a-loop approach, is a solution that can develop quality Use Case models, with decreased effort, compared to manual development or other proposed solutions.

References

- [1] G. Elliott, *Global Business Information Technology: An Integrated systems approach*, Addison-Wesley, 2004.
- [2] N. B. Ruparelia, "Software development lifecycle models," *ACM SIGSOFT Software Engineering Notes*, Vol. 35, No. 3, pp. 8-13, 2010.
- [3] C. Osman and P. Zalian, "From natural language text to visual models: A survey of issues and approaches," *Informatica Economica*, vol.20, no.4, pp. 44-61, 2016.
- [4] G. Ramackers, P. Griffioen, M. Schouten and M. Chaudron, "From Prose to Prototype: Synthesising Executable UML Models from Natural Language," 2021.
- [5] K. Peffers, T. Tuunanen, C. Gengler, M. Rossi, W. Hui, V. Virtanen and J. Bragge, "The Design Science Research Process: A Model for Producing and Presenting Information Systems Research," in *1st International Conference, DERIST 2006 Proceedings*, 2006.
- [6] R. Back, P. L. and I. Paltor, *Analysing UML Use Cases as Contracts*. In: France R., Rumpe B. (eds) «UML»'99-The Unified Modeling Language, Berlin, Heidelberg: Springer, 1999.
- [7] OMG, *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1* Object Management Group (Technical report, Object Management Group), 2011.
- [8] Y. Wautelet, S. Heng, K. M and I. Mirbel, "Unifying and extending user story models," in *Proceedings of the international conference on advanced information systems engineering (CAiSE), LNCS, vol 8484*, Berlin, 2014.
- [9] M. Cohn, *User Stories Applied: For Agile Software Development*, Addison-Wesley, 2004.
- [10] C. Sims, "New User Story Format Emphasizes Business Value," 2008. [Online]. Available: <https://www.infoq.com/news/2008/06/new-user-story-format/>.
- [11] t2Informatik, "User Story Definition," [Online]. Available: <https://t2informatik.de/en/smartpedia/user-story/>.
- [12] D. Klein and C. Manning, "Stanford Parser 1.6," 2007. [Online]. Available: <https://nlp.stanford.edu/software/lex-parser.shtml>.
- [13] G. Miller, "WordNet 2.1," 2006. [Online]. Available: <https://wordnet.princeton.edu/>.
- [14] L. Qiu, "JavaRAP," 2007. [Online]. Available: <https://www.comp.nus.edu.sg/~qiul/NLPTools/JavaRAP.html>.

- [15] K. D. Deeptimahanti and M. A. Babar Lero, "An Automated Tool for Generating UML Models from Natural Language Requirements," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand, 2009.
- [16] OpenNLP, "OpenNLP," [Online]. Available: <https://opennlp.apache.org/>.
- [17] P. More and R. Phalnikar, "Generating UML Diagrams from Natural Language Specifications," *International Journal of Applied Information Systems*, 2012.
- [18] C. Narawita and K. Vidanage, "UML generator – use case and class diagram generation from text requirements," *International Journal on Advances in ICT for Emerging Regions (ICTer)*, 10(1), pp. 1-10, 2017.
- [19] T. Yue, L. Briand and Y. Labiche, "A systematic review of transformation approaches between user requirements and analysis models," *Requirements Engineering*, 16, pp. 75-99, 2010.
- [20] T. Tang, "From Natural Language to UML Class Models: An Automated Solution Using NLP to Assist Requirements Analysis," Leiden University, Leiden, 2020.
- [21] G. Lucassen, M. Robeer, F. Dalpiaz and J. M. B. S. van der Werf, "Extracting conceptual models from user stories with Visual Narrator," *Requirements Engineering* 22, no. 3, pp. 339-358, 2017.
- [22] M. Elallaoui, K. Nafil and R. Touahni, "Automatic Transformation of User Stories into UML Use Case Diagrams using NLP Techniques," in *8th International Conference on Ambient Systems, Networks and Technologies, Procedia Computer Science 130*, 2018.
- [23] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami and C. Dyer, *Neural Architectures for Named Entity Recognition*, arXiv, 2016.
- [24] M. Honnibal, I. Montani, S. Van Landeghem and A. Boyd, "spaCy: Industrial-strength Natural Language Processing in Python," doi: 10.5281/zenodo.1212303, 2020.
- [25] M. Honnibal, "Embed, encode, attend, predict: The new deep learning formula for state-of-the-art NLP models," November 2016. [Online]. Available: <https://explosion.ai/blog/deep-learning-formula-nlp>.
- [26] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu and P. Kuksa, *Natural Language Processing (almost) from Scratch*, arXiv, 2011.
- [27] M. Honnibal, "SPACY'S ENTITY RECOGNITION MODEL: incremental parsing with Bloom embeddings & residual CNNs," November 2017. [Online]. Available: <https://www.youtube.com/watch?v=sqDHBH9IjRU&t=2781s>.
- [28] M. Honnibal, "Model Architectures," [Online]. Available: <https://spacy.io/api/architectures#tok2vec-arch>.

- [29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez and L. Kaiser, "Attention is All You Need," *CoRR*, 2017.
- [30] J. Alammam, "The Illustrated Transformer," 2018. [Online]. Available: <http://jalammar.github.io/illustrated-transformer/>.
- [31] Y. Liu, M. Ott, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer and V. Stoyanov, "RoBERTa: A Robustly Optimized {BERT} Pretraining Approach," *CoRR*, 2019.
- [32] J. a. C. M.-W. a. L. K. a. T. K. Devlin, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, arXiv, 2018.
- [33] Hugging Face Team, "RoBERTa base model," [Online]. Available: <https://huggingface.co/roberta-base>.
- [34] E. Schmuhl, L. Miranda, A. Kadar, S. Van Landeghem and A. Boyd, "Spancat: a new approach for span labeling," June 2022. [Online]. Available: <https://explosion.ai/blog/spancat>.
- [35] S. Van Landeghem, "SPACY v3: Custome trainable relation extraction component," 2021. [Online]. Available: <https://spacy.io/usage/layers-architectures>.
- [36] A. Ferrari, G. O. Spagnolo and S. Gnesi, "PURE: A dataset of public requirements documents.," in *2017 IEEE 25th International Requirements Engineering Conference (RE)*, Lisbon, Portugal, 2017.
- [37] F. Dalpiaz, *Requirements data sets (user stories)*, doi: 10.17632/7zbk8zsd8y.1, 2018.
- [38] OpenScience, "tera-PROMISE," [Online]. Available: <http://openscience.us/repo/>.
- [39] A. Cockburn, *Writing Effective Use Cases*, Pearson Education Limited, 2000.
- [40] G. Jalloul, *UML by Example*, Cambridge University Press, 2004.
- [41] K. Bittner and I. Spence, *Use Case Modeling*, Addison Wesley Professional, 2002.
- [42] P. J. Papagiorgji and P. M. Pardalos, *Software Engineering Techniques Applied to Agricultural Systems - An Object-Oriented and UML Approach*, Springer, 2005.
- [43] T. Lethbridge and R. Laganieri, *Object-Oriented Software Engineering - Practical Software Development using UML and Java*, McGraw-Hill Publishing Company, 2004.
- [44] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering using UML, Patterns and Java*, Pearson, 2009.
- [45] F. Armour and G. Miller, *Advanced Use Case Modeling, Volume One - Software Systems*, Addison-Wesley Professional, 2001.

- [46] INSPIRE, "Infrastructure for spatial information in Europe," [Online]. Available: <https://inspire.ec.europa.eu/cases-studies/pilot-projects/inspire-energy-pilot/440/bu-data-discoverable-inspire-geoportal/60732>.
- [47] Z. A. Hamza and M. Hammad, "Generating UML Use Case Models from Software Requirements Using Natural Language Processing," in *2019 8th International Conference on Modeling Simulation and Applied Optimization (ICMSAO)*, 2019.
- [48] H. Nakayama, T. Kubo, J. Kamura, Y. Taniguchi and X. Lian, "Doccano: Text Annotation Tool for Human," 2018. [Online]. Available: <https://github.com/doccano/doccano>.
- [49] M. Tkachenko, M. Mikhail, A. Holmanyuk and N. Liubimov, "Label Studio," 2020-2022. [Online]. Available: <https://github.com/heartexlabs/label-studio>.
- [50] O. Collective, "Universal Data Tool," [Online]. Available: <https://universaldatatool.com/app/>.
- [51] P. Stenetorp, S. Pyysalo, G. Topic, T. Ohta, S. Ananiadou and J. Tsujii, "brat: a Web-based Tool for NLP-Assisted Text Annotation," 2012. [Online]. Available: <https://brat.nlplab.org>.
- [52] tagtog, "tagtog," [Online]. Available: <https://www.tagtog.net/>.
- [53] I. Montani and M. Honnibal, "Prodigy: A new annotation tool for radically efficient machine teaching," *Artificial Intelligence*, 2018.
- [54] M. Honnibal and I. Montani, February 2021. [Online]. Available: https://www.youtube.com/watch?v=9k_EfV7Cns0.
- [55] SpaCy, "Matcher," [Online]. Available: <https://spacy.io/api/matcher>.
- [56] I. Montani, "Built-in Recipes," [Online]. Available: <https://prodi.gy/docs/recipes>.
- [57] I. Montani and M. Honnibal, "Embeddings, Transformers and Transfer Learning," [Online]. Available: <https://spacy.io/usage/embeddings-transformers>.
- [58] I. Montani, "Evaluate NER wrong results," 2019. [Online]. Available: <https://github.com/explosion/spaCy/issues/3909>.
- [59] Django Software Foundation, "django Documentation," [Online]. Available: <https://docs.djangoproject.com/en/3.2/ref/applications/>.
- [60] T. Tang, *From Natural Language to UML Class Models: An Automated Solution Using NLP to Assist Requirements Analysis*, Leiden University, 2020.
- [61] P. Griffioen, *Generating process models from textual requirements using transformer based natural language processing*, Leiden University, 2022.

- [62] N. Daly, "What Is a Use Case?," April 2022. [Online]. Available: <https://www.wrike.com/blog/what-is-a-use-case/>.
- [63] Indeed Editorial Team, "Use Cases: What They Are and a List of Examples," 2021. [Online]. Available: <https://www.indeed.com/career-advice/career-development/list-of-use-cases-examples>.
- [64] C. Little, "Spatial Data on the Web Use Cases & Requirements," 2016. [Online]. Available: <https://www.w3.org/TR/sdw-ucr/#MeteorologicalDataRescue>.
- [65] S. Sehlhorst, "Sample Use Case Example," 2007. [Online]. Available: <https://tynerblain.com/blog/2007/04/09/sample-use-case-example/>.
- [66] S. B. T. & V. R. de Abreu, "A review on Relation Extraction with an eye on Portuguese," *Braz Comput Soc* 19, p. 553–571, 2013.
- [67] K. Clark, D. Jurafsky, C. Manning and C. Potts, "The Stanford Natural Language Processing Group," [Online]. Available: <https://nlp.stanford.edu/projects/coref.shtml>.

Appendix

Appendix A

Named Entities Recognition and Span Categorisation Rules:

- a. ***“The clerk selects and views a claim from the list.”***
In this sentence there are two different use cases: (a) selects a claim from the list, (b) views a claim from the list. The phrase “from the list” is only mentioned once but it refers to both verbs. The rule for similar situations is to use only one label “USECASE” for both use cases.
- b. ***“The system displays a list of possible duplicate claims from within loss database.”***
Although there are no Actors mentioned in this example, the “clerk” is implied based on context. In such cases, the verb phrase will still be labeled as “USECASE”.
- c. ***“The application provides the user a list of claims.”***
The entity “provides a list of claims” is discontinued because the word “user” that represents an Actor split the Use Case. As it not possible to split a label in two parts, the “USECASE” label will include the Actor. While in prodigy’s NER manual, one word cannot have two labels, Span Categorisation allows overlapping, so “user” will also be individually labeled as “ACTOR”.
- d. ***“If the customer has had fewer than 3 attempts at entering the PIN, the system informs the customer that he or she should have another attempt.”***
In cases where a sentence starts with “If”, the verb phrase of the second part of the sentence will be labeled as “USECASE”. The verb phrase in the first part of the sentence can be indicating a pre-condition, an alternative flow or an exception.
- e. ***“The system asks the Bank System to approve the withdrawal. The Bank System responds with a withdrawal acceptance to approve the withdrawal.”***
In this example the “Bank System” is an external system, so although it is a system, it will be labeled as “ACTOR”.
- f. ***“... the new Service Provider will send a request to the NPAC SMS to change the Subscription Version status to pending.”***
Verb phrase “will send a request” is the indicator of an interaction between the Actor and the System so it will not be labeled as “USECASE”. The main Use Case in this example is the phrase “change the Subscription...”.
- g. ***“The loan agreement is presented to the customer for acceptance and signature.”***
The two Use Cases in this example: “accepts the loan agreement” and “signs the loan agreement” are expressed as part a verb phrase as nouns, because of the use of passive voice. In such samples, the whole verb phrase will be annotated: “is presented...signature”.
- h. ***“The general user has no ability to modify system settings.”***
In this sentence the Use Case expresses the inability of the actor to interact with the system. For consistency purposes, negations will also be labeled as “USECASE”.
- i. ***“The New and Old Service Providers use internal and inter-company processes to resolve the conflict.”***
In the scenario above, if the Actors were modeled in a UML Use Case diagram, three different Actor elements would be distinguished: The New Service Provider,

the Old Service Provider and their generalisation, Service Provider. In such cases, only the ancestor will be labeled as “*ACTOR*”.

- j. ***“A collection curator wants to have items be made available under the permissions they were configured once the embargo date has been reached.”***
The verb phrase “*items be made available...*” is labeled as “*USECASE*” indicating an action done by the System as interaction with the Actor.
- k. ***“An IT manager wants to know about IT resource requirements early in the project lifecycle, ...”***
The verb phrase “*know about IT resource requirements...*” will be labeled as “*USECASE*”, indicating a request the user has from the system.
- l. ***“As a user, I want to store behavior videos ... written by Christopher James.”***
In cases where the name of a specific person is used, the name will not be labeled as an “*ACTOR*”.
- m. ***“Based on the loan officer’s experience...”***
Although in this example the interaction between an Actor and the System is irrelevant, the phrase “loan officer” will be labeled as an Actor for consistency purposes.
- n. ***“An archivist restricts access to certain files by user, so that he or she can allow donor representatives to see certain files.”***
The word “*user*” in this sentence is an indicator of how files are sorted and not an Actor, so it will not be labeled.
- o. ***“A developer has an easy way to define questions the user can ask and perform.”***
In the scenario above, if the Use Cases were modeled in a UML Use Case diagram, three different Use Cases elements would be distinguished: (a)define questions (developer), (b) ask questions (user), (c)perform questions (user). In such cases the whole phrase “*define questions ... perform*” will be labeled as “*USECASE*”.

Relation Extraction Rules:

- a. ***“A repository manager wants to compose collections, limiting the collection to the items sharing the same provenance, limiting the collection to represent a part of a collection have a singular provenance, or assembly a collection from other collections and objects.”***
In this sentence “*compose collections*” is the main Use Case, while “*limiting the collection...*” and “*assembly a collection...*” are extending Use Cases and their relationship with the main Use Case will be labeled as “*EXTEND*”.
- b. ***“The Receiving Agent validates the box id with the TC registered ids and maybe signs the paper form for the delivery person.”***
The Use Case “*signs the paper form*” is extending the “*validates the box id*” Use Case because of the word “*maybe*” and their relationship will be labeled as “*EXTEND*”.
- c. ***“As a repository manager, I want to elect to either replicate remotely or not and possibly to replicate beyond the primary remote site.”***
Similar to the previous example, the use of the word “*possibly*” indicates that the verb phrase “*replicate beyond the primary remote site*” is an extending Use Case to the base Use Case “*elect to either replicate remotely or not*” and their relationship will be labeled as “*EXTEND*”.
- d. ***“The Bank System responds with a withdrawal acceptance to approve the withdrawal.”***

The main Use Case in the above sentence is “*approve the withdrawal*” and the included Use Case is “*responds with a withdrawal acceptance*”. The relationship between verb phrases that are connected with the word “*to*” will be labeled with the label “*INCLUDE*”. The second verb phrase is the base Use Case that is related to the Actor and the relationship between them will be labeled as “*INTERACT*”.

- e. “*The loan officer determines the appropriate terms of the loans, using suggested loan terms ...*”

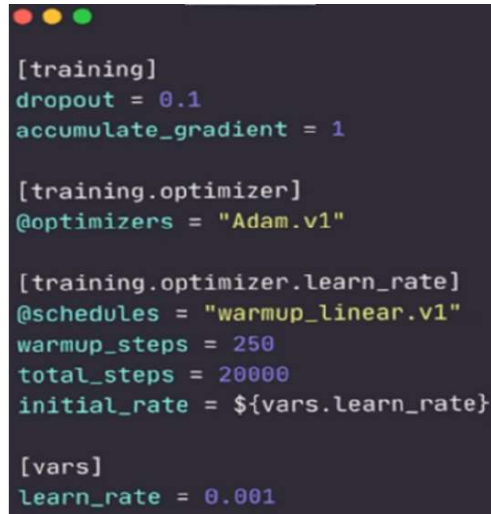
The base Use Case in the example is “*determines the appropriate terms of the loans*”, while the verb phrase “*using suggested loan terms*” is the included Use case and their relationship will be labeled as “*INCLUDE*”.

- f. “*As a recruiter, I want to be able to extend an ad for another 30 days by visiting the site and updating the posting, so that ...*”

Base Use Case “*extend an ad for another 30 days*” that includes the Use Cases “*visiting the site*” and “*updating the posting*” and their relationship will be labeled as “*INCLUDE*”. The indicator of the Include relationship in similar examples is the word “*by*”.

Appendix B

In Figure 27, the registry looks in its “optimisers” table for a function named “Adam v1”. The function will be called and the other elements of the block will pass in as arguments, in this case, the learning rate.



```
[training]
dropout = 0.1
accumulate_gradient = 1

[training.optimizer]
@optimizers = "Adam.v1"

[training.optimizer.learn_rate]
@schedules = "warmup_linear.v1"
warmup_steps = 250
total_steps = 20000
initial_rate = ${vars.learn_rate}

[vars]
learn_rate = 0.001
```

Figure 27 Example configuration file [54]

The configuration is resolved bottom-up, so the result of that function is computed, and the resulting object passed into Adam. With this approach each object only receives the configuration it needs itself [54].

When saving a model to disk, the config file is saved too and is used to reconstruct the model for later re-use.