Großer Beleg

Towards an Automatic Generation of UML Class Diagrams from Redocumented Textual Requirements

submitted by

Andreas Huber

born 04.05.1990 in Pegnitz

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisor: Dr.-Ing. Birgit Demuth
Professor: Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted August 26, 2020

Contents

1 The Need for Automatically Transforming (Redocumented) Requirements to UML

Building software systems can be compared to building houses. Before beginning construction, plans have to be designed to communicate, *what exactly* is to be built to ensure that every stakeholder knows what to do instead of just beginning to stack bricks. This directly translates to software development. During every phase in the software development process, models can be employed. They communicate, what the goal of the software is, how the implementation of this goal is to be designed and realized, and last but not least - they document functional and technical aspects of the developed system [KSHE18].

During the initial elicitation of software requirements, stakeholders typically refrain from using formal methods like modeling. Requirements are most likely conveyed via natural language. Problematically, natural language is not universal and can be ambiguous [DJ17]. This ambiguity can be reduced by using models to communicate aspects of the system in design. Yet generating these models can be cumbersome. With modern natural language processing (NLP) frameworks computers can aid in this task of translating natural language to models.

Particularly during analysis phases, the unified modeling language's (UML) class diagrams assist in communicating aspects of the problem domain. They enable customers and developers to discuss, whether the system, which is to be crafted, contains all relevant aspects and entities [KSHE18]. Having such a model available in the blink of an eye with an automatic analysis of the requirements simplifies the communication between all stakeholders. On the one hand, this empowers less tech or modeling educated people to generate such models. On the other hand, developers have a starting point for model-based development techniques like model-driven engineering or design models.

Furthermore, there are tools working with requirements, structuring, and analyzing them. One of them is *SoftRedoc*, a redocumentation framework written by Harry Sneed[1]. While its main purpose is to analyze source code to gain insights about data and control flow, it can extract business rules and concepts from e.g. requirement documents [SV20] [SE96]. With the idea being that redocumented requirements provide an even better entry point for the generation of models than plain natural language, this work seeks for ways of automatically transforming these redocumented requirements into class models. Therefore, a prototypical implementation of a tool named *MoreRedoc* (Modeling of Redocumented Requirement Documents) is implemented.

To solve the problem of translating natural language or redocumented requirements into UML class models, one has to have an overview of the landscape of existing approaches and how well these approaches fulfill their task. This overview provides the possibility to gain insights about possible gaps in the existing methods and their results. Furthermore, a proposed solution that tries to bridge these gaps has to be evaluated. This leads to the following research questions, which are to be answered in this thesis.

[1] http://www.harrysneed.de/

> ***Research Question 1**: What approaches and tools for automatically translating requirements given in natural language to class diagrams with the help of NLP exist and how are they evaluated?*
>
> ***Research Question 2**: What gaps exist in the analyzed approaches?*
>
> ***Research Question 3**: How well does the proposed tool translate redocumented requirements into UML class diagrams? What metrics can be used to quantitatively evaluate the findings and how can they be applied?*

To answer these questions, basics and the theoretical foundations have to be discussed first. A running example with a small set of requirements is provided in Chapter 2. It illustrates the findings in several chapters, ranging from explaining the basics of applying MoreRedoc to it to generate a class model, which then is evaluated. In Chapter 4, related work for translating natural language into UML models is discussed and gaps in these approaches are identified. Trying to bridge these gaps, requirements for a new approach are stated and the algorithm and the implementation of their solution are discussed in Chapter 5. Furthermore, in said chapter, an algorithm for a quantitative evaluation of generated models with the metrics precision and recall is provided and its application on the running example and an example project from SoftRedoc is presented. This thesis is then concluded in Chapter 6 with a summary of the findings and a brief outlook.

2 Running Example - A Shopping Customer

To describe the theoretical background and the approach of MoreRedoc in a more tangibly way, a small artificial running example is presented. Furthermore, this example will be used for the automatic generation of class models with MoreRedoc and an evaluation. Assuming that a document containing a small description of a system is given, the following requirements are presented:

- ***Requirement 1***: *The customer enters his customer id and his password. If they are correct, he is authenticated. Before shopping, the customer has to enter his address.*
- ***Requirement 2***: *The customer browses the available articles. He puts the articles in his cart.*
- ***Requirement 3***: *Every article has an unique article id.*
- ***Requirement 4***: *When the shopping is done, the customer can order the cart. The cart's articles are shipped to the customer if all articles are available.*

Suitable candidates for domain concepts, extractable by redocumentation or documentation tools (or even NLP tools via simple noun analysis), are *customer, id, password, address, article, article id* and *cart*. A possible UML class model for this domain can be seen in Figure 5.3 beside the generated model in Figure 5.4.

3 Basics of Redocumentation, Natural Language Processing, and UML

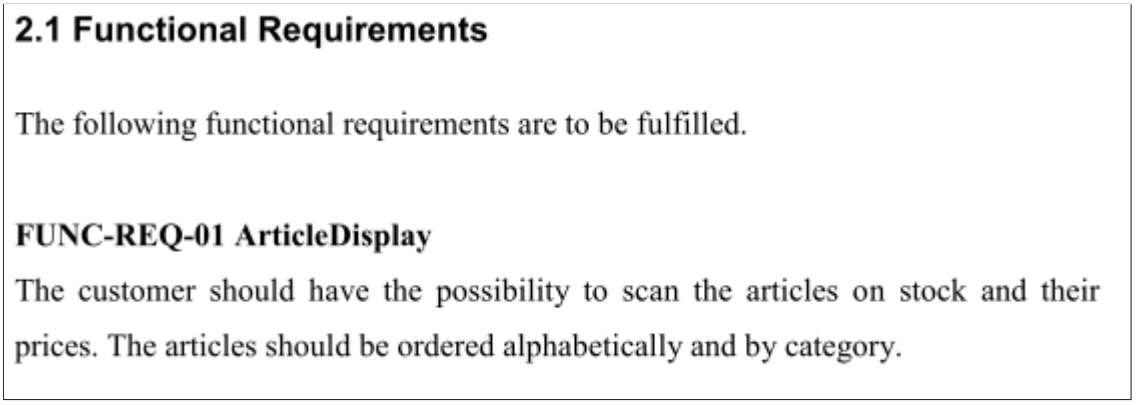
This chapter gives an introductory overview of the used methodologies and technologies. At first, a definition of redocumentation is given and why it can be used as a data basis for automatically generating models is explained. In the workflow of MoreRedoc, the requirement documents and its redocumentation are then analyzed using NLP methods. To get a grasp of what is happening during this analysis, certain aspects and methods of NLP are presented. Finally, the proposed solution generates a UML class diagram out of this information. Hence, UML is presented shortly and it is described which of its language elements are used during the translation of requirements to class diagrams.

3.1 Redocumentation as a Means of Surveying Software

[Mas06] defines redocumentation as generating comprehensible models from existing software. Software systems, especially when being in use for a long time, can bloat and get much more complex than anticipated due to steady development and bug fixing. When the documentation of this system is not maintained as steadily as the code, technical debt is accrued and it gets gradually harder to maintain the system. At this point, a redocumentation helps to get a better understanding of legacy systems and to support ongoing development. Ways of doing this are for example documenting the source code, extracting business processes from it, or generating new documents describing the system.

[AdO05] sees redocumentation more broadly as a redesign of artifacts that are generated during the software development process. For example, this means analyzing and compressing a requirement document into a more human or machine-readable form and extract information systematically.

There are tools for automatically redocumenting software. The one used for the approach of MoreRedoc is SoftRedoc, written by Harry Sneed. It analyzes source code and identifies data and control flows through the software to simplify the understanding of what it actually does, especially, when there is no documentation. A majority of the source code does not deal with depicting business-related processes, but with the technical environment of the software. Thus, one of the key aspects is separating these parts of the code which implement business processes and those which do not [SV20]. Furthermore, SoftRedoc can be used to extract business rules and concepts from other documents [SE96], e.g. requirements which are specified in natural language.



2.1 Functional Requirements

The following functional requirements are to be fulfilled.

FUNC-REQ-01 ArticleDisplay

The customer should have the possibility to scan the articles on stock and their prices. The articles should be ordered alphabetically and by category.

Figure 3.1: Example of a requirement document annotated for usage in SoftRedoc

Within the context of MoreRedoc, SoftRedoc is used to extract domain concepts from a given requirement document. SoftRedoc takes specifications in natural language in text format as input. The tool then scans the document for relevant domain concepts. While an initial list contains all substantives, this list can be altered by the user, thus domain knowledge can be applied. The result is a list containing relevant aspects of the application domain. Finally, SoftRedoc separates requirements and outputs two lists in a comma-separated value format. One list depicts the relevant domain entities per requirement, the other lists the sentences in natural language per requirement. An example can be seen in the following image and tables. Figure 3.1 contains part of an input document for SoftRedoc, Figures 3.2 and 3.3 represent part of the contents of the CSVs containing the output of SoftRedoc.

DOCU	OrderEntry	OWNS	REQU	FUNC-REQ-01 ArticleDisplay
REQU	FUNC-REQ-0	USES	DATA	customer
REQU	FUNC-REQ-0	USES	DATA	possibility
REQU	FUNC-REQ-0	USES	DATA	articles
REQU	FUNC-REQ-0	USES	DATA	stock
REQU	FUNC-REQ-0	USES	DATA	prices

Figure 3.2: Output of SoftRedoc describing the domain concepts of the example in Figure 3.1

FUNC-REQ-01	Function	your_name	defined	ArticleDisplay	1	ArticleDisplay
FUNC-REQ-01	Function	your_name	defined	ArticleDisplay	2	FUNC-REQ-01 ArticleDisplay
FUNC-REQ-01	Function	your_name	defined	ArticleDisplay	3	The <customer> should have the <possibility> to scan the <articles> on <stock> and their <prices>.
FUNC-REQ-01	Function	your_name	defined	ArticleDisplay	4	The <articles> should be ordered alphabetically and by <category>.

Figure 3.3: Output of SoftRedoc showing the text of the requirements, separated and indexed for each requirement, example from Figure 3.1

This redocumentation provides the data which MoreRedoc works with. On the one hand, it defines which domain entities are to be modeled. On the other hand, the clear separation of requirements allows a more precise analysis compared to analyzing the text in natural language without structured information. It's very clear which sentence belongs to which requirement. This limits the possibilities for ambiguity.

3.2 How Natural Language Processing Can Be Used to Analyze Requirements

So far, there are extracted domain concepts and the text of each requirement as a data foundation for the model. Equally important as these data entries are the relations between them and the actions they can perform. E.g. SoftRedoc has extracted the domain concepts of a *Customer* and an *Article*, it is of crucial importance for a model in which context they appear, e.g. a customer can *order* articles. SoftRedoc can not extract this kind of relation. To process further, a toolkit for analyzing the sentences that describe this kind of relationships and actions is needed.

Exactly this can be done with NLP methods. **NLP** is a field of computer science, artificial intelligence, and linguistics, which aims at making human languages like German or English understandable and generatable by computers. MoreRedoc utilizes the aspect of language comprehension. Human language is therefore translated to a formal representation (e.g. parse trees), which can be analyzed automatically [Kum11].

One of the first steps of natural language processing and analysis is **part-of-speech (POS) tagging**. A document containing text, which is to be analyzed, is split into individual sentences. The goal of POS tagging is to assign a label to every word of every sentence, representing a certain class of words. Basic classes can e.g. be the three major parts of speech - nouns, verbs, and adjectives/adverbs. One of the most used sets of classes is the Penn Treebank corpus [ID10]. It contains 48 tags, further dividing the aforementioned classes. E.g. nouns can either be in the classes of regular or proper nouns, in singular or plural. Verbs, adjectives, and adverbs are separated similarly. Furthermore, there are further auxiliary classes going beyond the three major parts of speech, e.g. punctuation, determiners, existential there, modal words, or interjections [San90]. Implementations employ rule-based approaches, Markov model approaches, or maximum entropy approaches to generate the models for POS tagging [ID10]. An example of a tagged text from the running example (*"The customer enters his customer id and his password."*) computed with MoreRedoc using the Penn Treebank POS tags can be seen in Table 3.1.

Word	**Tag**
The	DT (Determiner)
customer	NN (Singular or mass)
enters	VBZ (Verb, 3rd person singular present)
his	PRP (Personal pronoun)
customer	NN
id	NN
and	CC (Coordinating conjunction)
his	PRP
password	NN
.	. (Sentence-final punctuation)

Table 3.1: POS tagged sentence from the running example.

Once the document is split up and the sentences with every word are tagged, further processing can be done. When reviewing again the sentence from the running example *"The customer enters his customer id and his password."* (see 3.4), it can be assumed, that *id* and *password* refer to the *customer*. While this is true for speakers of the English language, this is not

the case for a machine. A document can contain a word and certain references to it. In the example, *his* refers to the *customer*. The task of automatically resolving this dependency is called **coreference resolution**. Generally, linguistic expressions referring to other words are called mentions. A common example are personal pronouns like *his* or *her*. Coreference resolution maps the mentions to a referent. Usually, coreference resolving systems employ supervised neural machine learning to generate models for this task. [JM14].

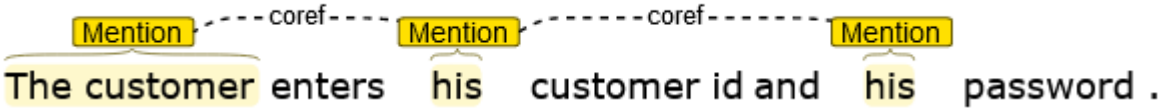


Figure 3.4: Coreference resolution for an example sentence from the running example, generated with Stanford CoreNLP via https://corenlp.run/

Another task to enable the semantic analysis of natural language is **constituency parsing**. Its purpose is to compute the structure of sentences. These structures may be represented in the shape of a tree, as can be seen in Figure 3.5. Approaches for this task include the Cocke-Kasami-Younger (CKY) algorithm, a dynamic programming approach for parsing context-free grammars or partial parsing approaches, which do not necessarily aim at producing a finest granular parse tree but rather a granular enough tree that splits sentences in more general structures like *noun phrase - verb phrase - noun phrase* instead of splitting up each phrase again. Approaches for partial parsing employ e.g. finite-state transducer algorithms or chunking, a task to structure a sentence in non-overlapping segments [JM14].

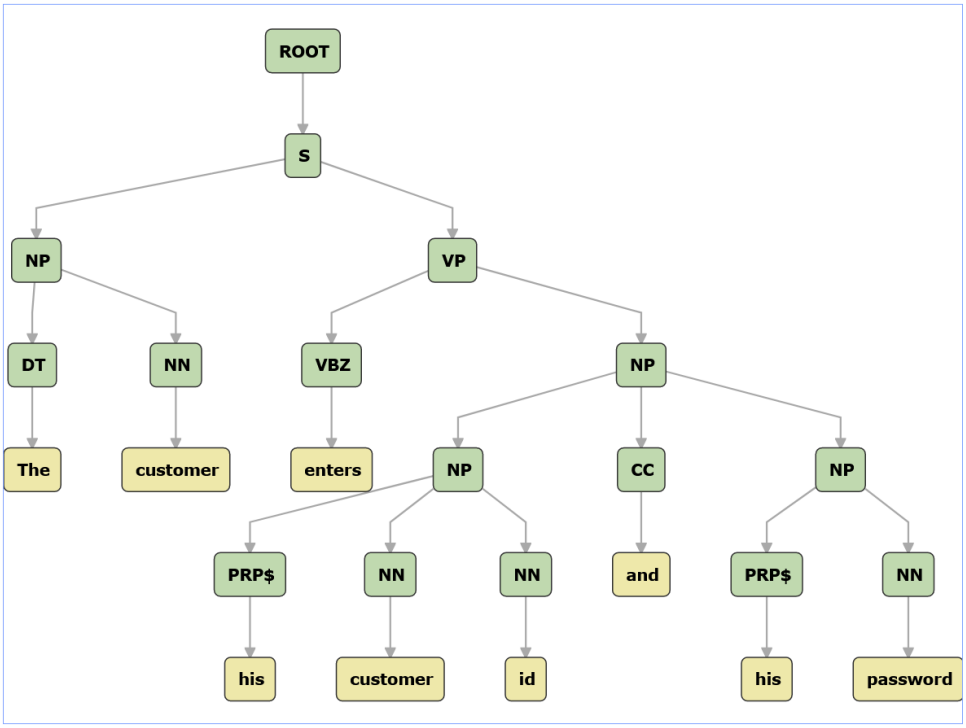


Figure 3.5: Constituency parsing for an example sentence from the running example, generated with Stanford CoreNLP via https://corenlp.run/

After the three aforementioned tasks, a given document is restructured to be analyzed more easily and in a more comprehensive way than just the raw sentences of the document. Focus then lays on the semantic analysis. **Information Extraction** is the task of turning unstructured information being textually represented into structured data. Two subtasks employed by MoreRedoc are the recognition of named entities (e.g. people, places, or organizations) and the extraction of relations between entities mentioned in the text. Named entities are entities that can be referred to by a proper name, for example, persons, locations, organizations, or quantities, that can be interpreted numerically. Implementations mostly rely on machine learning algorithms (neural models) or grammar-based approaches. Relation extraction on the other hand is the detection of relations between entities (which do not have to be named entities). Relations can be expressed as tuples between entities. A so-called RDF (Resource Description Framework) triple describes the relationship between a subject, a predicate, and an object in a sentence. Looking at the sentence "*The customer browses the available articles*" from the running example, triples extracted by MoreRedoc using an NLP framework can be seen in 3.2. Approaches rely on either pattern analyses of e.g. parse trees or (semi)supervised machine learning techniques [JM14].

Subject	**Predicate**	**Object**
customer	browses	available articles
customer	browses	articles

Table 3.2: RDF triples extracted from a sentence from the running example

There are many toolkits and frameworks providing support for natural language analyzing tasks. A mature and well-maintained one is **Stanford CoreNLP**. It provides support for core natural language analysis and information extraction. The basic concept is to annotate entities in a given text and extract information from these annotations. Raw text streams through various annotators, which add information to the extracted entities. This results in an annotated text containing a mapping from the occurring entities to their analysis, which themselves can be analyzed further. Annotators include POS tagging, sentence splitting, named entity recognition, coreference resolution, and more. Stanford CoreNLP is released under the GPLv3 license and therefore an open-source framework with a strong copyleft licensing model [MSB+14].

3.3 UML and its Relevant Elements for the Proposed Tool

Finally, after redocumenting the requirement specifications and analyzing the natural language, the resulting data can be modeled. A common language for realizing this is UML.

UML is a modeling language designed for system architects, software engineers, and developers as a tool for providing support in analyzing, designing, and implementing software systems [Gro17]. It provides a means of notation for static and dynamic aspects of systems. In general, UML can model the structural and behavioral facets of systems. Structural diagrams are e.g. class diagrams, objects diagrams, package diagrams, or component diagrams. The behavior of systems can e.g. be modeled with use case diagrams, activity diagrams, or state machine diagrams [KSHE18].

When first approaching the analysis of requirements with a certain problem domain, one is typically interested in the concepts of the domain and how they relate to getting a structural overview. UML class diagrams model the static classes and attributes of system components and

how they relate to each other. They are typically employed during analysis and development. [KSHE18]. During analysis, class models aim to give an understanding of the structure of the problem domain and to document it. During development, there is more focus on technical details. Analysis class diagrams typically contain fewer notation elements and abstract more than development models focusing just on the problem domain by abstracting technical or implementation details [Stö05] [KSHE18].

Since MoreRedoc aims to give a first overview of problem domains, analysis class diagrams are generated. MoreRedoc translates requirements to basic class diagrams with the model elements *class, attribute, method, association* and *aggregation.*

Classes are domain concepts that serve as blueprints for objects with the same structure. They contain attributes, which describe the structural properties of instances of these classes. Methods illustrate actions or services offered by classes. Classes can have relationships with each other or themselves [Stö05] [KSHE18].

Though there are many types of relationships or possibilities to enrich classes with e.g. attribute visibility or relationships with multiplicities, MoreRedoc utilizes just basic concepts of UML class diagrams. The main reasons for this are the complexity and ambiguity of natural language, which make automatically generating meaningful models complicated. Therefore a decision was made to reduce the generated model's complexity to a minimum. Employed UML class diagram concepts in MoreRedoc are:

- *Classes* with *attributes* and *methods* describing an entity in the problem domain
- *Associations.* Associations are a type of relationship. They indicate a semantic relation between two classes
- *Aggregations.* Aggregations are another type of relationship. They indicate a part-of relation between classes [KSHE18]

4 Related Approaches and their Shortcomings

To answer the research questions about the existing approaches and tools for automatically translating requirements to class diagrams as well as their evaluation and finally their gaps and shortcomings, the literature corpus has to be analyzed first. This analysis can be found in the following section. By working with this analysis, a comparison can be made in the chapter about the shortcomings and identification of gaps.

4.1 Existing Solutions for Transforming Natural Language into Class Diagrams

11 fitting papers describing tools and approaches for translating English natural language into class diagrams automatically with the help of natural language processing were found. For every one of them, a short **summary of the approach** is given as well as information about the **evaluation of the generated model** and if the information is present, the **output format of the model**.

[PGK02] presents a tool named GOOAL for automatically generating class and sequence models. Its main idea lies in the simplification of sentences given in natural language. GOOAL tries to apply a cognitive schema to each sentence called *4W language*. This scheme attempt to compute a result for the question of what the subject is doing, if there are receivers of this action when the action takes place and if there are other participants. A conceptual framework called role posets is then applied to the simplified sentences. Therefore, a set of roles is defined to label each noun, based on the associated verb and their relation. Roles are e.g. agent, user, part, or whole. According to its roles and whether an entity has attributes, behavior, or relationships, a probability is calculated whether a noun will be modeled as a class or not. Unfortunately, it is neither given which NLP toolkit was used nor an evaluation or information about the output format.

[MMC05] describes a semiautomatic approach for translating natural language into class models. It consists of three steps. In a first step, a dictionary and rule-based tokenizer reads the documents containing the natural language requirements. Just valid sentences fitting the given sentence structure rules are kept. Then, a custom made POS tagger annotates every word and extracts the noun phrases. This automatic analysis is fed into a term management system, where users manually can either remove non-relevant terms or classify them either as an entity, attributes, or functions. Although no model is created directly, the tool's output consists of structured data describing the hand made classifications and relations between entities, which can then be used by users for modeling tasks. No evaluation is given.

Another methodology is published in [IO05]. Requirements have to be analyzed linguistically first. Therefore, every sentence is POS tagged and each word is classified as either being a or being related to a certain subject, a predicate, or an object. This enables splitting complex sentences into smallest sub sentences that are interconnected with conjunctions or relative pronouns. These connections can then be analyzed by transforming the knowledge gained from this first phase into a semantic net, where subjects and objects form the nodes and predicated the

edges. Then, heuristics are applied to find candidate classes and the relationships between them. Heuristics are rule-based, which means certain predicates translate to certain model elements. An example of this is the verb *to consist of*, which translates to a part-whole relationship that in turn indicates an aggregation. Another example is *is a kind of*, translating to an inheritance relationship. The approach is evaluated qualitatively.

[LDP05] provides an overview of the used rule-based techniques for translating natural language into class models. It is necessary to POS tag all sentences. Nouns can then either be translated to classes or attributes, whereas predicates translate to either methods or relationships between classes. Adjectives may describe attributes of classes. Relationships can be identified by matching them with a defined set of relations, e.g. *is type of*, *can be* or *is a* indicate inheritance, *does*, *is identified by* imply associations, or *contain*, *is part of*, or *belong to* indicate aggregations or compositions. Due to this being an overview of applicable techniques, there is no evaluation or output.

[GSU06] proposes a fully automatic translation into class models. Prerequisite for their mapping from natural language to model concepts is POS tagging. Coreferences are resolved heuristically, with personal pronouns being matched with the nouns of previous sentences. Then another heuristic is applied to the results. Single nouns get mapped to classes, if these classes get attributes in the following steps, while so-called noun-nouns (compound nouns, e.g. *customer number*) get split up - the first noun becomes a class, the second noun becomes an attribute for the class of the first noun. Verbs will be translated to methods, depending on their position after or between nouns which are modeled as classes. A quantitative evaluation is given, comparing the generated models of 100 samples with 500 lines each with reference models, which are expert-created. It is stated, that no classes or methods are missed, but 12.4% additional classes and 7.4% additional methods are introduced. The output format is not specified.

In [KS08] a tool named SUGAR is described. This tool creates class and use case diagrams out of the given requirements employing parts of the Stanford Core NLP framework. With this toolkit support, SUGAR tries to simplify complex sentences into structures that are more easily approachable during analysis. This so-called syntactic reconstruction tries to break down composed structures to simple structures. In a first step, coreferences are resolved employing JavaRAP, another NLP toolkit. Then, composed sentences containing multiple subjects, predicates, or objects are simplified and turned into separate simpler sentences using patterns of POS tags. Conjunctions are split at the conjunction into single sentences. These simplified sentences act as the knowledge base for the following class model generation. SUGAR then analyzes the noun phrases and divides them into four categories - the *redundant class* (e.g. ATM card and card, decided by a user-provided glossary, instances of this class are not modeled twice), the *adjective class* (for nouns phrases including an adjective, if the noun matches another class which is to be modeled, it is kept), the *attribute class* (for numerical values, information is provided via user input and these nouns will not result in classes), or the *irrelevant class*. Nouns are generally treated as classes in the generated model, while the attributes are adjectives in relation to a noun. Relationships are then modeled using a rule-based approach. Associations result from predicates like *has, next to* or *contained in*. Aggregations are modeled via composite nouns like in [GSU06]. Lastly, inheritance structures result from classes having similar attributes or methods. The commonalities are then extracted to a superclass. SUGAR displays the generated model in it's GUI, no way or format of exporting is specified. Unfortunately, no evaluation is presented.

Authors of [KS08] presents subsequent development on the tool SUGAR in [DB09] and [DS11]. The tool is renamed to UMGAR and its algorithm is extended. Though support for behavioral

models like collaboration diagrams are introduced, class diagram generation didn't change much, rules for recognizing relationships based on predicates were extended and compound noun analysis is expanded by hand-written rules to extinguish errors which were done during analysis in the old approach. Considering the tool's output, XMI support was added to enable the user to further use the generated models. Neither paper contains an evaluation of the approach.

The approach from [HA12] is derived from the founder of the entity relationship model (ERM) Chen's rules to compose such diagrams. While ERMs are primarily used to model semantic relationships as a base for designing relational databases, this approach tries to apply these rules to generate class diagrams. Once again, the input sentences are POS tagged and syntactically parsed to tag sentence parts like subjects, predicates, and objects, using the GATE NLP framework for these tasks. Then the rule-based approach for translating nouns and predicates to model elements follows. Generally speaking, all nouns become class or attribute candidates. Nouns matching elements from a blacklist containing generic words like *database*, *system*, *record* or *information* are omitted. Proper nouns like *London* or names will be cropped from the candidate list, too. All remaining non-composite nouns are mapped to classes. Attributes are derived from composite nouns like *customer number*, where the second noun gets mapped as an attribute of the class of the first noun, if present. Furthermore, if a genitive case like *the customer's id* is found, the noun after the genitive indicator will be mapped as an attribute to the first noun's class, if it exists. Analogously this will be applied to possessive cases like *id of the customer* in reverse order. Like other approaches, composite nouns are split into class and attribute, too. The analysis of transitive verbs between two classes tries to identify attributes and relationships. Attributes are identified by a predicate if the verb equals a form of *to have*. Relations are on the one hand derived by verbs following *in, on, to* or *by* which indicate an association, on the other hand once again, typical verbs like *include, contain, comprise, be divided* or *embrace* indicate aggregations. Finally, *is a* indicates a specialization of an inheritance relationship. This paper is one of the few giving a quantitative evaluation. Metrics used are *recall* (88%) and *precision*(93%), which are popular metrics for classification tasks. An explanation of the evaluation method is omitted, just the results are presented. The output is structured as XML and further useable for users.

[MP12] present a tool named RAPID. It takes a textual representation of the requirements and optionally a domain ontology in XML format as input and produces UML class diagrams, which are displayed graphically without a possibility to export them. No further information about the ontology is provided. RAPID relies on the sentence simplification rules from [KS08], [DB09] and [DS11]. The Apache OpenNLP framework is used for computational linguistic analysis to extract information about the POS tags and sentence structures. Before categorizing classes, their structure and behavior and the relationships between them, this tool tries to extract all concepts for these entities. At first, all stop words like *a*, *an* or *the* are marked. For every word without the stop words, the frequencies are calculated. This will serve as an indicator of the importance of each concept. Then, every word is normalized with a stemming algorithm. This reduces redundancy by transforming every word into its base form, e.g. by singularizing nouns or transforming verbs to their base form via a hand made implementation without framework support. After this, the normalized requirements are POS tagged and nouns, noun phrases, and verbs are extracted into a concept list. Then, to further reduce ambiguity, WordNet, a lexical database for the English language, is queried for synonyms for every concept. If a concept has a synonym in the concept list, it is marked as semantically related. Analogously, WordNet is queried for hyponyms. Matches are saved in a separate list indicating a generalization relationship. This knowledge serves as a database for object-oriented modeling. For identifying

classes, the concept list is filtered based on several predicates. If the frequency of a concept is larger than 2%, does not equal generic terms like *system* or *data*, is a proper noun, is not at a high level in the hierarchy of hyponyms, or can't have a value (according to a predefined list), it becomes a class. Compound nouns are dealt with like in [GSU06]. Considering relationships, at first, the list of hyponyms will be looked at. More general hyponyms will become parent classes if they are to be modeled as a class. Further analysis is based on all verbs in the full, non cropped concept list. If a candidate is a verb and its position in the full text is between to class concepts, it's either a composition if the verb matches indicating verbs like *to consist of* or *to include*, a dependency if it matches *to require* or *to use* or an association in every other case. No evaluation is given for the results.

[SSB15] structures the given input text with grammatical knowledge patterns. These patterns represent certain sentence structures, which can be analyzed further. Patterns represent e.g. simple structures like *subject - predicate - object* or more complex structures indicating dependent clauses or sentences besides conjunctions. The sentence structure is found with the help of the Stanford CoreNLP framework. Sentences matching the grammatical knowledge patterns are analyzed further. Therefore, certain positions in the patterns are labeled with tags indicating the role of the positions in the pattern, e.g. *actor*, *action*, *object*, *preposition* or *modifier* (for e.g. an adjective or compound noun). Based on these roles, the requirements are transformed into a domain model. Roles representing nouns are class candidates. Depending on the frequencies of the term and the modifier, classes are generated. If a noun is connected with a preposition like *to*, *of*, *from* or *for*, it is assigned to become a class, too. Attributes are identified by either a predicate in any form of *to have* between actors and objects or if a possessive relationship exists according to the specified sentence structures. Occurrences of *is a*, *type of* or *kind of* indicate an inheritance relationship between two classes. Other relationships or methods are not modeled. This tool is evaluated with the metrics *recall* (fraction of correct classifications among all classifications made by reference classifications), *precision* (fraction of correct classifications among the sum of correct and incorrect classifications) and *over specification* (fraction of incorrect classifications among total reference classifications). No information about the output format is given.

4.2 Shortcomings and Identification of Gaps

So far, 11 different papers describing approaches for translating requirements to class diagrams have been presented individually. To try to answer the second research question, which aims at identifying gaps in the existing tool landscape, a comparison between these tools is made in Table 4.1.

There are some techniques which are common in a few approaches and are therefore classified as follows:

- **Rule based simplification** refers to techniques which try to decompose sentences into smaller and more easily analyzable parts like the syntactic reconstruction in [KS08], which detects sentence structures matching specified patterns

- **Sentence pattern matching** describes all approaches for analyzing parts of those sentences which match a certain pattern. This allows for applying semantic patterns on these syntactic patterns assuming a certain structure and relationship between the parts of speech. An example is [SSB15]

- **Mapping based on POS** is a weaker form of applying semantic patterns, e.g. just tagging all nouns and mapping them to classes, like in [LDP05]

- **Rule based classification** applies specified mappings from a concrete instance of a part of speech to a UML model element, as can be seen in e.g. [HA12]

Tool/Approach	**Technique**	**Reusable output**	**Evaluation**
[PGK02]	Rule based simplification Sentence pattern matching	-	-
[MMC05]	Sentence pattern matching Manual Tagging	Structured Data (DSL)	-
[IO05]	Rule based simplification Rule based classification	-	Qualitative
[LDP05]	Rule based classification Mapping based on POS	-	-
[GSU06]	Coreference resolution Mapping based on POS	-	Quantitative (completeness) Qualitative
[KS08]	Rule based simplification Coreference resolution Mapping based on POS Rule based classification	-	-
[DB09] + [DS11]	Rule based simplification Coreference resolution Mapping based on POS Rule based classification	Structured data (XML)	-
[HA12]	Mapping based on POS	Structured data (XML)	Quantitative (recall, precision)
[MP12]	Rule based simplification Frequency based classification Hyponoym/Synonym analysis Mapping based on POS	-	-
[SSB15]	Rule based simplification Mapping based on POS Sentence pattern matching	-	Quantitative (recall, precision)

Table 4.1: Comparison of the related work

Three findings can be derived from the comparison of the related work in Section 4.1. To begin with, most approaches use either rule-based simplifications or rule-based mappings. This, of course, can reflect human reasoning, because e.g. certain verbs do indicate part-whole-relationships inherently. However, rule-based approaches have to have a complete set of rules to be able to produce a complete result. Combining this finding with the ambiguity of the English language and complex sentence structures makes this approach error-prone.

When comparing the output formats of the related work, it is noticeable that most tools don't offer functionality to export the generated models to industry standards like *XML Metadata*

Interchange (XMI[1]). This lacks the opportunity for the user to directly work with the generated models, especially if he or she is not just interested in the graphical representation of the model, but e.g. in the capability to edit or refine the generated models in a separate editor compliant to certain standard ways of interchanging UML models or to have a base for model-driven engineering.

The last finding is that in most cases there is either no or just qualitative evaluation of the generated models. This may lie in the fact that evaluation has to be concluded against a model of the provided requirements which can be accepted as the ground truth. Due to the inherently subjective nature of modeling, this task is cumbersome. Furthermore, an evaluation algorithm has to be specified.

These insights provide answers to the second research question - the identification of gaps. Future work has to implement a functionality to export the generated models in an accepted industry standard and has to provide a comprehensive model of quantitative evaluation.

[1] https://www.omg.org/spec/XMI/About-XMI/

5 MoreRedoc - a New Approach for Modeling Redocumented Requirements

As being discussed in Section 4.2, three of the shortcomings of the related work are a focus on rule-based techniques for mapping natural language to class model concepts, missing support for directly usable exported diagrams, and the absence of evaluation concepts. Therefore, an attempt is made to implement a new approach to try to bridge these gaps. In this chapter, this new approach for translating redocumented requirements in natural language into class models is presented. To begin with, requirements for a solution based on the findings of the literature survey are posed. Then the algorithmic approach is discussed, followed by a quantitative evaluation of the generated models against hand-tailored models of the requirements of two example projects.

5.1 Requirements

The main purpose of the proposed solution is to generate domain models out of redocumented requirements automatically without manual user input. For a possibility to directly see the results, the models have to be represented graphically, but furthermore, models have to be exportable to a standard of interchanging models to achieve the possibility of reuse. Therefore, the class models should be importable in UML editors like *ArgoUml*[1] and *StarUML*[2].

One problem in automatically generated domain models is possible incompleteness. Since a stated requirement for the tool is being able to modify the generated resources in graphical UML editors, cropping unnecessary model parts is rather uncomplicated for users. Therefore, overgeneration for the sake of the model's completeness in favor of the user having to prune overgenerated model elements is a trade-off that can be made.

Though the used redocumentation tool is *SoftRedoc*, the new solution should be open for extension and not just tailored to a certain redocumentation framework.

Due to the analyzed approaches of the literature corpus relying on NLP tools, a new solution should use comparable techniques to enable direct comparisons.

Another aspect is the heavy use of rule-based approaches in the related work. A new tool should employ such techniques as little as possible to avoid potential pitfalls like incomplete sets of rules or limitations due to input sentences being too complex, not fitting the specified rules.

5.2 Algorithmic Approach

In this section, the approach made to implement the stated requirements is discussed. Though focusing on the algorithm, there will be references to the implementation of MoreRedoc to illustrate and explain certain realizations of the requirements. MoreRedoc is implemented in

[1] https://github.com/argouml-tigris-org
[2] http://staruml.io/

Java 8, its source code is published under the GNU General Public License v3.0 and available on GitHub via *https://github.com/andauh/MoreRedoc*.

On account of MoreRedoc being developed to work with SoftRedoc, MoreRedoc takes two comma-separated value files generated by SoftRedoc as input. One contains requirements that are separated individually. The other contains domain concepts for each requirement. This tool will just use redocumented functional requirements. Though implemented to work with results from SoftRedoc, MoreRedoc is open to extension by handing input over to an implementation of a `moreredoc.datainput.InputDataHandler`, which is an interface containing a method that returns a list of requirements based on two file paths. One file path describes the location of the keywords for each requirement, the other to the natural language requirements for each requirement. This provides the possibility to implement support for other redocumentation tools or even just to files containing the unstructured requirements and a list of domain concepts that are to be modeled.

With SoftRedoc's `InputDataHandler`, a list of requirements containing the full text and domain concepts of each requirement and possibly a list of extra domain concepts, which are specified in other parts than the functional requirements (e.g. business objects or system actors), are initialized. In further steps, the unprocessed text will be analyzed linguistically to remove ambiguity. Therefore, with the help of the Stanford CoreNLP framework, four steps are passed through:

- **Coreference resolution**: As explained in Section 3.2, coreferences will be resolved. After a mention other than the original mention of a subject or object was found, said reference is replaced with the mentioned subject or object
- **Non-rule based decomposition of complex sentences**: Due to the analysis being less difficult when sentences are simplified as already analyzed in Chapter 4, complex sentences are simplified. To ensure that no information is lost, the text will remain as it is, the extracted sentences are stored separately. Decomposition is implemented by employing CoreNLP's constituency parsing API
- **Regularization and normalization of every sentence and word**: Stop words (*a, an,* and *the*) are removed. To ensure comparability, nouns are transformed into their base form. Therefore, genitive indicators are removed, e.g. transforming *customer's* to *customer* and ending *-s* indicating plural are removed. Once again, the results are stored separately. Word manipulation is done using JBoss DNA's inflector and handwritten rules for pruning genitive indicators
- **Normalization of the domain concepts**: The list of domain concepts for each requirement is normalized with the same rules

After these steps, a processed requirement contains its keywords in normalized form, the original text, and its regularized, normalized, coreference resolved and decomposed text. These processed requirements serve as a database for the analysis and mapping from parts of speech to class models. All steps are performed or at least called in the classes `moreredoc.project.data.Requirement` and `moreredoc.project.data.ProcessedRequirement`.

During the first step of the analysis, *possession tuples* are initialized. These tuples indicate an owner-owned relationship between domain concepts, where the preceding domain concept may consist of, aggregate or has an attribute in form of another domain concept. Heuristics for being a tuple are the following:

- if an extracted **domain concept consists of two or more concepts**, which are concepts themselves, a tuple will be generated. Example: *customer id*, *customer* and *id* are domain concepts. This will result in a tuple *(customer, id)*
- **compound nouns** in the full text produce a tuple, if the adjacent nouns in it are defined domain concepts from the requirements. For example, *customer id* will create a tuple (*customer, id*), if both *customer* and *id* are domain concepts
- **genitive constructs** result in tuples, if both parts of them are domain concepts. For example, *customer's id* and *customers' id[s]* will create a tuple, again
- lastly, **possessive relationships** indicated by the word *of* generate tuples, e.g. *id of [the] customer* results in a tuple

Both the initial and processed texts will be analyzed for the latter described three tasks.

The most important model elements to start with are classes because essentially everything revolves around them - if its either refining them with attributes and methods or modeling relationships between them. Therefore, the initialization of classes is the next step. Since SoftRedoc defines domain concepts, they are assumed to be the only candidates for either being a class or an attribute in a model. Every domain concept in every requirement is analyzed whether it can serve as a class candidate, attribute candidate, or both. Considering the previously defined possession tuples, the candidates are calculated. Assuming that a concept serves as an owner in a possession tuple, it is a class candidate. The reasoning behind that being that since it is a concept that is being refined and enriched with information of some kind, it has the potential to serve as a class. Attribute candidates in contrast are domain concepts that serve as the weaker owned-part in a possessive relationship. Having initialized both candidate sets, the classes can finally be initialized, too. Of course, each class candidate will be mapped to a class in the resulting model. If a concept is part of both sets, it can be assumed that the owned class is an attribute of the owning class and therefore a part of it, nevertheless important enough to be a class, too. An aggregation will be modeled between both classes, with the aggregating class being the class that is the owner in the possession tuple describing their relationship. If a class is just an attribute candidate to another domain concept that is a class, it will refine the class as an attribute. Lastly, for the sake of allowed overgeneration that can be easily pruned by the user, classes which are neither class nor attribute candidates will be modeled as classes to try to generate a complete model.

So far, the generated model consists of classes with attributes and possibly aggregations between said classes. The analysis of predicates in the sentences of the coreference-resolved and decomposed sentences can result in both methods for classes and associations between them due to verbs expressing either actions of objects or relationships between them. Examples are *"A customer **enters** his password"* or *"An order **consists of** articles"*. Hence, CoreNLPs information extraction is employed to generate relation triples as discussed in Section 3.2. CoreNLP's OpenIE (open information extraction) API[3] extracts binary relation tuples in the form of *subject-predicate-object* or unary tuples consisting of a subject and a predicate. Therefore, every extracted relation tuple is surveyed. Only the binary tuples with subject and object equaling a domain concept or the unary tuples with the subject being the same as a domain concept are kept. The obvious choice for unary tuples is to model the predicate as a method of the subject's class, but the binary tuples have to be interpreted first to be modeled correctly. Due to one

[3] https://nlp.stanford.edu/software/openie.html

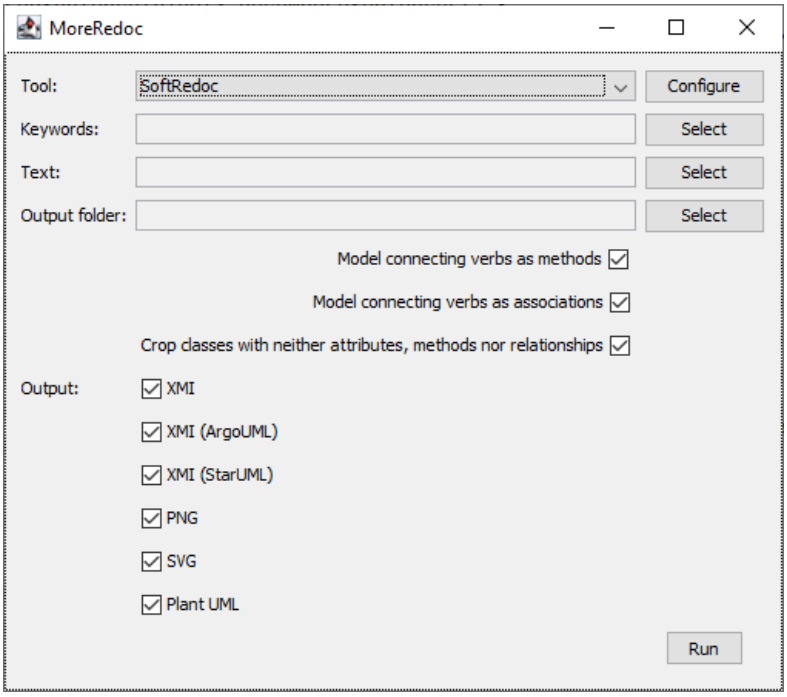


Figure 5.1: Screenshot of MoreRedoc

of the requirements for this tool is that rule-based analysis is to be kept to a minimum and overgeneration is not unwelcome, a *one-solution-fits-all* approach is used. Before starting the model generation, the user can specify whether to model connecting verbs - verbs, which are qualifying a relationship between a subject and an object in a tuple - are to be modeled as methods or associations or both. The last option will result in overgeneration. Associations are chosen because they indicate a semantic relationship between two classes, which is expressed by a predicate.

Lastly, according to the configuration of the analysis, classes without attributes, methods, or relationships with other classes can be pruned from the model.

All steps for the generation of classes, attributes, methods, and relationships between classes are called in `moreredoc.analysis.MoreRedocAnalysis`.

Up to now, an internal representation of the class model is produced. To illustrate the results, the open-source library *PlantUML*[4] is used. Besides class models, it can generate several other types of UML models as sequence diagrams, activity diagrams, or state diagrams. PlantUML uses a domain-specific language to express UML models. Therefore, the internal representation has to be transformed into it. PlantUML supports the generation of either graphical models or an XMI representation, which can be imported to ArgoUML or StarUML and therefore fitting all the requirements. The model can be exported in the portable network graphics (PNG) format, scalable vector graphics (SVG) format, XMI for ArgoUML and StarUML, or the PlantUML domain-specific language string.

Figure 5.1 shows the graphical user interface of MoreRedoc with the possibilities to choose the input files and output folder, the analysis configuration, and the output format.

[4]https://plantuml.com/

5.3 A Way of Quantitatively Comparing Class Models

It is a stated requirement that the results have to be evaluated quantitatively. As discussed in Section 4.2, the related work is rarely evaluated at all and if it is, the strategy of evaluating the generated models is not properly explained.

To evaluate the generated results, they will be compared against manually modeled domain diagrams. Two example projects were chosen to be modeled with MoreRedoc for this task. One is the running example described in Chapter 2, containing exactly the requirements and the domain concepts of each requirement that are specified in that chapter. This demo is hardcoded as an `InputDataHandler` in the git repository[5]. The other example consists of an example project called *Order Entry* for studies about traceability between requirement and code by the author of SoftRedoc, Harry Sneed. The textual requirements[6] as well as their redocumentation[7] are located in the repository, too. The redocumentation was generated by SoftRedoc. As it was manufactured from the author of SoftRedoc directly, it can be assumed that the tool was used in the best way possible and therefore the redocumentation output in form of the separated requirements and domain concepts is as correct as possible.

So far, each example project has a manually tailored domain model and a generated domain model. Somehow the relationships between the elements of the different models have to be compared to evaluate the correctness of the generated model when accepting the manually created domain model as the ground truth. This can be reduced to a classification task.

Binary classification tasks classify data elements in classes of interest or irrelevant classes. This renders four possible outcomes when comparing instances, whose actual classes are known, with the output of the classifier for this instance:

- an element is classified as relevant, its actual class is relevant, too (**TP, true positive**).
- an element is classified as relevant, but its actual class is irrelevant (**FP, false positive**).
- an element is classified as irrelevant, its actual class is relevant, too (**TN, true negative**).
- an element is classified as irrelevant, but its actual class is relevant (**FN, false negative**) [CDD18]

Transferring this to the evaluation of MoreRedoc, this classification can be done for each class of model elements in the ground truth models. Due to the only elements used being classes, attributes, methods, and two sorts of relationships, an evaluation will only be done for these classes and the ground truth models will resort to these elements, if possible. Furthermore, multiplicities will be omitted. Technically, the classification task classifies text elements from the requirements into model elements, with a ground truth class model being compared against. Nevertheless, the goal is to compare the two models and derive differences in numerical terms. Therefore, the translation of natural language to domain concepts is abstracted from and only the models are compared. To realize this, the following assumptions are made:

- True positives describe model elements that are *basically* the same in both models
- False positives are model elements that are part of the generated model, but not of the ground truth

[5] https://github.com/andauh/MoreRedoc/blob/master/src/main/java/moreredoc/datainput/DemoDataHandler.java
[6] https://github.com/andauh/MoreRedoc/tree/master/example/requirements
[7] https://github.com/andauh/MoreRedoc/tree/master/example/redocumentedRequirements

- True negatives are abstracted from since they represent the absence of a mapping from words to model elements
- False negatives describe model elements occurring in the ground truth but not in the generated model

The exact algorithm for calculating the number of elements in the result classes is described in pseudo-code in Figure 5.2. The available classes are true positives, false negatives, and false positives. To reduce the errors being based on ambiguity in relationships, their type is not relevant. If they're either aggregations or associations, just the corresponding classes count. From the number of elements in each class, the metrics precision and recall can be derived [CDD18]:

- **Precision** = $\frac{TP}{TP+FP}$
- **Recall** = $\frac{TP}{TP+FN}$

Precision can be seen as the reliability of a classifier. This metric indicates how often the mapping in MoreRedoc's case is correct in relation to all mappings done by the tool. Precision is influenced heavily by overgeneration, the more elements are overgenerated, the worse, i.e. lower the precision gets. In contrast, recall specifies the number of correct mappings related to all correct mappings, not just the ones done by this solution [CDD18]. Recall, therefore, implies the correctness of a model in terms of it having modeled all necessary elements according to the reference model. Table 5.1 shows precision and recall for the two evaluation projects for every model element class. The tool's analysis configuration was set to maximally overgenerate and prune empty classes, i.e. modeling connecting verbs as methods as well as relationships. Graphical representations of the models of the running example can be seen in Figure 5.3 and Figure 5.4. Models for the order entry project are located in the appendix in Figures A.1 and A.2

	P_{class}	R_{class}	$P_{attribute}$	$R_{attribute}$	P_{method}	R_{method}	$P_{relationship}$	$R_{relationship}$
Running example	100%	100%	100%	100%	70%	100%	34%	100%
Order Entry	57%	87%	33%	25%	32%	30%	40%	57%

Table 5.1: Evaluation of the generated models using the evaluation algorithm. P denotes precision for each model element class, R denotes recall

As can be seen in the evaluation, MoreRedoc works well with a small set of requirements. The larger a text becomes, the harder it gets to analyze it - both for humans as well as machines. When comparing the models for the Order Entry project (Figure A.1, Figure A.2), it is observable, that ambiguity and complex sentences introduce errors. For example, generated classes like *Customer Order*, *Bill* or *Item* don't have correspondence in the manually created model, nevertheless they do convey knowledge being semantically comparable to the information in the handcrafted model. The strict way of counting results in the result classes of the classification can't account for these types of errors. However, as being reflected in the metric precision, a good amount of model elements are overgenerated and to be pruned. Structurewise, the most important model elements are classes since they are refined and enriched with information. They, therefore, form the basic parts of a class model. Analyzing this aspect, MoreRedoc provides a recall of 87% in the Order Entry project. This indicates that of the relevant classes, 87% were modeled correctly and therefore providing a good base for refinement for the user.

C = {class, attribute, method, relationship}
$\forall\ c \in C$: $TP_c = 0$ initially
$\forall\ c \in C$: $FN_c = 0$ initially
$\forall\ c \in C$: $FP_c = 0$ initially
$M_{manually}$ = Ground truth model
$M_{generated}$ = Automatically generated model
$E_{c,manually}$ = Set of instances of class c in $M_{groundtruth}$
$E_{c,generated}$ = Set of instances of class c in $M_{generated}$

procedure MODELCORRESPONDENCE($element, model$)
  if ($element$ is a class $\wedge$
    $model$ contains a class that has the same or similar name as $element$) **then**
    return *true*
  else if ($element$ is an attribute to a class c $\wedge$
    MODELCORRESPONDENCE($c, model$) = *true* $\wedge$
    c has an attribute with the same or similar name as $element$) **then**
    return *true*
  else if ($element$ is a method to a class c $\wedge$
    MODELCORRESPONDENCE($c, model$) = *true* $\wedge$
    c has a method with the same or similar name and signature as $element$) **then**
    return *true*
  else if ($element$ is an aggregation or association between the classes c_1 and c_2 $\wedge$
    MODELCORDENCRESPONE($c_1, model$) = *true* $\wedge$
    MODELCORRESPONDENCE($c_2, model$) = *true* $\wedge$
    $c1$ and $c2$ are connected with that relationship) **then**
    return *true*
  else
    return $false$
  end if
end procedure

for all $c \in C$ **do**
  for all $e_{manually} \in E_{c,manually}$ **do**
    if MODELCORRESPONDENCE($e_{manually}$, $M_{generated}$) = *true* **then**
      $TP_c = TP_c + 1$
    end if
    if MODELCORRESPONDENCE($e_{manually}$, $M_{generated}$) = *false* **then**
      $FN_c = FN_c + 1$
    end if
  end for

  for all $e_{generated} \in E_{c,generated}$ **do**
    if MODELCORRESPONDENCE($e_{generated}$, $M_{manually}$) = *false* **then**
      $FP_c = FP_c + 1$
    end if
  end for
end for

Figure 5.2: Calculating the result classes for the evaluation

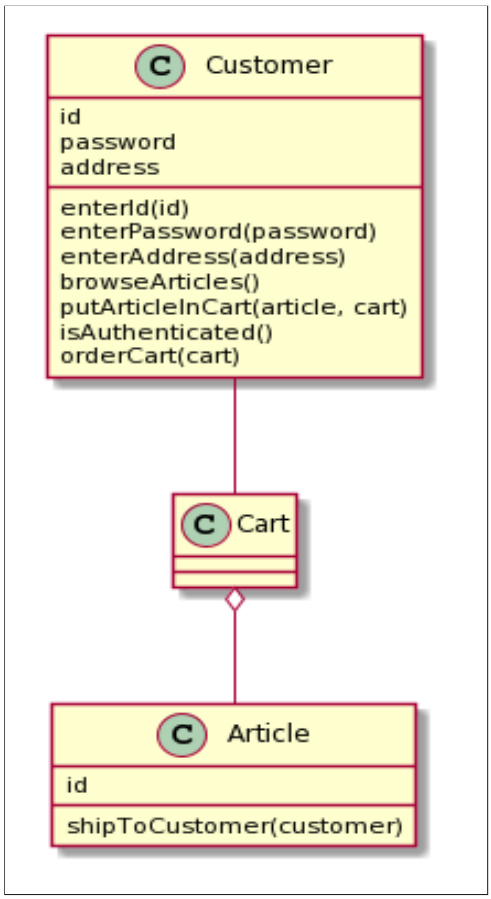


Figure 5.3: Manually designed domain model of the running example from Chapter 2

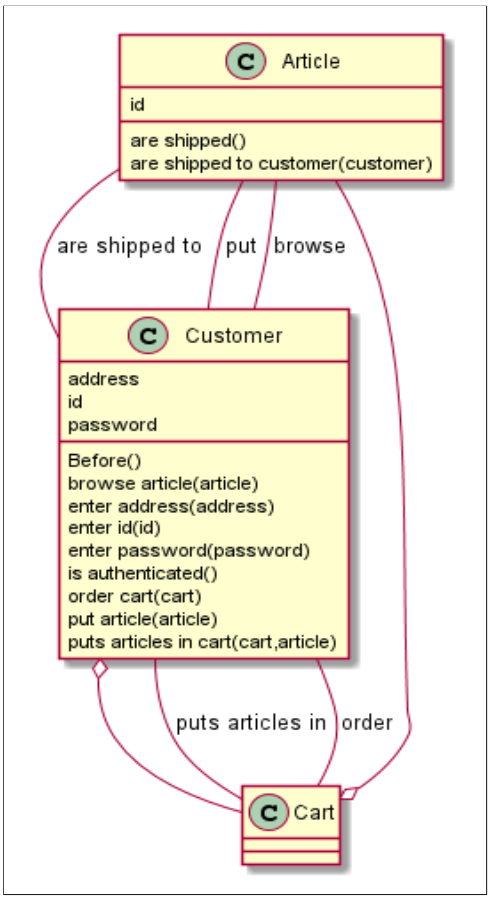


Figure 5.4: Automatically generated domain model for the same requirements

6 Conclusion and Outlook

In this thesis, the problem of automatically generating UML class models out of redocumented requirements with the help of NLP tools is treated. Three research questions were posed in Chapter 1.

The **first research question** asked *what approaches and tools exist for this task and how they are evaluated*. Chapter 4 discusses 10 different approaches. The approaches were summarized and analyzed. A majority of the approaches didn't provide an evaluation. Only two papers described an evaluation with classification metrics like recall and precision, but the evaluation approach was not described. Simply the results were presented without any method of verifying the evaluation or knowing how to calculate the metrics, which is non-trivial for comparing models.

To answer the **second research question**, which asked about *gaps and shortcomings of the compared tools*, their approaches were compared in Chapter 4.2. It was found that the majority of them used rule-based simplifications of complex sentence structures and rule-based mappings from parts of speech to model elements. The drawback of this approach is that rules have to be complete, which is a task hard to complete considering the ambiguity and complexity of natural language. Another observation was that the majority of these tools didn't provide a way to reuse the results. None of the tools could generate exports for industry standards to represent UML models like XMI.

A set of requirements for a new solution was derived from these shortcomings in Chapter 5. To push into these gaps, a new tool has to rely on non-rule based approaches, if possible and has to provide support for exporting the generated models in formats like XMI. Furthermore, the tool has to be evaluated quantitatively. Since no concrete evaluation method was tangible, an evaluation concept had to be designed. The classification metrics *precision* and *recall* were applied. It was taken into consideration, that the evaluation had to be done implicitly because no direct classification from parts of speech to model elements was made. To solve this, manually tailored ground truth models and generated models were compared directly. An algorithm for this task is presented in pseudo-code in Figure 5.2, describing how to calculate the true positive, false positives, and false negatives in this classification.

The **third research** question asked about an *evaluation concept and an evaluation of the proposed solution*. After presenting the algorithm, the designed evaluation algorithm was applied with mixed results. A simple set of requirements translated rather well into a class model, but the larger the requirement input became, the worse were the results. Just classes were comparable between a ground truth model and the generated model in the case of the larger requirement document. Attributes, methods, and relationships performed worse. One reason for this poor performance considering the metrics may be the strictness of the evaluation algorithm.

Despite the mixed performance, MoreRedoc has the potential to generate meaningful models. Limitations momentarily lie in the limited amount of model elements. Just two types of relationships are supported and multiplicities are not modeled at all. It would be interesting to try to find non-rule based approaches for these modeling tasks. Another possible enhancement can be derived from MoreRedoc's input data handling being programmed against an interface. This enables smooth integration of other redocumentation tools. Furthermore, MoreRedoc could be

transformed into a general-purpose tool for translating natural language into class models without a focus on redocumentation tools. A way of achieving this goal could be implemented by ignoring the input of keywords describing the domain concepts and just analyze the given text and extract the possible domain concepts by searching for noun phrases.

Another interesting approach besides just using NLP is generating statistical models for translating natural language into class diagrams like in [SKH13]. This combination of NLP and supervised learning for model creation runs in another direction than plain language analysis and seems promising.

Bibliography

[AdO05] Nicolas Anquetil and Káthia Marçal de Oliveira. Software re-documentation process and tool. *The 17th Conference on Advanced Information Systems Engineering (CAiSE '05)*, 2005.

[CDD18] Subramania Chandramouli, Saikat Dutt, and Amit Das. *Machine Learning*. Pearson Education India, Boston, MA, USA, 1st edition, 2018.

[DB09] D. K. Deeptimahanti and M. A. Babar. An automated tool for generating uml models from natural language requirements. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 680–682, 2009.

[DJ17] Elizabeth Dick, Jeremy Hull and Ken Jackson. *Requirements Engineering*. Springer, Cham, 4th edition, 2017.

[DS11] Deva Kumar Deeptimahanti and Ratna Sanyal. Semi-automatic generation of uml models from natural language requirements. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, page 165–174, New York, NY, USA, 2011. Association for Computing Machinery.

[Gro17] Object Management Group. Omg® unified modeling language® (omg uml®). https://www.omg.org/spec/UML/2.5.1/PDF, 12 2017. visited: 09.07.2019.

[GSU06] Anandha Mala G. S. and G. Uma. Automatic construction of object oriented design models [uml diagrams] from natural language requirements specification. In *Lecture Notes in Computer Science*, volume 4099, pages 1155–1159, 2006.

[HA12] Hatem Herchi and Wahiba Ben Abdessalem. From user requirements to UML class diagram. *International Conference on Computer Related Knowledge*, abs/1211.0713, 2012.

[ID10] Nitin Indurkhya and Fred J. Damerau. *Handbook of Natural Language Processing*. CRC Press LLC, 2010.

[IO05] M. G. Ilieva and Olga Ormandjieva. Automatic transition of natural language software requirements specification into formal presentation. In *Proceedings of the 10th International Conference on Natural Language Processing and Information Systems*, NLDB'05, page 392–397, Berlin, Heidelberg, 2005. Springer.

[JM14] Daniel Jurafsky and James H. Martin. *Speech and language processing*. Pearson Education, USA, 2nd edition, 2014.

[KS08] D. D. Kumar and R. Sanyal. Static uml model generator from analysis of requirements (sugar). In *2008 Advanced Software Engineering and Its Applications*, pages 77–84, 2008.

[KSHE18] Christoph Kecher, Alexander Salvanis, and Ralf Hoffmann-Elbern. *UML 2.5: das umfassende Handbuch*. Galileo Computing. Rheinwerk Verlag, Bonn, 2018.

[Kum11] Ela Kumar. *Natural Language Processing*. I.K. International Publishing House, Neu Delhi, 2011.

[LDP05] Ke Li, R. G. Dewar, and R. J. Pooley. Object-oriented analysis using natural language processing, 2005.

[Mas06] Dieter Masak. *Legacysoftware*. Springer, Berlin, 2006.

[MMC05] Stephen MacDonell, K. Min, and Andy Connor. Autonomous requirements specification processing using natural language processing. In *ISCA 14th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-05)*, October 2005.

[MP12] Priyanka More and Rashmi Phalnikar. Generating uml diagrams from natural language specifications. *International Journal of Applied Information Systems*, 1:19–23, April 2012.

[MSB+14] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Prismatic Inc, Steven J. Bethard, and David Mcclosky. The stanford corenlp natural language processing toolkit. In *In ACL, System Demonstrations*, 2014.

[PGK02] Hector Perez-Gonzalez and Jugal Kalita. Gooal: a graphic object oriented analysis laboratory. *17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '02)*, pages 38–39, January 2002.

[San90] Beatrice Santorini. Part-Of-Speech tagging guidelines for the Penn Treebank project (3rd revision, 2nd printing). Technical report, Department of Computer and Information Science, School of Engineering and Applied Science, University of Pennsylvania, Philadelphia, PA, USA, 1990.

[SE96] H. M. Sneed and K. Erdos. Extracting business rules from source code. In *WPC '96. 4th Workshop on Program Comprehension*, pages 240–247, 1996.

[SKH13] Viliam Simko, Petr Kroha, and Petr Hnetynka. Implemented domain model generation. Technical report, Department of Distributed and Dependable Systems, Charles University, 2013.

[SSB15] R. Sharma, P. K. Srivastava, and K. K. Biswas. From natural language requirements to uml class diagrams. In *2015 IEEE Second International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, pages 1–8, 2015.

[Stö05] H. Störrle. *UML 2 für Studenten*. Pearson Studium, 2005.

[SV20] Harry M. Sneed and Chris Verhoef. *From COBOL to Business Rules—Extracting Business Rules from Legacy Code*, pages 187–208. Studies in Computational Intelligence. Springer Verlag, Germany, January 2020.

A Appendix

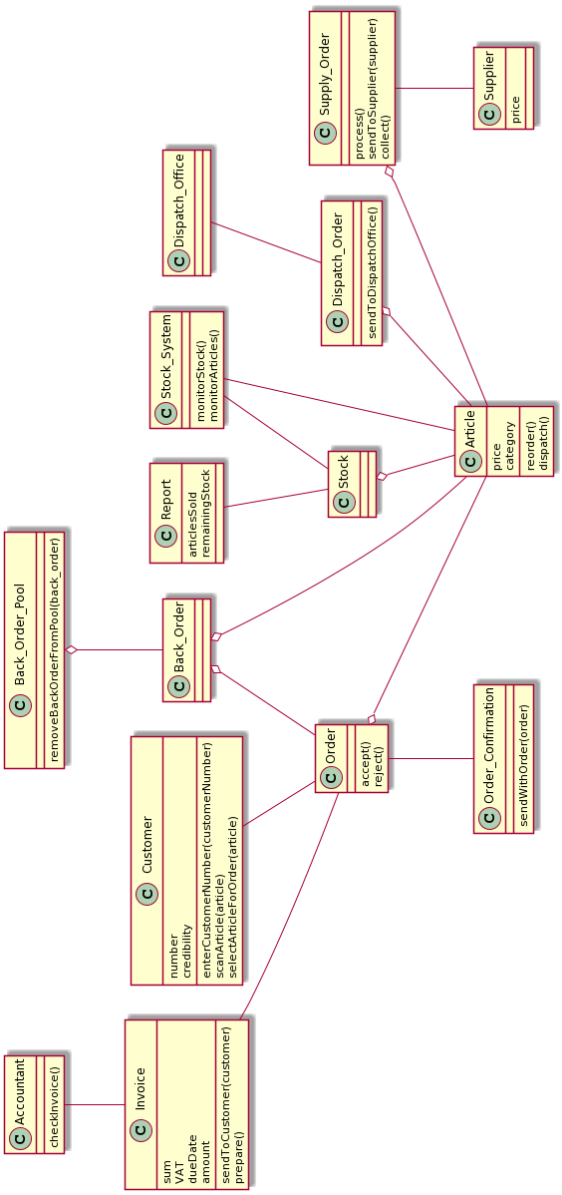


Figure A.1: Manually designed domain model of the Order Entry project

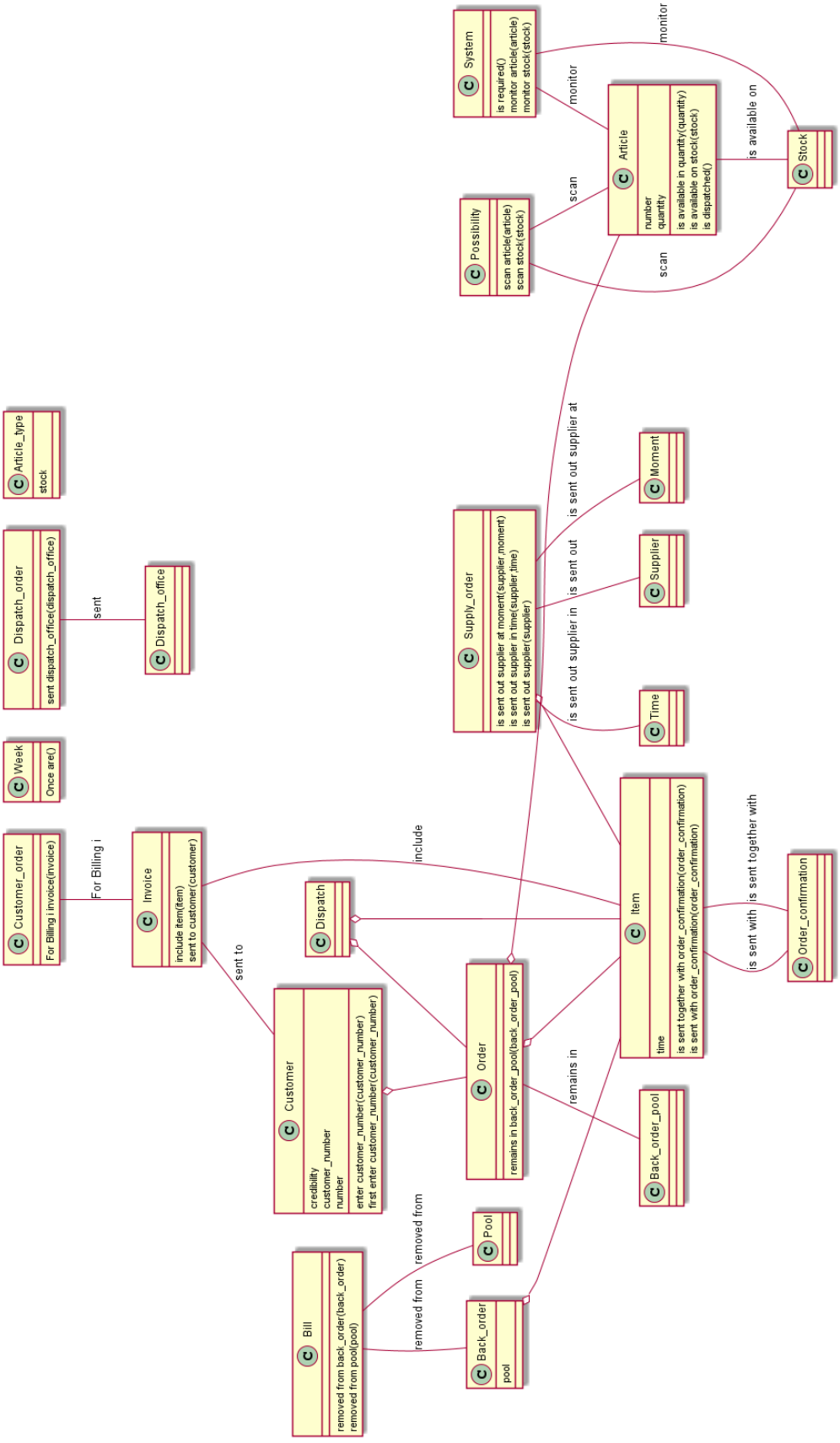


Figure A.2: Automatically generated domain model for the Order Entry project

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, August 26, 2020