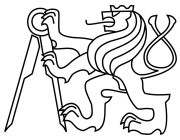
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Generating of UML entities from textual requirements specifications
Student:	Bc. David Šenkýř
Supervisor:	prof. Dr. Ing. Petr Kroha, CSc.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2016/17

Instructions

The goal of this thesis is to investigate the method of grammatical inspection and its suitability for processing of textual requirements specifications in the sense of ambiguity, inconsistency, and completeness.

1. Analyze the problem and the possibility to use appropriate systems like Stanford Core NLP to generate a static UML model.
2. Implement an editor in Java that helps to indicate whether the given word within a sentence is a candidate for a class, a relation, or an attribute.
3. Implement an interface (e.g., in XMI) that allows to export the found model to the next step of processing, e.g., using Enterprise Architect.
4. Evaluate the system using simple examples and compare it with the intuitive solution.
5. Summarize and evaluate the results reached.

It is a research work.

References

Will be provided by the supervisor.

L.S.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague November 30, 2015

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

Generating of UML Entities from Textual Requirements Specifications

Bc. David Šenkýř

Supervisor: Prof. Dr. Ing. Petr Kroha, CSc.

9th May 2017

Acknowledgements

I would like to kindly thank my thesis supervisor, Professor Kroha, for guidance, optimism, and valuable advice. I would also like to thank my family and friends for their continuous support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 9th May 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Šenkýř, David. *Generating of UML Entities from Textual Requirements Specifications.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Kvalita zpracování požadavků na softwarový systém zastává důležitou roli v rámci celého životního cyklu vývoje a údržby softwarového projektu – protože ostatní fáze jsou na ní závislé. Formulace takových požadavků jako text v přirozeném jazyce je běžnou praxí. Přirozený jazyk je však náchylný k řadě nepřesností jako je například nejednoznačnost, nekonzistence či neúplnost. Tato práce představuje CASE nástroj pojmenovaný *TEMOS*, který je schopný generovat fragmenty *UML modelu tříd* z textových požadavků na softwarový systém, a zároveň může být uživateli nápomocný v odhalovaní zmíněných nepřesností v textu.

Klíčová slova textové požadavky na softwarový systém, analýza textu, zpracování přirozeného textu při návrhu softwarového systému, UML model tříd

Abstract

The quality of Requirements Engineering plays an important role in the whole development life cycle of every software project – because the other phases depend on it. Writing requirements specifications in natural language is a common practice. The natural language is, unfortunately, prone to a number of inaccuracies like ambiguity, inconsistency, and incompleteness. This thesis presents the CASE tool called *TEMOS* that is able to generate fragments of the *UML class model* from textual requirements specification and also helps the user with the detection of some inaccuracies in the text.

Keywords software requirements specification, requirements engineering, text mining, text analysis, natural language processing in software engineering, UML class model

Contents

List of Figures

Introduction

Motivation

The significant phase of the *software development life cycle* (*SDLC*[1]) is undoubtedly the investigation and the processing of the *requirements specification*. Actually, by well-known standard phases of *SDLC*, illustrated in Fig. 0.1, the mapping of the requirements specification is the first one or one of the first output from the initial investigation of a software system utilization. Independently of *SDLC* model or used methodology[2] [1]. The form of such specifications is usually textual – because a contract for the development of a software system should define the scope of the system and its functionality based on the requirements specification.

The requirements specification is also the essential input for next development steps – such as modeling individual parts of a system. And the quality of the outputs of these steps is, of course, dependent on the quality of the input requirements. As Leonid Kof stated in his paper [2] – *Requirements Engineering is the Achilles heel of the whole software development process.*

The requirements investigation and other development processes are repeated with each new contract for the new software system. This led to the development of *CASE* tools. *CASE* is an used abbreviation of computer-aided software engineering tools. These are tools that assist the engineer during different stages of *SDLC* to simplify his or her task.

Given the severity of requirements specification and new possibilities of computer-aided support for natural language processing, we were motivated for design and develop of such CASE tool. The tool that assists mapping

[1] In some literature also used as *system development life cycle*.

[2] E.g. waterfall model, iterative and incremental model, v-shaped model, spiral model, agile development – SCRUM method, extreme programming, etc.

parts of textual requirements specification to corresponding fragments from static UML model. Investigation of possibilities of appropriate system for natural language processing and implementation of mentioned tool is the main goals of this master's thesis.

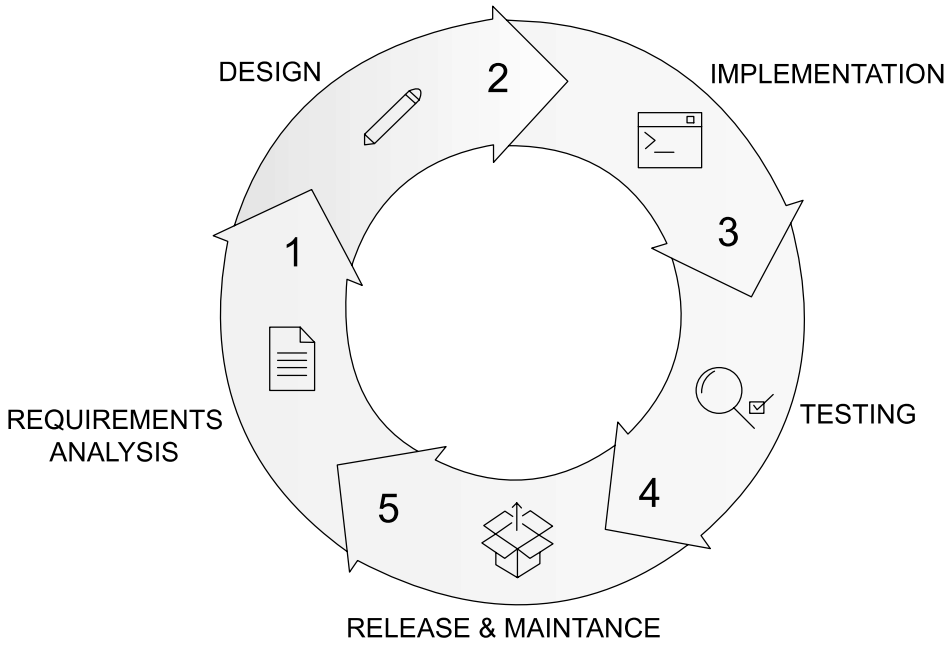


Figure 0.1: The Software Development Life Cycle (SDLC)

Text Structure

This thesis is organized in the following way that meets the software engineering view of the *software development process* – the *analysis* (chapters 1 and 2), the *design* (chapter 3 and 4), the *implementation* (chapter 5), and *the testing* (chapter 6).

Chapter 1 covers a brief overview of domains that relate to this thesis.

Chapter 2 is devoted to related work in previously introduced domains.

Chapter 3 is focused on solution design.

Chapter 4 serves our approach and overview of processing of textual requirements specifications.

Chapter 5 describes implementation of designed solution from the third chapter.

Chapter 6 is devoted to presenting solving examples using our application.

Finally, the conclusion summarizes achieved results and offers possibilities of the future work.

Chapter 1

Domain Introduction

This thesis is primary focused on the requirements engineering, the natural language processing, and the UML models. In this introductory chapter, we present a brief overview of these domains.

1.1 Requirements Engineering

The process called *requirements engineering* is the initial process by which it comes into contact analysts and client to clarify client's expectations of the future software. The *requirements specification* verified by the client (the stakeholder) is then the main output of this process. The requirements engineering primary consist of the following phases [3]:

- *elicitation* – meetings and appointments, observations of users, etc.,
- *analysis* – thinking and inventing, discussions, notes, etc.,
- *specification* – writing documents, using agreed notation, etc.,
- *verification* – other meetings, reading documents, presenting prototypes, clarifying the scope of functionality, etc.

All this can be repeated multiple times with various people from various departments.

1.1.1 Requirements Specification

The requirements specification covers the scope of the future software – which should be characterized by at least the following categories [3]:

- the *functional requirements* – requirements related to software functionality like the workflow[3] of tasks and activities that will be supported,
- the *interface requirements* – user interface design, software/hardware integration requirements, etc.,
- the *non-functional requirements* – properties of systems as a whole like performance (e.g. response time), accessibility and availability, extensibility and scalability, security, etc.,
- the *other requirements* – legislative, multilingualism, etc.

From the formal point of view, there exist standards like *ISO/IEC/IEEE 29148-2011*[4] [4], methodologies like *SWEBOK Guide* [5] or *Volere Requirements Specification Template* [6] (that is translated in various languages), as well as CASE tools that support better requirements organization or even manual assignment of requirements with parts of the model like *Enterprise Architect* [7].

The *SWEBOK Guide* (The Guide to the Software Engineering Body of Knowledge), at the time of writing this work in version 3, published by IEEE, includes a chapter dedicated to previously mentioned requirement engineering phases with guidelines and best practices.

Actually, as mentioned below, the most widely used approach is to write requests in natural language as non-structured text. And these are the specifications for which we are interested in.

1.2 Natural Language

The natural language plays in this field very important role. The market research [8] states that nearly 80 % of all specifications account for requirements are written in common natural language.

Many clients who award contracts on software projects, and IT projects in general, obviously operate outside the IT sector. In these cases, the natural language is surely the clear choice for the description of the expected functionality and requirements. Otherwise, even if the client knows more formal methods of requirements formulation like diagrams, models, etc., requirements formulation in the natural language is necessary because of a contract. The

[3] The set of inputs, the behaviour, and the set of outputs.

[4] ISO/IEC/IEEE 29148-2011: Systems and Software Engineering – Life Cycle Processes – Requirements Engineering. The successor of IEEE 830-1998: Recommended Practice for Software Requirements Specifications.

contract is then the primary relevant source, which can be assessed in the event of a legal case.

The advantage of using natural language is that can be interpreted both by the customer and by the analyst. However, its freedom without any formal restriction making them prone to a number of inaccuracies and incomplete expressions. Previous customer nescience of the formal methods of requirements formulation is now balanced by analyst nescience of the customer's business domain. What is natural for the domain expert from the customer team may not be fully evident for the analyst – also domain expert, but in a different domain.

Writing requirements may also be cooperative works of several people. In cases where is the contractor a company, it can be assumed that it is almost a rule. This is another source of inaccuracies and misleading descriptions. For example, simple describing a term in the text by several synonyms, may – in the case of little-know or domain specific term – result in a situation, where the analyst denotes synonyms as various terms.

Maintain requirements specification written in natural language complete and straightforward is a difficult task with a high probability of introducing new inaccuracies as illustrates a schema in Fig. 1.1 that is adapted by Professor Easterbrook's presentation [9].

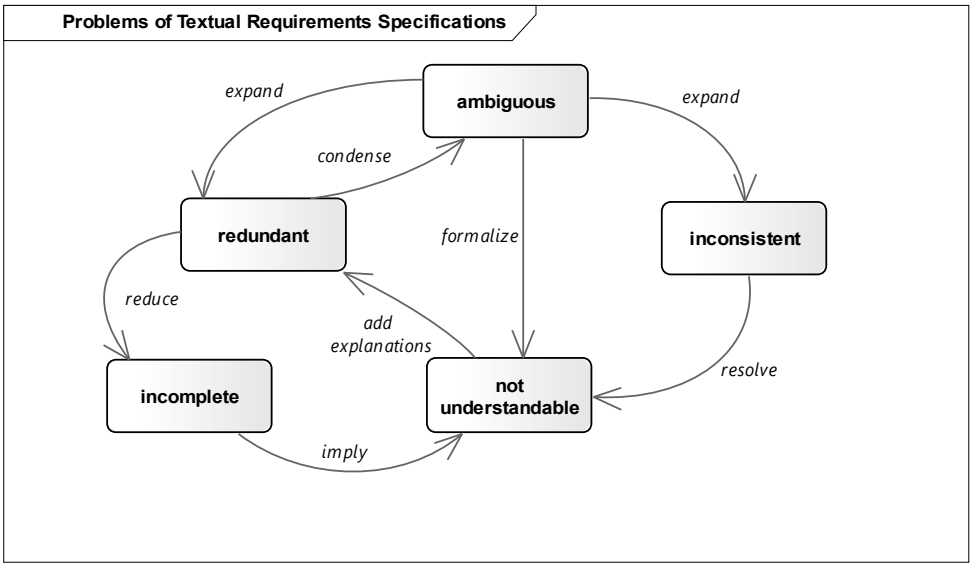


Figure 1.1: Problems of Textual Requirements Specifications

Some natural language pitfalls are covered in the third chapter where we describe their typical characteristics and introduce our approach to minimize them.

1.2.1 Natural Language Processing

A computerized processing of natural language requires the collaboration of engineers and especially linguistic experts. A interdisciplinary field devoted to this domain is called *computational linguistics*. Its origins date back to 1950s [10] where there were the first mention of a computer controlled translations of text from one language to another.

The increasing evolution of natural language processing (NLP) offers new opportunities in the fields of information retrieval, text mining, question answering, speech recognition, etc.

Nowadays it is possible to choose from a variety of natural language processing systems. Below are presented some of the complex NLP systems, however nice overview of standalone tools for various text processing tasks is available at [11].

- *Stanford CoreNLP* [12] – a suite of NLP tools written in Java language, but also has portations for other programming languages ,
- *Natural Language Toolkit (NLTK)* – written in Python,
- *Apache OpenNLP* – written in Java.

1.3 Ontology Databases

The natural language processing in the case of requirements engineering depends on various *domains (knowledge)*:

1. the domain of the scope of the business of the client organization (this domain can be very sophisticated – e.g. pharmacy, medicine, etc.),
2. the domain of internal organization knowledge (”know-how”) – terms, processes, workflows, etc. – the primary domain which is to be supported by the created software,
3. the domain of the software and the software engineering – the client specifies his or her view of the software behavior.

The human IT analyst is a domain expert for the third listed domain, but he or she is typically not focused on the first two listed domains. Therefore the solution of this task is not possible without the knowledge of a semantic of the text.

Computerized processing of natural language in this case can be supported by acquire semantic knowledge from some ontology database. Get the ontology related to the client's scope of the business may be difficult – for many sectors may not even exists. On the other hand, ontology databases for common language are available. Examples of such databases follow.

- WordNET
- ConceptNet
- DBPedia
- Freebase
- OpenCyc

1.4 UML Models

The model can be interpreted as implementation of *requirements*. It is common practice that the model precedes the implementation. So, it becomes an intermediary between the software analysis and the software implementation.

The *UML* is an abbreviation of *Unified Modeling Language* – the language designed for visual modeling (primary of *object oriented software systems*) [13]. The UML was accepted in 1997 by OMG[5] as the first open, industry standard object oriented visual modeling language. These days, the UML is de facto standard No. 1 in the software development life cycle and it is supported by previously mentioned CASE tools.

The adjective *unified* refers to various diagrams throughout the entire development cycle, independent of an application domain, a platform, and a programming language. That is why many software design patterns are expressed using the UML.

There are also a development approach called *Model Driven Development* (*MDD*) that is based on generating prototypes from the models – users can quickly get an idea of the system being developed.

[5]The Object Management Group – international not-for-profit technology standards consortium, founded in 1989 [14].

Although, as mentioned, the UML provides various diagram, we are focused on the UML class diagram in the next subchapter, because our goal is generating fragments of just this diagram.

1.4.1 UML Class Diagram

As the name suggests, class diagram deals with units called classes and relations between them. The notation is fully described in [13].

The example diagram in Fig. 1.2 describing the simple model that contains 3 classes – the *building*, the *hotel*, and the *group of hotels*. Every class has some attributes. Every *building* has an *area* (e. g. in square meters), every *hotel* has a *name* and *star rating*, etc.

This diagram also contains two relations. The first one between *building* and *hotel* means that every *hotel* is also a *building*. Therefore every building also has an *area*. The second relation is called *association* and in this context means that every *hotel* is part of some *group of hotels*.

These basic presented features of the *class diagram* will be sufficient for the purposes of this thesis.

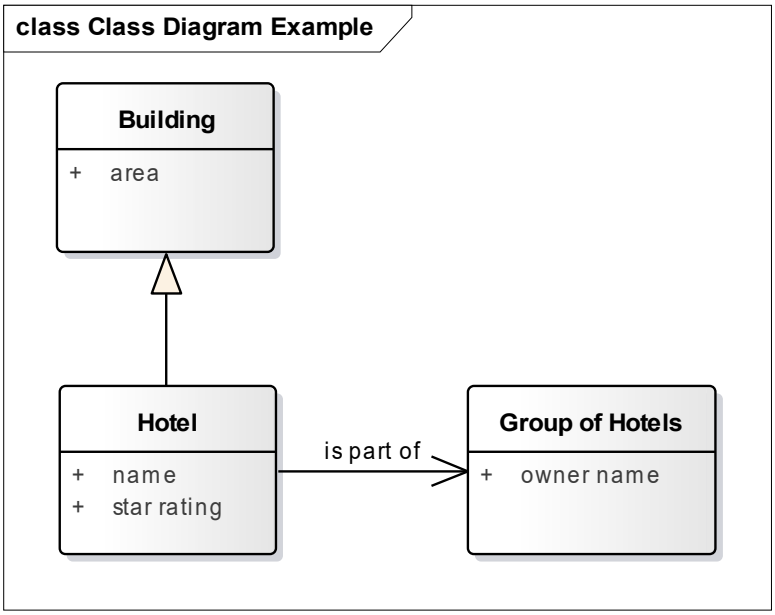


Figure 1.2: Class Diagram Example

Chapter 2

Related Work

There are many interesting papers proposing that requirements engineering should be supported by a CASE tools based on the linguistic approach. The cited papers are chronologically sorted.

1992 Professor Rolland with collective in their paper [15] from 1992 introduced a tool called *OISCI*. The mentioned tool processes French natural language. A key idea of this paper is *sentence patterns*.

Let's consider the following borrowed example sentences *"A subscriber has a name and an address."* and *"The colydrena have a pedistylus and a folicul."*. The first one contains well-know words – so the analyst probably introduces the *subscriber* as entity type and the *name* and the *address* as this entity's attributes. In the second case, the analyst probably similarly introduces the *colydrena* as entity type and the *pedistylus* and the *folicul* as this entity's attributes. However, the situation in the second sentence differs. The point is that decision can be taken without the knowledge of the meaning of the words *colydrena*, *pedistylus*, and *folicul*. This linguistic approach is based on the recognition of a particular *sentence pattern*. In this case, the pattern is described as <Subject Group><Verb expressing ownership><Complement Group>.

The approach presented in this paper targets the creation of the characterization of the parts of the sentence patterns that will be thereafter matched.

OISCI also uses a text generation technique from the conceptual specification to natural language for the validation purposes.

1997 V. Ambriola and V. Gervasi in their paper [16] from 1997 presented web-based system called *Circe* that primary processes Italian natural language (but may be also adapted for other languages). It consist of partial tools. For our purposes, the most interesting tool is the main one called *Cico*. *Cico* performs recognition of natural language sentences and prepares inputs for other tools – graphical representation, metrication, and analysis.

This paper presents idea that requirements specification may be connected with a corresponding *glossary* describing all the domain-specific terms used in the requirements. The glossary also handles synonyms of terms. Similarly to previous paper, *Cico* uses predefined patterns that are matched against the sentences from requirements specification.

2000 Leonid Cof in his papers [2] [17] broke down the NLP approaches into three groups based on the related work.

- The first one is related to lexical methods – methods that don't rely on basic NLP approaches like part-of-speech tagging nor any other parsing. These methods perceive the text as a sequence of characters and looking for terms (subsequences) that occur repetitively.
- Syntactical methods create the second one group. These methods use part-of-speech tagging or looking for special sentence construction. On this basis, they are able to distinguish objects and relationships.
- The last group is semantic methods – methods that interpret each sentence as a logical formula or looking for predefined patterns.

2001 *Linguistic assistant for Domain Analysis* (*LIDA*) is a tool presented in the paper [18] from 2001.

According to previous tools, *LIDA* is conceived as a supportive tool – it can recognize multi-word phrases, retrieving base form of words, present frequency of word, etc. – and doesn't contain algorithms for automatic model elements recognition. This decisions are fully user-side – the user marks candidates for entities, attributes, and relations (inclusive operations and roles). On the other hand, *LIDA* besides the text analyzing environment also contains the model editing environment that is naturally based on the user's marked candidates. The advantage is that every change in the model requires corresponding adjustment of

textual requirements – therefore, the model still corresponds with the input requirements specification.

LIDA also provides generating of the hypertext document based on the model that contains descriptions of classes (superclasses, class attributes, class relations and subclasses).

2015 A. Arellano, E. Zontek-Carney and M. A. Austin are authors of the paper called *Frameworks for Natural Language Processing of Textual Requirements* [19]. In this paper, there is presented tool *TextReq* based on *The Natural Language Toolkit* (NLTK). This toolkit is a open source platform for natural language processing in Python. It is an alternative to *Stanford NLP* platform that is primary focused on Java language, but also has portations for other programming languages (Python as well). The concept presented in this paper is the closest from mentioned papers to our approach.

Mentioned tool *TextReq* requires to run:

- Ruby,
- Bundler,
- Python,
- MySQL.

Chapter 3

Analysis and Design of Solution

This and the following chapters are already devoted to design and implementation of a prototype of our software tool called *TEMOS*. It is an acronym formed from **Te**xtual **Mo**delling **S**ystem. We would like to introduce a full-featured CASE tool, therefore, in addition to the possibility of highlight parts of the textual requirements and their mapping to the *UML fragments* by the user, *TEMOS* will provide automatic text processing and automatic fragments mapping. Due to domain-specific requirements, *TEMOS* will handle terms in an editable glossary. Based on the processed text, *TEMOS* will be able to generate found models in various formats to the next step of processing.

3.1 Software Requirements

In this thesis, a combination of words *software requirements specification* is much used term – this time we present requirements focused on the characteristics of *TEMOS*. Because the domain of our tool is evident from the previous chapters, we limit ourselves only to the structured form of *functional* and *non-functional software requirements* and expected *use cases*.

3.1.1 Functional Requirements

According to this chapter introduction paragraph and thesis assignment, we expect to meet these functional requirements.

- **F1**: The tool should be able to insert the textual form of requirements specification as a plain text.

- **F2a**: The tool should be able to provide editor in which the user is able to mark parts of text as candidates for a class, an attribute, or a relation. The tool should be able to help with this indication.
- **F2b**: The tool should be able to provide automatic indication of candidates for a class, an attribute, or a relation by text processing. The user still should be able to make changes in automatic indication.
- **F3**: The tool should be able to provide an editable glossary of terms from textual requirements.
- **F4**: The tool should be able to obtain a description of glossary terms from the on-line resources.
- **F5**: The tool should be able to generate found model based on the processed textual requirements in these formats:
  - XMI,
  - ECORE,
  - DOT.
- **F6**: The tool should be able to export the created glossary.
- **F7**: The tool should be able to save and to load a project.

3.1.2 Non-Functional Requirements

- **N1**: The tool should be independent of the user's operating system.
- **N2**: The tool should be written in Java (according to thesis assignment).

3.1.3 Use Cases

The overview of possible use cases is illustrated in *UML Use Case diagram* in Fig. 3.1. The indication of candidates for a class, an attribute, or a relation is hereafter also called as *an annotation*.

The expected typical use case scenario consists of the following use cases (steps).

- **UC1**: Insertion of the textual form of requirements specification as a plain text.
  - The user selects a *New Document* option.
  - The tool shows a text editor.

  - The user can type the text or use the copy and paste command.

- **UC2**: Customization of the editor.
  - The user select a *Settings* option from the menu.
  - The user can change a font size and colors of the highlights of a class, an attribute, a relation, or a selected word.
- **UC3**: Automatic indication of candidates for a class, an attribute, or a relation.
  - The user select a *Analyze (automatically)* option from the menu.
  - The tool automatically highlight candidates and map them to the UML fragments.
- **UC4**: Manual indication of candidates for a class, an attribute, or a relation.
  - The user select a *Analyze (manually)* option from the menu or use previously automatic indication.
  - The user can change or delete existing indication or create a new one.
- **UC5**: Manipulation with the glossary.
  - The user select a *Glossary* option from the menu.
  - The user can change or delete existing glossary term or create a new one.
  - The user can select a *Recognize* option and the tool then tries to get a description from on-line dictionary.
- **UC6**: A model validation.
  - The user select a *Model validation* option from the menu.
  - The tool validates a model and notifies the user of deficiencies.
- **UC7**: Exporting a model.
  - The user select an *Export model* option from the menu.
  - The user select a required format of the exported model – XMI, ECORE, or DOT.
  - The user can also export the created glossary by an *Export glossary* option from the menu.

- **UC8**: Save or load a project.
  - The user selects a *Project* > *Save* option or a *Project* > *Load* option, respectively, from the menu.
  - The user select an appropriate file in his or her computer that will be used for the save or the load operation.

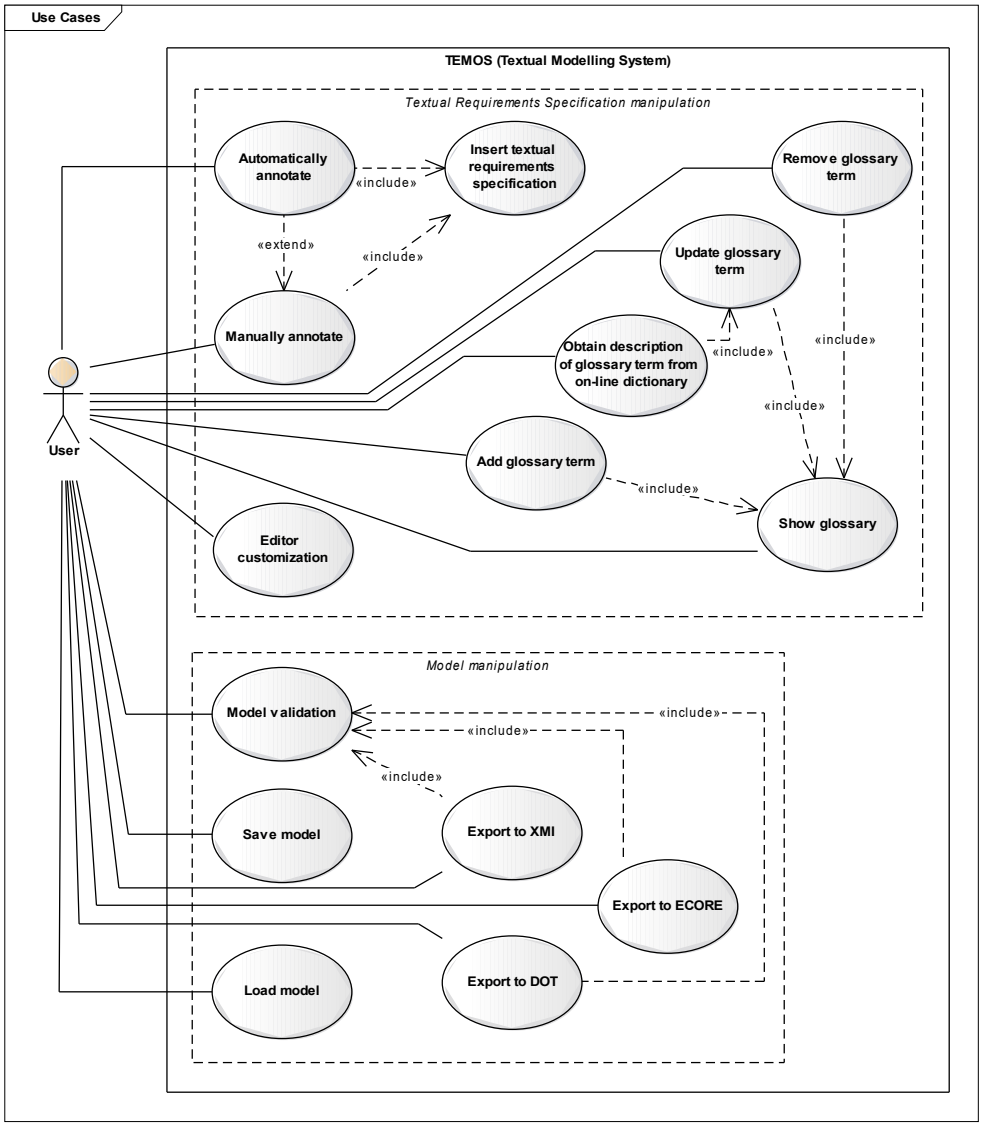


Figure 3.1: Use Case Diagram

3.2 User Interface

The tool provides a *graphical user interface* (GUI). Based on the fact that the commonly used screen resolution is aimed at landscape (it is longer in the width than in the height), also the GUI is oriented at landscape. The wireframe in Fig. 3.2 shows the GUI organization. On the left, there is a fixed width menu column. The top part is reserved for a navigation bar between views devoted to the annotations editor, the glossary, etc. Naturally, the remaining main space contains the selected view.

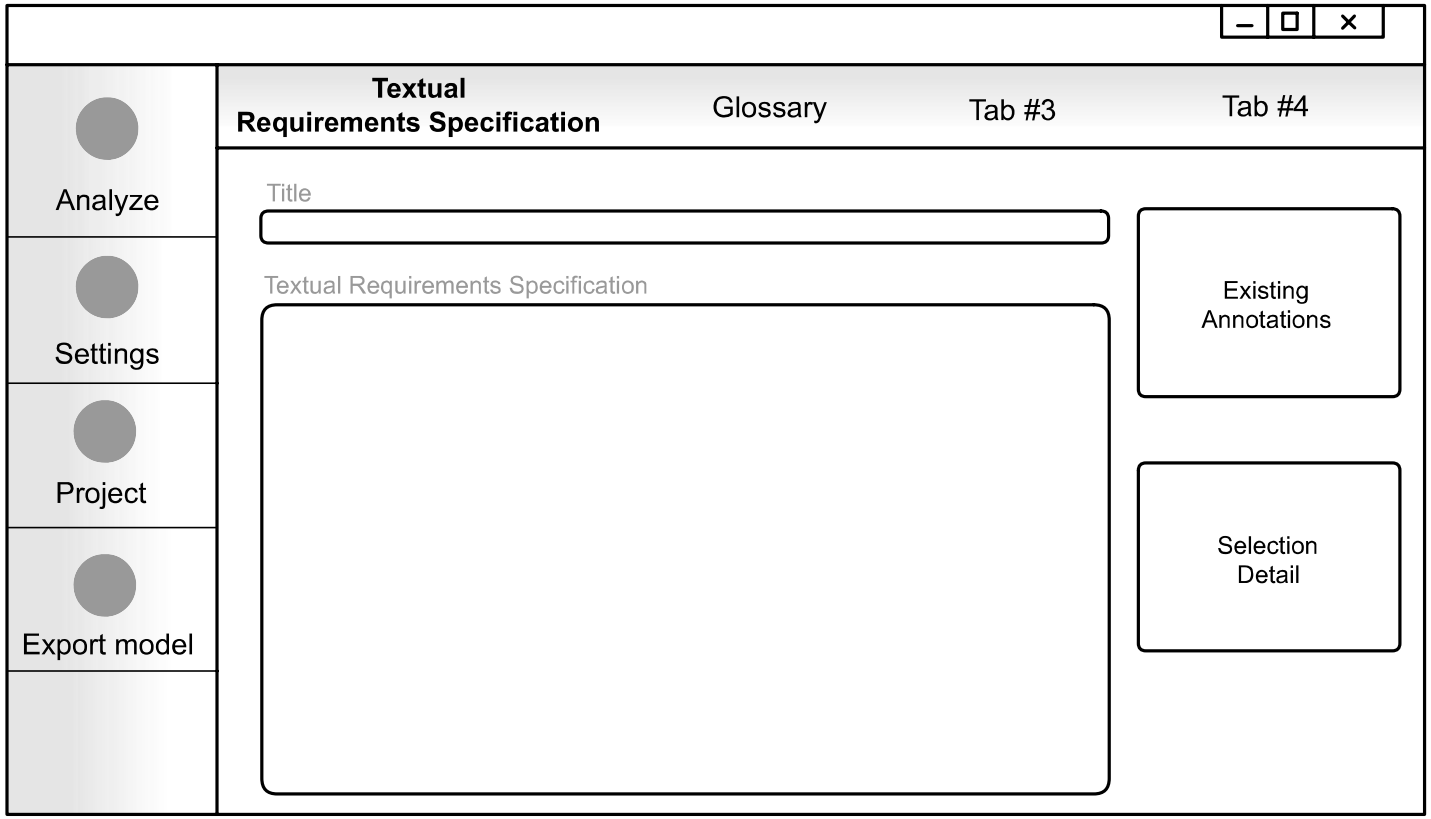


Figure 3.2: GUI Wireframe

Chapter 4

Our Approach

This chapter covers our approach to the main feature of our CASE tool – problems related to natural language as a form of expression of requirements for the software system. We try to handle repeating obstacles mentioned in the introduction and related work in the second chapter. It is really important to solve issues in requirements specification at the level of requirements engineering and thereby reduce uncertainties in the next phases of development (as well as too high cost of later changes [20]). It is obvious that the success or the failure of the software project is highly dependent on the quality of processing and understanding client needs and idea about the new software system.

With regard to subchapter 1.1, which show the various forms of the requirements specifications, we restrict our approach to the *functional software requirements*. This restriction reflects the focus of this thesis on the fragments of *static UML model*.

4.1 Overview

The schema in Fig. 4.1 demonstrates the whole analyze process implemented in our tool. The schema also contains swim lines that visualize which parts are computed by our algorithms (*TEMOS* swim lines) and which part is provided by *Standford Core NLP* framework (the middle swim line). We recall that our tool accepted any free text as the input.

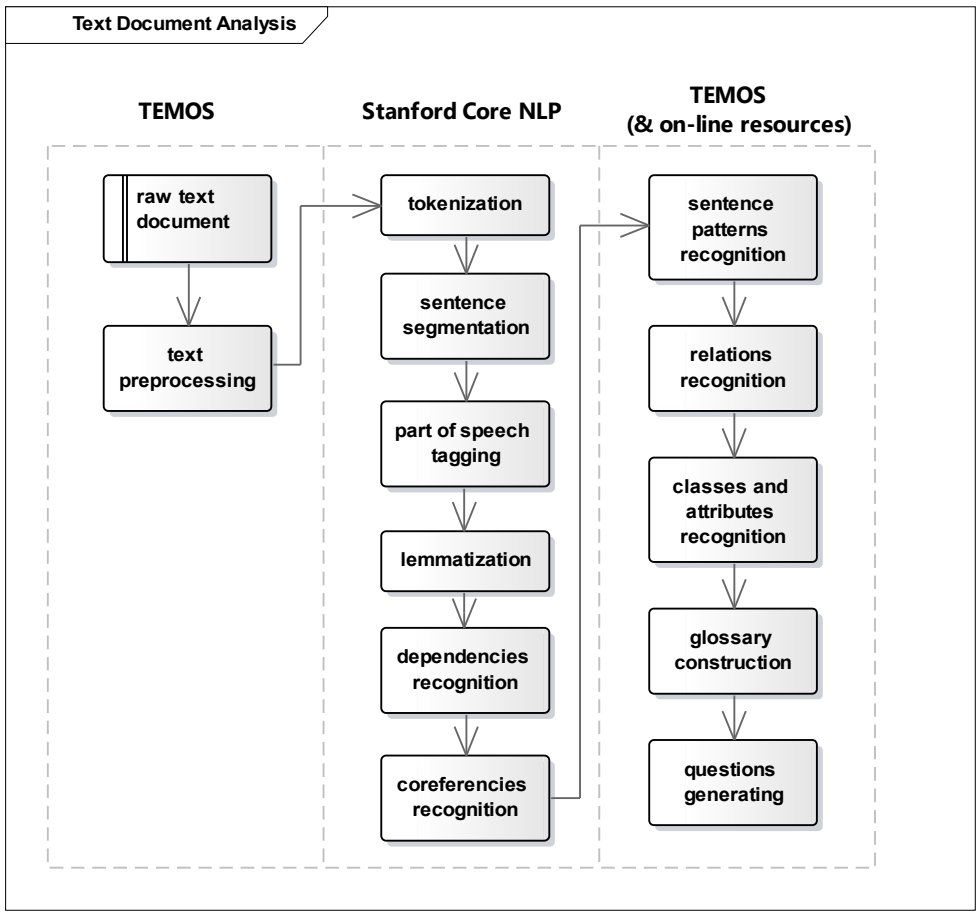


Figure 4.1: The Text Document Analysis

4.2 Text Preprocessing

In the first phase, the text is perceived as a plain sequence of characters. During the continuously testing of text processing, we identify some cases that is not properly handled by *Stanford Core NLP* system. That is the reason why we introduced text preprocessing phase. The following list illustrates problematic cases that are checked.

C1 Slash character within the meaning of the enumeration values.

- *Example:* His/her -> his or her.

C2 New-lines handling.

Example: In a numeric list, *Stanford CoreNLP* creates for every item number a separate line without the item that is up to the next line.

4.3 Nature Language Processing using Stanford CoreNLP

Text processing using *Stanford CoreNLP* is based on annotators. Considering [12], we present below the description of used annotators. The showed order is important due to dependencies of each annotator. The advantage is that all annotators do not have to run at once but can be added sequentially as needed.

The example figures showing the results of individual steps of annotation were created using a web version of *Stanford CoreNLP* system[6].

4.3.1 Tokenization

The tokenization is the first step provided by Stanford CoreNLP system. The corresponding annotator is called *Tokenizer Annotator*, and its goal is to parse the input text (the set of characters) into a sequence of tokens. A single token (a unit carrying significance) represents a word or a special character like an interpunction, etc. Whitespace characters (like spaces) don't represent tokens, but they are control characters for token recognition.

The result of tokenization process is shown in Fig. 4.2. The parts of the text bounded by the top curve with the *T* character represents individual tokens. As mentioned before, the dot character is also a separate token.

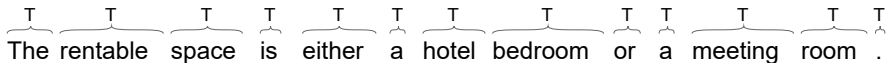


Figure 4.2: The Result of Tokenizer Annotator

4.3.2 Sentence Segmentation

Based on the tokenization, tokens are crowding into sentences in the second step. The responsible annotator is called *Words to Sentence Annotator*.

[6] http://www.corenlp.run

4.3.3 Part of Speech Tagging

The third annotator called *POS Tagger Annotator* provides part of speech (POS) annotation (tagging) of every token – such as noun, verb, adjective, etc. Interpunction and other special characters are annotated with the same character that represents.

The following example in Fig. 4.3 demonstrates the result of *POS Tagger Annotator* annotator.

Figure 4.3: The Result of POS Tagger Annotator Annotator

The English taggers use *the Penn Treebank tag set* [21]. This set makes a distinction between different meanings of the base part of speech tags. For example, the word *is* has the *VBZ* tag – *VB* tag means that the word is a verb and the expanded *VBZ* tag informs that the verb is in the present tense and is it the 3rd person singular verb. Similarly, nouns are categorized as singular or plural, etc.

This information is useful for *TEMOS* besides automatic annotation, also in the case in which the user manually annotate words – the editor strikes annotate a non-word tokens like mentioned interpunction.

4.3.4 Lemmatization

The lemmatization is a process in which the *Morpha Annotator* generates base forms (lemmas) for every token. E.g. the verbs *read*, *reading*, and emphreads have the same lemma *read*, the words *better* and *best* have *good* as its lemma, etc.

4.3.5 Dependencies Recognition

From our view, the most interesting annotator is called *Dependency Parse Annotator*. It analyzes the grammatical structure of a sentence and looking for relationships between words. The Fig. 4.4 presents the output of dependencies annotation performed on the same example that was shown in subchapter 4.3.3. The dependency direction is indicated by an arrow.

Every sentence has a one or more root words. These are the words that have no input dependencies. In the example above, there is one root word – *bedroom*. We can see that bedroom contains *a compound dependency* on

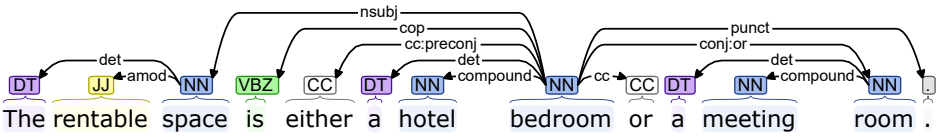


Figure 4.4: The Result of Dependency Parse Annotator – Enhanced++ Dependencies

the word *hotel*. It may indicate that *a hotel bedroom* is a multi-word term, similarly like *a meeting room*.

Stanford CoreNLP provides more than one dependencies annotator [22]. In our *TEMOS* tool, we use *Enhanced++ Dependencies*. This type of annotator, inter alia, introduces relation called *augmented conjunct*. If you compare Fig. 4.4 (Enhanced++ Dependencies) and Fig. 4.5 (Basic Dependencies), you can see that Fig. 4.4 a label of relation between a word *bedroom* and the last word *room* differs. The *Enhanced++ Dependencies* annotator labeled this relation as *conj:or* directly and we then not need iterate for the word represented a conjunction *or* – this is an example of *augmented conjunct*. A list of enhanced dependencies with examples is available on the web [23]. This site is created in English, but it also offers a list of dependencies for many other languages and their regional variants.

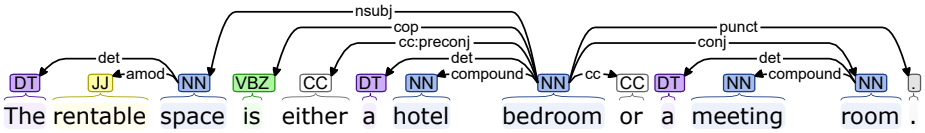


Figure 4.5: The Result of Dependency Parse Annotator – Basic Dependencies

4.3.6 Coreferencies Recognition

The last annotator by *Stanford CoreNLP* that we use is *Coref Annotator*. Its purpose is to identify to which words like pronouns refers – as show in Fig. 4.6.

Figure 4.6: The Result of Coref Annotator

4.4 Grammatical Inspection

This subchapter is focused on the third swim line of overview shown in Fig. 4.1. In our implementation of *TEMOS*, we exploit previously described annotation provided by *Standoford CoreNLP*. We use the grammatical inspection – primarily based on the *dependencies recognition* and *part of speech tagging* – to identify a grammatical role of words in textual requirements, i.e. object, subject, etc.

4.4.1 Annotations

The idea of mapping parts of the text document to *UML class diagram fragments* is based on the annotations. Similarly, as a user can highlight individual words in the text editor (e. g. *Microsoft Word*), he or she can also assign annotation in the editor included in *TEMOS* to individual words or group of words. Also, every *Standford CoreNLP* annotator uses custom annotations to extend tokens with new information.

TEMOS introduces these annotation types:

- *Class Annotation* – a basic annotation that can exist separately.
- *Attribute Annotation* – the annotation that is associated with the owner of the *class annotation* type.
- *Relation Annotation* – the annotation that mediates a link between 2 and more *class annotations*. Therefore, *relation annotation* contains collections of *source class annotations* and *target class annotations*.

Every annotation is identified by the *source word* (token) and can also consist of multiple words.

4.4.2 Patterns

According to [17], our approach of classes and relations recognition belongs to *semantic methods*. Similarly to paper [15] presented in related work chapter, we adapt approach of *patterns*.

The patterns-based recognition is based on the idea that the grammatical role of a word in a sentence corresponds with the role of the entity assigned to the word that the entity plays in the model.

The recognition process iterate trough *root words* introduced in subchapter 4.3.5. With regard to the part of speech tag of the current *root word*, it is then matched against defined patterns for recognize a class, an attribute, or a relation.

4.4.2.1 Class–Specialization Pattern

For example, the *class–specialization pattern* is defined by these rules:

1. The *root token* must be a *noun*. This root token will be the *class annotation* (C_1).
2. There must exist a *verb* (V) as a child of dependency of type *copula* (briefly *cop*). This verb must be *"to be"* verb.
3. There must exist a *noun* as a child of dependency of type *nominal subject* (briefly *nsubj*). This noun will be the *class annotation* (C_2).
4. If there exist any nouns as children of dependency of type *conjunct* (briefly *conj*), they will be the class annotations ($C_3 \dots C_n$).
5. The relation annotation is created with the verb V as a source token and C_1 as a source class annotation and $C_3 \dots C_n$ as target class annotations.

Let's take a look at the Fig. 4.4 again. The word *bedroom* as a root token meets this pattern. Therefore, a *hotel bedroom* and a *meeting room* are the specialization of *space*. For completeness, it should be noted that the *class annotation* is made up of the source noun and other nouns that are children of dependency of type *compound*.

4.5 Ambiguity

The significant part of the textual requirements specification analyze is building *a glossary of terms*. Every class candidate is automatically introduced as a term of the glossary.

Using the on-line ontology database *ConceptNet*[7], we can look for synonyms between terms and existing classes. These synonyms can be then grouped as one term in the process of creation of the model.

We also use the on-line English dictionary *Wordnik*[8] to provide a default definition of the glossary term.

[7] http://www.conceptnet.io
[8] https://www.wordnik.com

4.6 Inconsistency and Incompleteness

Documents containing no proper describe of entities may lead to the undefined behavior. Similarly, introduced entity which is not further used indicates missing information. The reasons of incompleteness may be:

- The customer forgot to mention it.
- The customer means that some facts are best-known, and he does not explain them.

We try to avoid the inconsistency and the incompleteness by checking if a class has at least one attribute and if it is in relation to another class. This check is part of the *model validation* offered by *TEMOS*.

We are also able to check relation in the way of correct usage of the verb. The English verbs can take 0, 1, or 2 objects, depending on the verb. Verbs without objects are called *intransitive*, and the other ones are called *transitive*. Using the dependencies recognition, we check if the verb has any objects. If no object is found, we check the verb against the list of intransitive verbs.

The list of *intransitive verbs* was composed using data from Wiktionary[9]. Unfortunately, this data contains also non-verb words and some transitive verbs, so we use our testing utility (see Appendix E) to remove them.

We use detected problems as a source for generating warnings and questions for the user.

[9] https://en.wiktionary.org/wiki/Category:English_intransitive_verbs

Chapter 5

Implementation

5.1 Technologies

5.1.1 JavaFX

Based on the thesis assignment and corresponding non-functional requirements **N1** and **N2** from subchapter 3.1.2, we chose the *JavaFX* software platform which provides the advantage of independence of the user's operating system.

The software platform *JavaFX* (the first version was released in 2008) is an open source project based on the software platform *Java*. *JavaFX* brings new possibilities to define a GUI and replaces previous UI toolkit *Swing* [24].

5.1.2 Formats of Exported Files

TEMOS allows the user to export the model in three different formats:

1. *XMI* (*XML Metadata Interchange*) – as it flows from the name, this format uses the *XML structure*. You can import these files into e. g. *Enterprise Architect* or *OpenPonk* (platform developed at our faculty).

2. *ECORE* – a custom structure based on the *XML structure* used in *Eclipse Modeling Framework* (*EMF*).

3. *DOT* – a custom structure for graph representation – suitable for instant visualization. It is used, for example, by *Graphviz*[10].

[10]There is also an on-line version available at: http://www.webgraphviz.com

5.2 Architecture

TEMOS was designed and implemented as *client-side application*. The application includes *Stanford CoreNLP framework* and therefore the primary functionality is available without having to connect to the Internet. If an Internet connection is available, *TEMOS* can use the on-line resources mentioned below.

This application is based on the *Model-View-Controller* (*MVC*) pattern that corresponds to the *client-side JavaFX applications* architecture [25].

TEMOS is also multithreading application. The most demanding time-consuming task is undoubtedly text processing using *Stanford CoreNLP*. To maintain application response, we used technique of background threads.

5.2.1 On-line Resources

The disadvantage of using free on-line resources may be their limitation on the number of requests. *TEMOS* uses:

1. Wordnik – on-line dictionary used to get the default definition of glossary terms – limitation: 250 requests per minute per our *API key*.

2. ConceptNet – ontology database used to find synonyms – limitation: 1 request per second per *IP address*.

If the limit is exhausted, the user is reminded that it is necessary to wait.

5.2.2 Package Structure

Package diagram in Fig. 5.1 shows a grouping of related classes and interfaces into the packages. Main package (*cz.cvut.fit.temos*) and every package within *controls* package also contain another package with graphical resources – which were not shown for clarity.

5.3 Graphical User Interface

The *GUI* in *JavaFX* can be created using old-fashioned code-behind way or a developer may use the *FXML files* which are defined similarly to the *HTML structure* of web pages. The great advantage of *JavaFX* is then styling the view (customizing colors, sizes, etc) using *CSS*.

The implemented GUI reflects draft from chapter 3. It is implemented as two overlapping layers – the top layer is hidden most of the time and is used

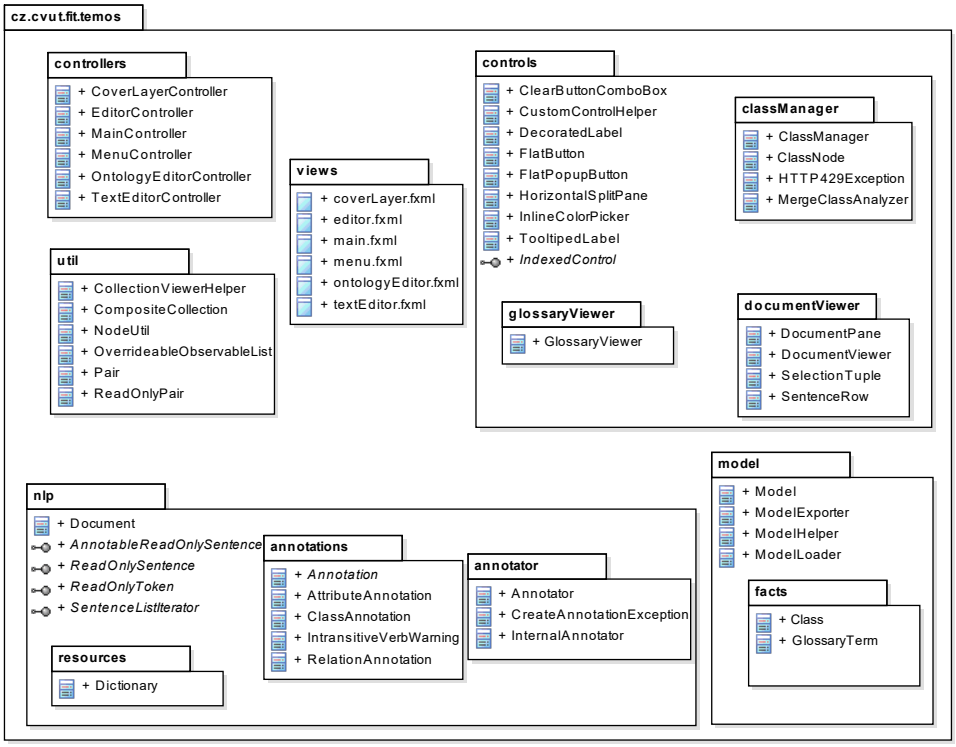


Figure 5.1: The Package Diagram

to display information messages and animation during the time-consuming operations.

5.4 Custom Controls

Traditional approach of creating a *GUI application* is composition a view using the controls. In the same way, it is possible create the GUI also in *JavaFX* with using predefined *JavaFX controls*. If the front-end developer need more customization of the GUI, he or she needs create a custom control – such control that provides more functionality than predefined control or a specific design.

As shown in package diagram in Fig. 5.1, the package called *controls* contains a lot of custom controls tailored to the needs of GUI. The meaning of most of them is apparent from their name – and therefore we only describe the one following control here.

5.4.1 HorizontalSplitPane Control

The one of the first implementations of *TEMOS* used the original *SplitPane control* provided by *JavaFX*. Unfortunately, during GUI implementation, we encountered a few bugs regarding behavior of that violate the specification. We reported these issues through *Oracle Java Bug Database* at website `http://bugs.java.com`. These issues were accepted and incorporated into the database with labels *JDK-8171071* and *JDK-8172029*. That was the reason for creating our custom *HorizontalSplitPane* control. It is based on the horizontal layout control *HBox* and it implements possibility of varying width of the left and the right content based on the divider position. For this purpose, our control has *dividerPosition property* and methods for setting content – *getLeftChildren* and *getRightChildren*.

Unlike the original SplitPane control, our control allows you to hide divider directly via *dividerVisible property*. Based on this, we implemented another feature – in the case of hidden divider and change of its position, the divider shortly animates to see where it is.

Developer can use our control directly in *FXML view* design similarly as presented in listing 5.1.

Listing 5.1: Example of Usage of *HorizontalSplitPane* Control

```
<HorizontalSplitPane dividerPosition="0.7"
                     dividerVisible="true">
        <leftChildren>
               <!-- Custom content. -->
        </leftChildren>
        <rightChildren>
               <!-- Custom content. -->
        </rightChildren>
</HorizontalSplitPane>
```

Chapter 6

Testing

6.1 Software Development and Testing

The application was continuously tested using *JUnit tests* based on the *JUnit4 framework*. The testing was primarily focused on the *Model class* (that provides a data layer across the entire application) and the *Document class* (that is responsible for parsing textual requirement specifications). These tests are part of source files of *TEMOS project* and can be found on the enclosed DVD. All functionality of the GUI was also tested with the minimum resolution declared in Appendix C.

6.2 Experiments and Results

Create models based on the textual requirement specification is a creative activity. Different analysts can create different models from the same requirements based on their experience. Therefore, testing the quality of generated models by our *TEMOS* tools and testing models, in general, is not an easy task and may be – from a certain point of view – subjective. For testing purposes, we take some examples from [20] which are listed below. The examples are organized in the following way. In the gray box, there are original requirements. These are followed by the generated model by *TEMOS* in the form of the *UML class diagram*. This diagram was acquired using *Enterprise Architect* after importing the model generated by *TEMOS* in the *XMI* format. Every example is also supplemented with a brief comment on the quality of the generated model.

6.2.1 Musical Store

6.2.1.1 Original Requirements

1. The musical store receives tape requests from customers.
2. The musical store receives new tapes from the Main office.
3. Musical store sends overdue notice to customers.
4. Store assistant takes care of tape requests.
5. Store assistant update the rental list.
6. Store management submits the price changes.
7. Store management submits new tapes.
8. Store administration produces rental reports.
9. Main office sends overdue notices for tapes.
10. Customer request for a tape.
11. Store assistant checks the availability of requested tape.
12. Store assistant searches for the available tape.
13. Store assistant searches for the rental price of available tape.
14. Store assistant checks status of the tape to be returned by customer.
15. Customer can borrow if there is no delay with return of other tapes.
16. Store assistant records rental by updating the rental list.
17. Store assistant asks the customer for his address.

6.2.1.2 Generated Model

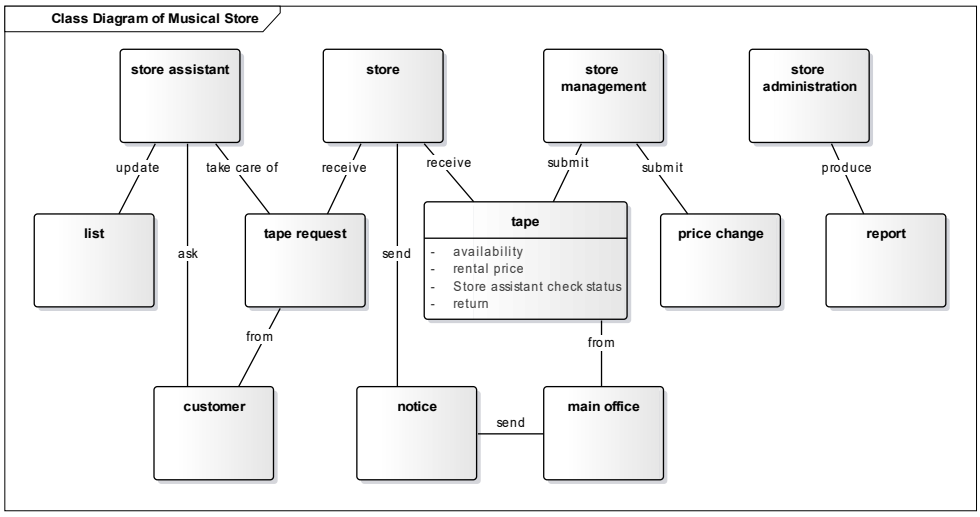


Figure 6.1: The Generated Class Diagram of Musical Store

6.2.1.3 Discussion

The generated model fairly reflects the part of the requirements that corresponds with *static UML model*. These requirements also describes some internal processes and system roles (like *the manager*, *the assistant*, and *the administrator*) that do not necessarily have to be class candidates.

A clear failure occurred in the 14th sentence when the first words (*"Store assistant check status"*) were marked as an attribute of the class *tape*. As can be seen in Fig. 6.2, this error occurred because *Stanford CoreNLP* system did not recognize the verb *check* – this word was incorrectly marked as a plural noun. The same problem also occurs in sentences (11), (12), (13), and (16).

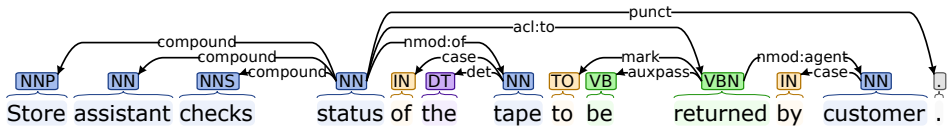


Figure 6.2: The Incorrect Result of Dependency Parse Annotator

6.2.2 Automatic Teller Machine

6.2.2.1 Original Requirements

1. Design the software to support a computerized banking network including both human cashiers and automatic teller machines ATMs to be shared by a consortium of banks.
2. Each bank provides its own computer to maintain its own accounts and process transactions against them.
3. Cashier stations are owned by individual banks and communicate directly with their own bank's computer.
4. Human cashiers enter account and transaction data.
5. Automatic teller machines communicate with a central computer which clears transactions with the appropriate banks.
6. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts.
7. The system requires appropriate record keeping and security provisions.
8. The system must handle concurrent accesses to the same account correctly.

9. The banks will provide their own software for their own computers; you are to design the software for the ATMs and the network.
10. The cost of the shared system will be apportioned to the banks according to the number of customers with cash cards.

6.2.2.2 Generated Model

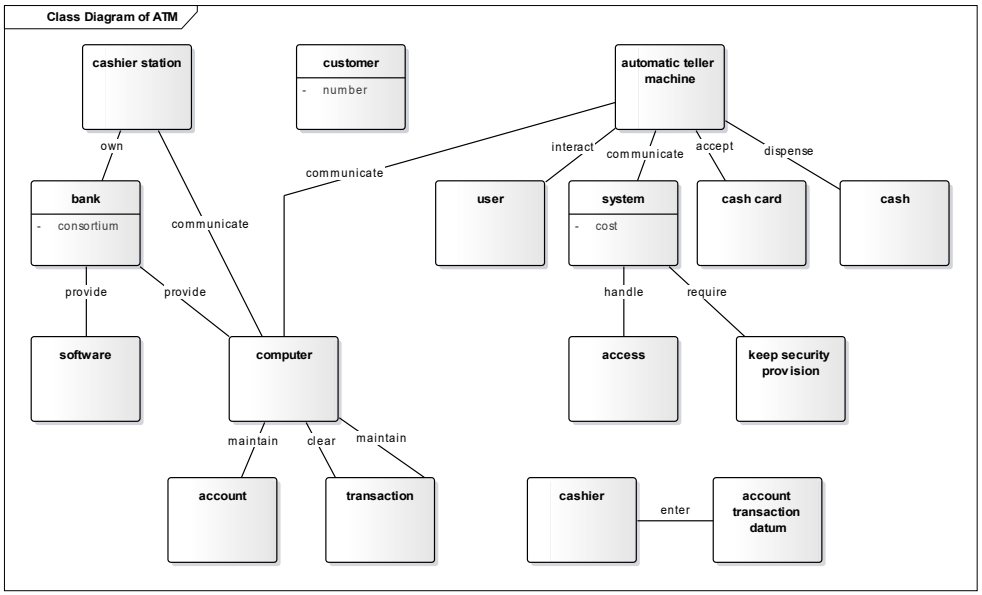


Figure 6.3: The Generated Class Diagram of Automatic Teller Machine

6.2.2.3 Discussion

Recognition of class and relation candidates meets presented entities in requirements. The automatic analysis only failed to distinguish the difference between *the human cashier* and *the automatic teller machine* with respect to the central system. The system is also marked differently in different sentences. Therefore, this model requires more modifications than the previous one.

6.2.3 ABC Video Rental

6.2.3.1 Original Requirements

1. Customers select at least one video for rental.
2. The maximal number of tapes that a customer can have outstanding on rental is 20.
3. The customer's account number is entered to retrieve customer data and create an order.
4. Each customer gets an id card from ABC for identification purposes.
5. This id card has a bar code that can be read with the bar code reader.
6. Bar code Ids for each tape are entered and video information from inventory is displayed.
7. The video inventory file is updated.
8. When all tape Ids are entered, the system computes the total bill.
9. Money is collected and the amount is entered into the system.
10. Change is computed and displayed.
11. The rental transaction is created, printed and stored.
12. The customer signs the rental form, takes the tapes and leaves.
13. To return a tape, the video bar code ID is entered into the system.
14. The rental transaction is displayed and the tape is marked with the date of return.
15. If past-due amounts are owed they can be paid at this time; or the clerk can select an option which updates the rental with the return date and calculates past-due fees.
16. Any outstanding video rentals are displayed with the amount due on each tape and the total amount due.
17. Any past-due amount must be paid before new tapes can be rented.

6.2.3.2 Generated Model

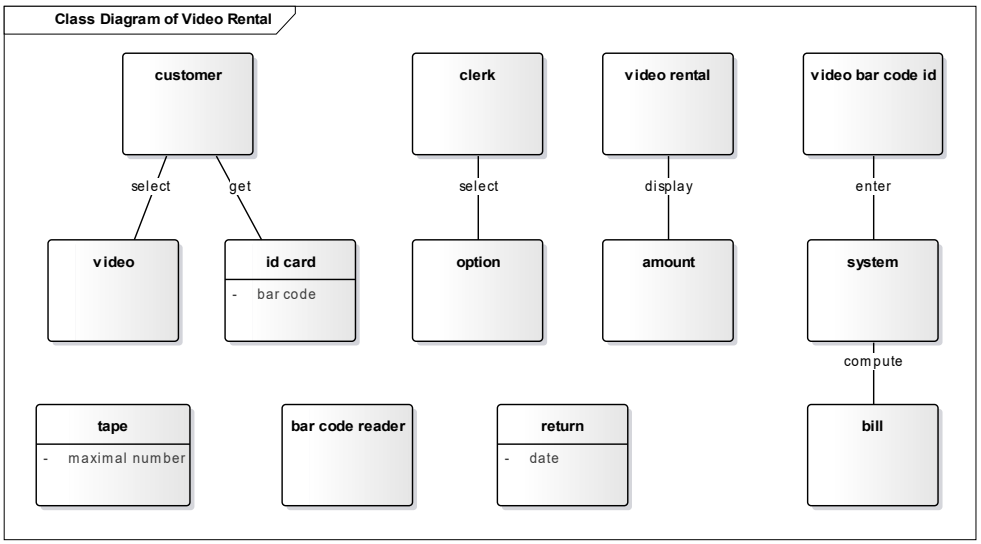


Figure 6.4: The Generated Class Diagram of Video Rental

6.2.3.3 Discussion

This example describes the internal processes of the video rental company. Although the generated model may look incomplete, the requisite entities were captured from the point of view of the class diagram. The requirements are primarily focused on the conditions and the following actions that are displayed through other diagrams.

6.2.4 Hotel Booking System

The last tested example was created by us and it mostly contains straightforward definitions (the structure of the sentence is in the format: *subject–verb–object(s)*) which are, of course, the best for automated processing.

6.2.4.1 Original Requirements

1. We would like to create a hotel booking system.
2. Our business group owns many hotels.
3. Every hotel offers some rentable spaces.
4. The rentable space always has a specified rent cost and area (measured in square meters).
5. The rentable space is either a hotel bedroom or a conference room.
6. The hotel bedroom has a unique room number and a number of beds.
7. The hotel bedroom may contain a television.
8. We would like to record a serial number and a display size of each television.
9. Every hotel employs at least one receptionist.
10. A receptionist takes care of reservations from customers.
11. Every booking has a customer and a selected rentable space.
12. The customer is identified by name, surname, and address.
13. The reservation contains a start date and an end date.
14. The booking also has a unique identifier.
15. Every conference room has a name and a maximum capacity.
16. The meeting room can contain a projection screen.

6.2.4.2 Generated Model

You can find the generated *XMI file* in Appendix D. The appearance of this *class diagram* corresponds to the status after accepting *class unification tips* (*booking=reservation* and *conference room=meeting room*).

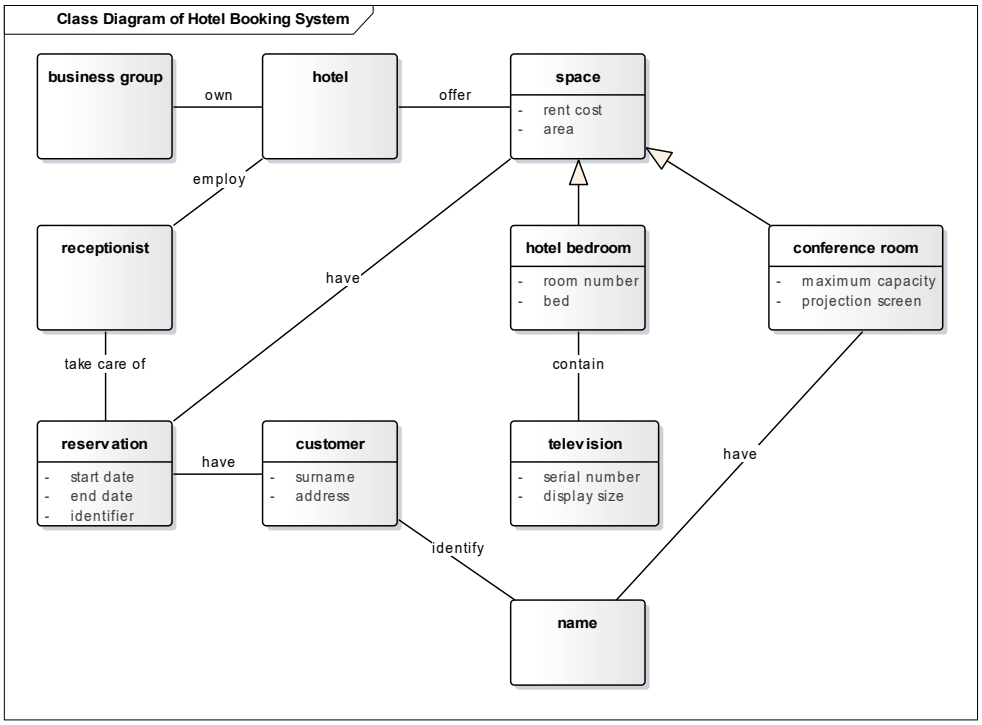


Figure 6.5: The Generated Class Diagram of Hotel Booking System

6.2.4.3 Discussion

This example shows recognition of all three types of annotations (*the class annotation*, *the relation annotation*, and *the attribute annotation*) and also shows case of *specialization* between the class *space* and subclasses *hotel bedroom* and *conference room*.

This result is already very close to real processing by the analyst. The human analyst would probably not designate *the name* as a separate class – *TEMOS* did not consider *the name* to be an attribute because it is a shared information between two classes. The second attribute of the class *hotel bedroom* should be labeled as *number of beds* with respect to the 6th sentence – unfortunately, the current version of *Stanford CoreNLP* does not generate the necessary link here.

6.3 Summary

TEMOS revealed most of the class candidates and the relationships between them in presented examples. Greater success of generated models occurred with requirements describing system properties.

In any case, the user can use automatic analysis as the initial model estimate, which can be further edited directly in *TEMOS*. After that, the user can export a model that matches his or her ideas for further processing in *XMI*, *ECORE*, or *DOT format*.

The screenshots capture the testing of the last example of the hotel booking system.

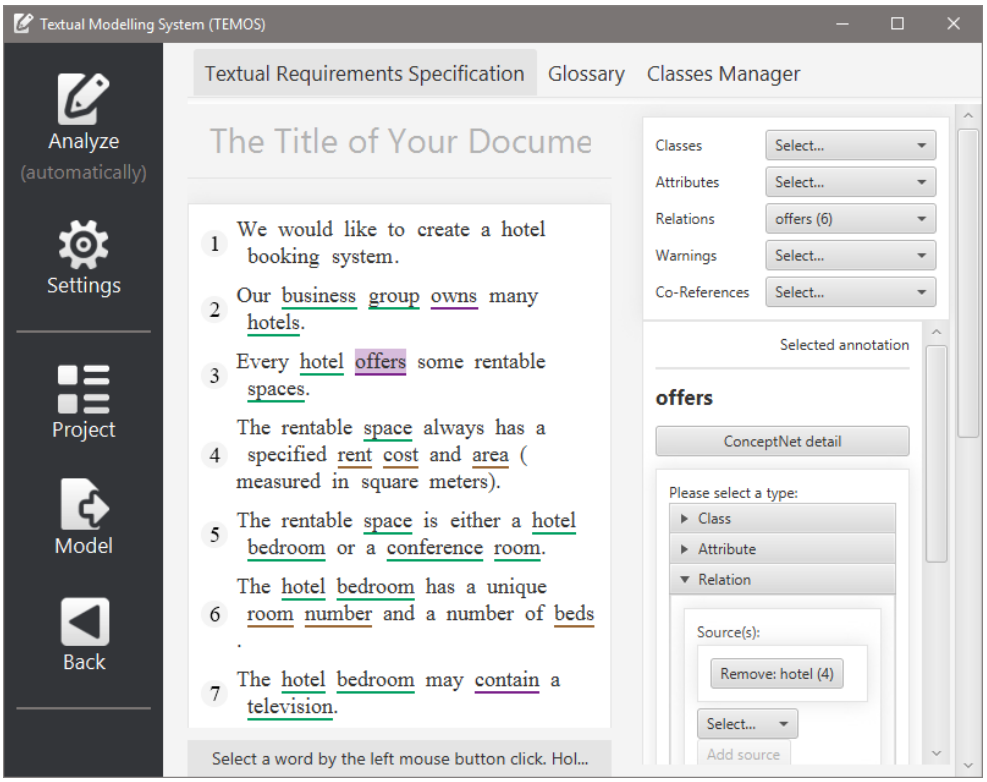


Figure 6.6: *TEMOS* – View of Specification

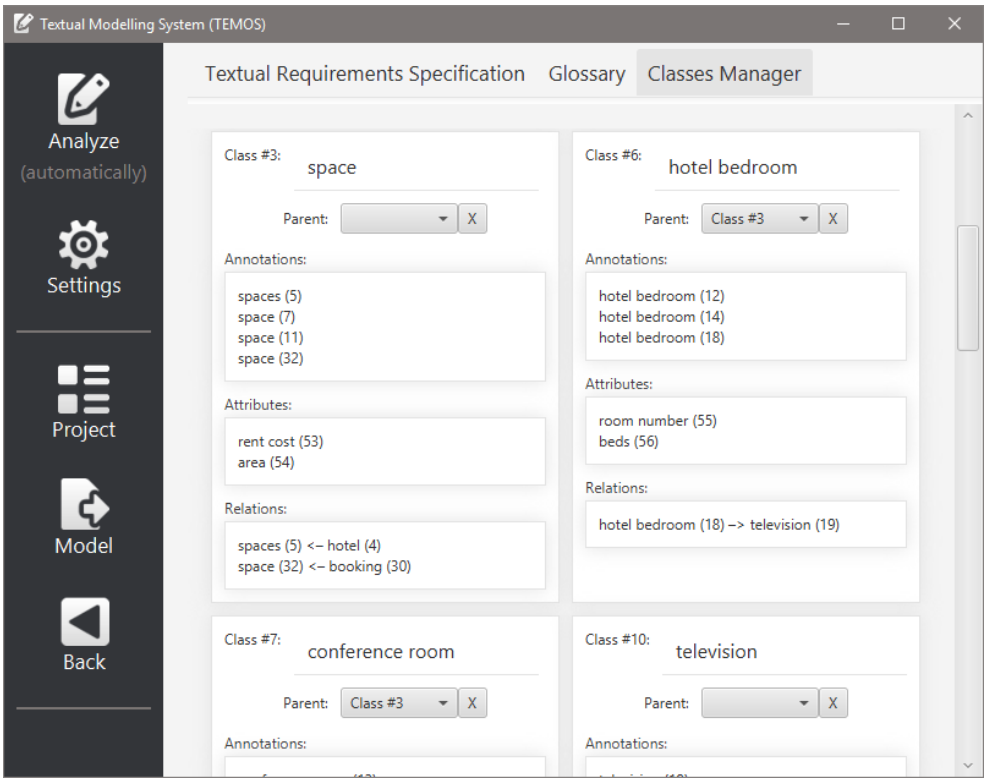


Figure 6.7: *TEMOS* – View of Classes Manager

Conclusion

Evaluation

We designed and implemented custom CASE tool *TEMOS* in *JavaFX* reflecting requirements from the chapter 3. It is a portable multi-threading application using built-in *Stanford CoreNLP platform* and on-line resources (like the English dictionary *Wodnik* and the ontology database *ConceptNet*).

TEMOS handles the requirements for the software system written in plain text. Based on the test from the previous chapter, *TEMOS* is able to generate drafts of *UML class models*. These models can be further modified or may be exported for further processing (*XMI* and *ECORE formats*) or may be visualized directly (*DOT format*).

During implementation, we also found the previously mentioned bugs that we had reported and which may be useful for the *JavaFX community*.

Future Work and Ideas

Due to the high computational complexity (primarily due to *Stanford CoreNLP* analysis), the client–server architecture might be interesting. This architecture could also be used to collect data for the improvement of the analysis process.

Bibliography

[1] ISAIAS, Pedro and Tomayess ISSA. *Information System Development Life Cycle Models*. In: *High Level Models and Methodologies for Information Systems* [online]. Springer New York, 2015, pp. 21–40 [accessed: Nov 5, 2016]. ISBN: 978-1-4614-9254-2. Available from http://dx.doi.org/10.1007/978-1-4614-9254-2_2

[2] KOF, Leonid. *Natural Language Processing: Mature Enough for Requirements Documents Analysis?* [online]. 2005 [accessed Nov 14, 2016]. Available from: https://wwwbroy.in.tum.de/publ/papers/Is_NLP_Mature.pdf

[3] KRÁTKÝ, Tomáš and Bohumír ZOUBEK. *Requirements Engineering* [online presentation]. 2016 [accessed Nov 23, 2016]. Available from: http://profinit.eu/wp-content/uploads/2016/12/02_Requirements_intro-1.pdf

[4] ISO/IEC/IEEE 29148-2011. *Systems and Software Engineering – Life Cycle Processes – Requirements Engineering* [online]. 2011 [accessed Dec 12, 2016]. Available from: https://doi.org/10.1109/IEEESTD.2011.6146379

[5] *About SWEBOK* [online]. IEEE Computer Society, 2004. [accessed: Nov 28, 2016]. Available from: https://www.computer.org/web/swebok

[6] *Volere Requirements Specification Template* [online]. The Atlantic Systems Guild. [accessed: Nov 28, 2016]. Available from: http://www.volere.co.uk/template.htm

[7] SPARX Systems. *Enterprise Architect* [software]. Version 12. Available from: http://www.sparxsystems.com.au

[8] LUISA, Mich, Franch MARIANGELA and Novi Inverardi PIERLUIGI. *Market Research for Requirements Analysis Using Linguistic Tools* [online]. [accessed Oct 18, 2016]. DOI: 10.1007/s00766-003-0179-8. Available from: http://link.springer.com/10.1007/s00766-003-0179-8

[9] EASTERBROOK, Steve. *Lecture 17: Requirements Specifications* [online presentation]. 2004 [accessed Dec 27, 2016]. Available from: http://www.cs.toronto.edu/ sme/CSC340F/slides/17-specifications.pdf

[10] HUTCHINS, John. *Retrospect and prospect in computer-based translation* [online]. 1999 [accessed Nov 27, 2016]. Available from: http://www.hutchinsweb.me.uk/MTS-1999.pdf

[11] NEUBIG, Graham. *Natural Language Processing Tools* [online]. [accessed Nov 11, 2016]. Available from: http://www.phontron.com/nlptools.php

[12] MANNING, Christopher, Mihai SURDEANU, John BAUER, Jenny FINKEL, Steven BETHARD, and David McCLOSKY. *The Stanford CoreNLP Natural Language Processing Toolkit* [online]. 2014 [accessed Nov 5, 2016]. Available from: http://nlp.stanford.edu/pubs/StanfordCoreNlp2014.pdf

[13] ARLOW, Jim and Ila NEUSTADT. *UML 2 and The Unified Process: Practical Object-Oriented Analysis and Design.* 2nd ed. Boston: Addison-Wesley, 2005. ISBN 978-0-321-32127-5.

[14] *About OMG* [online]. Object Management Group, 2015. [accessed: Nov 14, 2016] Available from: http://www.omg.org/gettingstarted/gettingstartedindex.htm

[15] ROLLAND, Colette and Christophe PROIX. *A natural language approach for Requirements Engineering* [online]. [accessed: Nov 5, 2016]. DOI: 10.1007/BFb0035136. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.1277&rep=rep1&type=pdf

[16] AMBRIOLA, Vincenzo and Vincenzo GERVASI. *Processing Natural Language Requirements* [online]. 1997 [accessed: Dec 28, 2016]. Available from: http://dl.acm.org/citation.cfm?id=786786

[17] KOF, Leonid. *An Application of Natural Language Processing to Domain Modelling: Two Case Studies* [online]. 2004 [accessed Dec 2, 2016]. Available from: https://pdfs.semanticscholar.org/ff28/af43b15a25bb59b2044c2e9bba9fa0008820.pdf

[18] OVERMYER, Scott, Lavoie BENOIT and Owen RAMBOW. *Conceptual Modeling through Linguistic Analysis Using LIDA* [online]. 2001 [accessed: Jan 2, 2017]. Available from: http://www.cogentex.com/papers/lida-paper-197-final.pdf

[19] ARELLAANO, Andres, Edward ZONTEK-CARNEY, and Mark A. AUSTIN. *Natural Language Processing of Textual Requirements* [online]. [accessed: Dec 28, 2016]. Available from: https://www.isr.umd.edu/ austin/reports.d/ICONS2015-AA-EC-MA.pdf

[20] LANDHÄUßER, Mathias, Sven J. KOERNER, and Walter TICHÝ. *From Requirements to UML Models and Back: How Automatic Processing of Text Can Support Requirements Engineering* [online]. 2014 [accessed: Dec 27, 2016]. Available from: http://link.springer.com/article/10.1007/s11219-013-9210-6

[21] *The University of Pennsylvania (Penn) Treebank Tag-set* [online]. Eric Atwell. [accessed: Dec 12, 2016]. Available from: http://www.comp.leeds.ac.uk/amalgam/tagsets/upenn.html

[22] SCHUSTER Sebastian, Christopher D. MANNING. *Enhanced English Universal Dependencies: An Improved Representation for Natural Language Understanding Tasks* [online]. 2016 [accessed: Jan 2, 2017]

[23] *Universal Dependencies v2* [online]. [accessed: Apr 14, 2017]. Available from: http://universaldependencies.org

[24] EBBERS, Hendrik. *Mastering JavaFX 8 Controls: Create Custom JavaFX Controls for Cross-Platform Applications*. New York: McGraw-Hill Education, 2014. ISBN 978-0-07-183378-3.

[25] DEA, Carl, Mark HECKLER, Gerrit GRUNWALD, Jose PEREDA and Sean M. Phillips. *JavaFX 8: Introduction by Example*. Second edition. New York: Apress, 2014. ISBN 978-1-4302-6460-6.

Appendix A

Acronyms

CASE Tools Computer-Aided Software Engineering Tools

CSS Cascading Style Sheets

GUI Graphical User Interface

HTML HyperText Markup Language

IEEE Institute of Electrical and Electronics Engineers

MDA Model Driven Architecture

MDD Model Driven Development

NLP Natural Language Processing

SDLC Software Development Life Cycle

UML Unified Modeling Language

XML Extensible Markup Language

XMI XML Metadata Interchange

Appendix **B**

Contents of DVD

- `README.txt` ..... the file with DVD contents description
- `Source Files` ..... the directory of source codes
  - `Intransitive Verbs Testing Tool` ... the directory of auxiliary tool source codes
  - `TEMOS` .. the directory of *TEMOS* source codes (*IntelliJ IDEA project*)
  - `Thesis` ..... the directory of LaTeX source codes of the thesis
    - `Figures` ..... the thesis figures directory
    - `*.tex` ..... the LaTeX source code files of the thesis
- `TEMOS` ..... the directory with the executable *TEMOS* tool
  - `Presentation Video.avi` ..... the video presenting *TEMOS* functionality
  - `README.txt` ..... the file with the launch instructions
  - `TEMOS.jar` ..... the executable *TEMOS* tool
  - `TEMOS.bat` .. the launch shortcut with the recommended JVM settings
- `Text` ..... the thesis text directory
  - `Master's Thesis.pdf` ..... the Master's thesis in a PDF format

Appendix C

System Requirements for TEMOS

Here are listed the *recommended system requirements* to run *TEMOS tool*:

- minimum recommended Java version: **1.8.0 (update 91)**,
- minimum recommended JVM (Java Virtual Machine) heap size: **2.5 GB**,
- minimum screen resolution: **800x600**.

C.1 Additional Information

Installation *TEMOS* is a portable *Java JAR application* – no installation is needed.

Launch You can run the *TEMOS.jar*, or you can use (if you are a Windows user) the enclosed *Run.bat* file that runs the *TEMOS* application with the minimum recommended heap size.

Internet connection *TEMOS* can run without an Internet connection. On the other hand, some features use on-line resources such as *dictionaries* and *ontological databases* and their launch requires an available Internet connection.

Application log An application log (*TEMOS.log* file) is located in the sub-folder called *logs*, and it is overwritten every time you run *TEMOS*.

Appendix D

Example of Generated XMI File

This *XMI* file was generated by *TEMOS* based on the example in subchapter 6.2.4.

Listing D.1: Example of Generated *XMI* File

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.1"
         xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
         xmlns:uml="http://schema.omg.org/spec/UML/2.1">
  <xmi:Documentation exporter="Textual Modeling System (TEMOS)"
                     exporterVersion="1.0" />
  <uml:Model xmi:type="uml:Model" name="Hotel_Model">
    <packagedElement xmi:type="uml:Package"
                     name="Hotel" xmi:id="0">
      <packagedElement xmi:type="uml:Class"
                       name="business group" xmi:id="1">
      </packagedElement>
      <packagedElement xmi:type="uml:Class"
                       name="hotel" xmi:id="2">
      </packagedElement>
      <packagedElement xmi:type="uml:Class"
                       name="space" xmi:id="3">
        <ownedAttribute xmi:type="uml:Property"
                        name="rent cost" visibility="private" />
        <ownedAttribute xmi:type="uml:Property"
                        name="area" visibility="private" />
      </packagedElement>
      <packagedElement xmi:type="uml:Class"
```

```
                name="hotel bedroom" xmi:id="6">
  <generalization xmi:type="uml:Generalization" general="3" />
  <ownedAttribute xmi:type="uml:Property"
                  name="room number" visibility="private" />
  <ownedAttribute xmi:type="uml:Property"
                  name="bed" visibility="private" />
</packagedElement>
<packagedElement xmi:type="uml:Class"
                name="conference room" xmi:id="7">
  <generalization xmi:type="uml:Generalization" general="3" />
  <ownedAttribute xmi:type="uml:Property"
                  name="maximum capacity" visibility="private" />
</packagedElement>
<packagedElement xmi:type="uml:Class"
                name="television" xmi:id="10">
  <ownedAttribute xmi:type="uml:Property"
                  name="serial number" visibility="private" />
  <ownedAttribute xmi:type="uml:Property"
                  name="display size" visibility="private" />
</packagedElement>
<packagedElement xmi:type="uml:Class"
                name="receptionist" xmi:id="11">
</packagedElement>
<packagedElement xmi:type="uml:Class"
                name="reservation" xmi:id="12">
  <ownedAttribute xmi:type="uml:Property"
                  name="start date" visibility="private" />
  <ownedAttribute xmi:type="uml:Property"
                  name="end date" visibility="private" />
</packagedElement>
<packagedElement xmi:type="uml:Class"
                name="booking" xmi:id="13">
  <ownedAttribute xmi:type="uml:Property" name="identifier"
                  visibility="private" />
</packagedElement>
<packagedElement xmi:type="uml:Class"
                name="customer" xmi:id="14">
  <ownedAttribute xmi:type="uml:Property"
                  name="surname" visibility="private" />
  <ownedAttribute xmi:type="uml:Property"
```

```
                    name="address" visibility="private" />
</packagedElement>
<packagedElement xmi:type="uml:Class"
                 name="name" xmi:id="15">
</packagedElement>
<packagedElement xmi:type="uml:Class"
                 name="meeting room" xmi:id="22">
  <ownedAttribute xmi:type="uml:Property"
                   name="projection screen" visibility="private" />
</packagedElement>
<packagedElement xmi:type="uml:Association"
                 name="own">
  <memberEnd xmi:idref="23" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="23">
    <type xmi:idref="1" />
  </ownedEnd>
  <memberEnd xmi:idref="24" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="24">
    <type xmi:idref="2" />
  </ownedEnd>
</packagedElement>
<packagedElement xmi:type="uml:Association"
                 name="offer">
  <memberEnd xmi:idref="25" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="25">
    <type xmi:idref="2" />
  </ownedEnd>
  <memberEnd xmi:idref="26" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="26">
    <type xmi:idref="3" />
  </ownedEnd>
</packagedElement>
<packagedElement xmi:type="uml:Association"
                 name="employ">
  <memberEnd xmi:idref="27" />
  <ownedEnd xmi:type="uml:Property"
```

```
            isOrdered="true" xmi:id="27">
    <type xmi:idref="2" />
  </ownedEnd>
  <memberEnd xmi:idref="28" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="28">
    <type xmi:idref="11" />
  </ownedEnd>
</packagedElement>
<packagedElement xmi:type="uml:Association"
                 name="contain">
  <memberEnd xmi:idref="29" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="29">
    <type xmi:idref="6" />
  </ownedEnd>
  <memberEnd xmi:idref="30" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="30">
    <type xmi:idref="10" />
  </ownedEnd>
</packagedElement>
<packagedElement xmi:type="uml:Association"
                 name="have">
  <memberEnd xmi:idref="31" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="31">
    <type xmi:idref="7" />
  </ownedEnd>
  <memberEnd xmi:idref="32" />
  <ownedEnd xmi:type="uml:Property"
           isOrdered="true" xmi:id="32">
    <type xmi:idref="15" />
  </ownedEnd>
</packagedElement>
<packagedElement xmi:type="uml:Association"
                 name="take care of">
  <memberEnd xmi:idref="33" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="33">
```

```
    <type xmi:idref="11" />
  </ownedEnd>
  <memberEnd xmi:idref="34" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="34">
    <type xmi:idref="12" />
  </ownedEnd>
</packagedElement>
<packagedElement xmi:type="uml:Association"
                 name="have">
  <memberEnd xmi:idref="35" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="35">
    <type xmi:idref="13" />
  </ownedEnd>
  <memberEnd xmi:idref="36" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="36">
    <type xmi:idref="14" />
  </ownedEnd>
</packagedElement>
<packagedElement xmi:type="uml:Association"
                 name="have">
  <memberEnd xmi:idref="37" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="37">
    <type xmi:idref="13" />
  </ownedEnd>
  <memberEnd xmi:idref="38" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="38">
    <type xmi:idref="3" />
  </ownedEnd>
</packagedElement>
<packagedElement xmi:type="uml:Association"
                 name="identify">
  <memberEnd xmi:idref="39" />
  <ownedEnd xmi:type="uml:Property"
            isOrdered="true" xmi:id="39">
    <type xmi:idref="14" />
```

```
        </ownedEnd>
        <memberEnd xmi:idref="40" />
        <ownedEnd xmi:type="uml:Property"
                  isOrdered="true" xmi:id="40">
          <type xmi:idref="15" />
        </ownedEnd>
      </packagedElement>
    </packagedElement>
  </uml:Model>
</xmi:XMI>
```

Appendix E

Intransitive Verbs Testing Tool

We created this tool to verify if the verb is *intransitive*. This tool expects to input a text file of words separated by a new line. We then test each word using the on-line dictionary *Wordnik*. Below is a screenshot of the result of *Intransitive Verbs Testing Tool*.

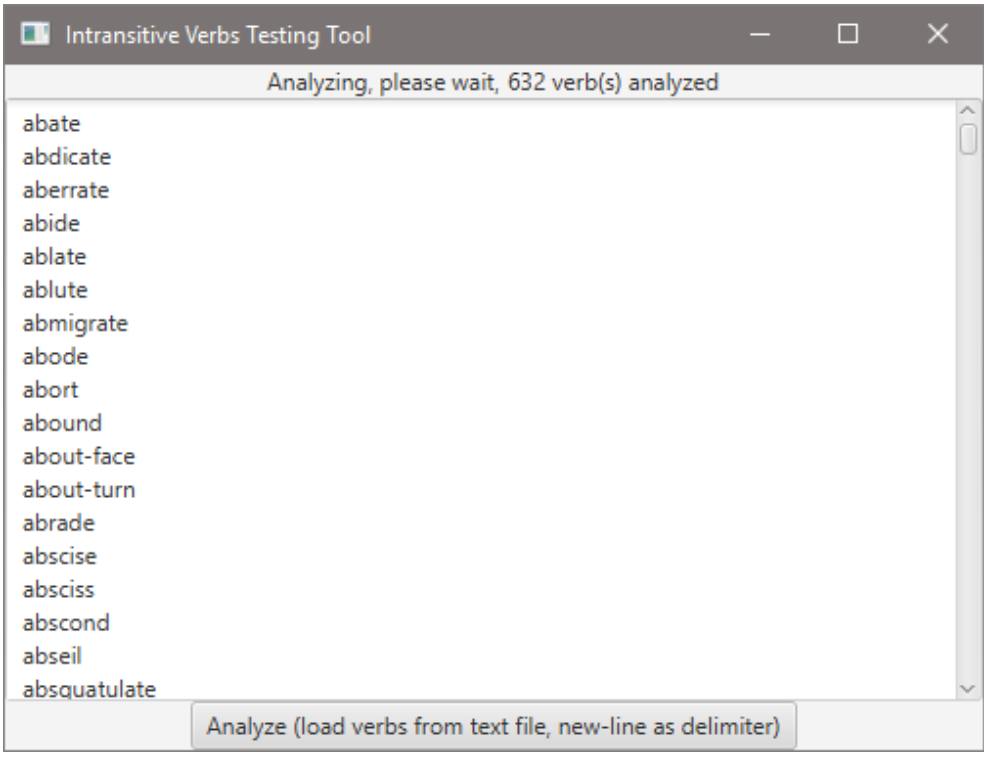


Figure E.1: Intransitive Verbs Testing Tool

You can find this tool included on the enclosed DVD.