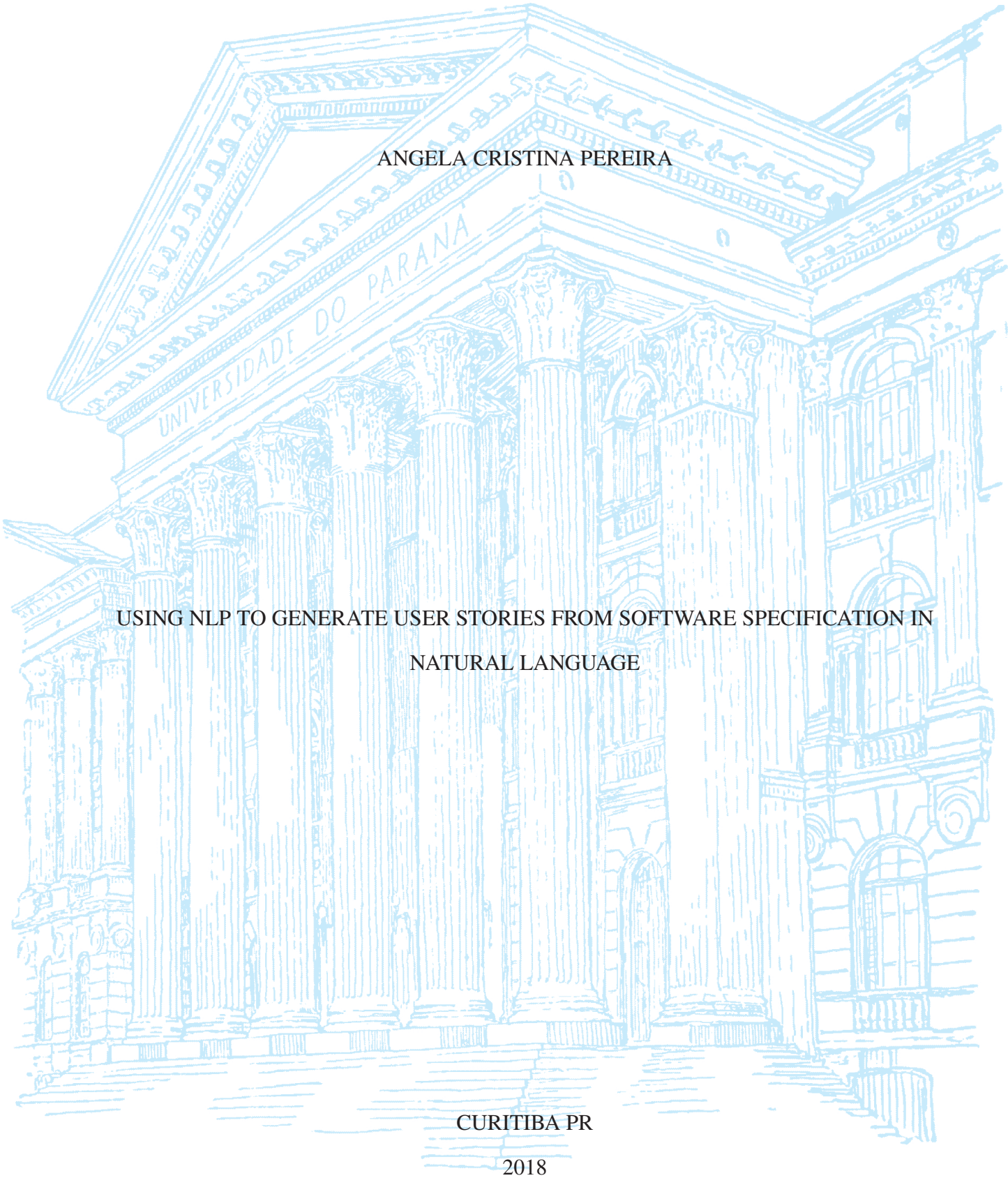
UNIVERSIDADE FEDERAL DO PARANÁ

ANGELA CRISTINA PEREIRA




# USING NLP TO GENERATE USER STORIES FROM SOFTWARE SPECIFICATION IN NATURAL LANGUAGE

CURITIBA PR

2018

ANGELA CRISTINA PEREIRA

# USING NLP TO GENERATE USER STORIES FROM SOFTWARE SPECIFICATION IN NATURAL LANGUAGE

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Andrey Ricardo Pimentel.

CURITIBA PR

2018

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

P436u Pereira, Angela Cristina
Using NLP to generate user stories from software specification in natural language / Angela Cristina Pereira. – Curitiba, 2018.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2018.

Orientador: Andrey Ricardo Pimentel.

1. Processamento de linguagem natural (Computação). 2. Software – Desenvolvimento. 3. Desenvolvimento ágil de software . I. Universidade Federal do Paraná. II. Pimentel, Andrey Ricardo. III. Título.

CDD: 006.35

Bibliotecária: Vanusa Maciel CRB- 9/1928

MINISTÉRIO DA EDUCAÇÃO
SETOR SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA

# TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da dissertação de Mestrado de **ANGELA CRISTINA PEREIRA** intitulada: **Using NLP To Generate User Stories From Software Specification in Natural Language**, após terem inquirido a aluna e realizado a avaliação do trabalho, são de parecer pela sua aprovação no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 27 de Agosto de 2018.

ANDREY RICARDO PIMENTEL
Presidente da Banca Examinadora (UFPR)

ADOLFO GUSTAVO SERRA SECA NETO
Avaliador Externo (UTFPR)

SILVIA REGINA VERGILIO
Avaliador Interno (UFPR)

*To Jocélia and Angelo, my parents, for their unconditional support.*
*To Anderson Sanji, my boyfriend, who always gave support and had a lot of patience with me during my MCA times.*
*To all my teachers who are my inspiration and always motivated me, from my excellent public school's teachers to my notable UFPR's teachers and my advisor Dr. Andrey Pimentel.*
*To all my friends Software Engineers that give me a lot of support to achieving this, and mainly to Xênia and Morgana.*

# Acknowledgements

To God who gave me the life and always will be my strength, my hope, my best example of love, the one who is always with me. I thank God for giving me the opportunity of realizing so many dreams, for helping me to break several obstacles to achieve my personal goals. And finally, I thank God for putting so many good people in my way, these people who inspire us, who always have a word for us even when everything goes wrong and we're hopeless and sad, these people who are like God's angels in our lives, because when I'm with these people, knowing that God is inside us, I can feel God's love.

To my advisor, professor Dr. Andrey Ricardo Pimentel, for the guidance, competence and dedication so important. During these times, you were my best example of what is being a teacher, because educate is more than teach the lessons, but it's also about to understand all the internal and external factors that influence in this process, and in that matter I think the society needs more masters like you. To arrive at this point was very difficult for me, because I kept working in the industry while doing my Master's degree, but you were definitely the person who better understood me in this situation, and always helped me to trust in myself when I arrived discouraged, you always had a few words of encouragement and motivation. Thanks for believing in me and the many compliments and incentives.

I give my special thanks to the professors, members of the examination boards: Prof. Dra. Silvia Regina Vergilio and Prof. Dr. Adolfo Gustavo Serra Seca Neto, who contributed in the evaluation of this work and gave me valuable suggestions for improvements.

I thank my professors from UFPR: Dra. Rafaela Mantovani Fontana and Dr. Lucas Ferrari, for motivating me to start the master's degree and give me recommendation letters. You are incredible professors.

I thank my colleagues Xênia k. Amorim and Morgana Borguezam, who helped me providing data mass for the evaluation of this work, and also the company who shared its data with me with secrecy terms. I also thank my managers and leaders that encouraged me during these times and also made my schedules more flexible. I thank my colleagues who had supported me during this time, the ones that helped to keep myself motivated, and again to Xênia and Morgana that read this work to help me identifying improvement opportunities.

To my love Anderson S. Sanji, for all the support that you always gave me, for your saint's patience with me during these difficult times. Thank you for staying by my side, even without the attention due and after so many moments of lost leisure. Thank you for every laugh that you made me do, for every sweet word of motivation, for your unconditional love even when I was so stressed, thank you for making me happy.

A special thanks to my parents, Jocélia and Angelo, for all the lessons of love, respect, discipline, for teaching me to fight to reach what I desire. Thank you both for supporting me in the way you could and for always encourage me to study and bring me books that increased my passion for knowledge. Thank you for always being there for me.

I'd like to take this opportunity to also thank all the teachers that passed through my life. You're all heroes! From my public school's teachers to my UFPR's teachers, you're all very important in my life. And especially in this moment of Brazil's history when the teachers are being persecuted and poorly treated by many citizens, I give to you all my support. To pass through all these barriers, hate, being so underpaid and keep fighting for education is something

that I really admire in you all. Nobody can do what you all do every day, no one loves so much the children, teenagers and adults to dedicate your lives for something so honorable like educating the society. I'd like to give my special thanks to all you teachers that always motivated me, lent your personal books or gave me the best reading suggestions, and for my Math teachers that went beyond and really impacted on my choice for the computer science world.

*"Dreams disdain fine lines and finishing touches on landscapes – they content themselves with thick but representative brushstrokes."(Machado de Assis)*

# RESUMO

O processo de elicitar as Histórias de Usuário requeridas para o desenvolvimento de *software* exige tempo e dedicação, e pode apresentar muito retrabalho se as conversas com partes interessadas não fornecerem informações coesas. O principal problema enfrentado é que o cliente muitas vezes não tem clareza sobre o que ele realmente quer, e no estado da arte não havia uma abordagem ou ferramenta que auxiliasse a transpor o que cliente deseja em histórias de usuário. Pensando nisso, propusemos a abordagem e ferramenta UserStoryGen para simplificar todo esse trabalho extenso, resolvendo esse problema através do uso de técnicas de Linguagem Natural (NLP) com estruturas adequadas para esse propósito e usando o modelo padrão de descrição de história para gerar automaticamente histórias de usuários. A abordagem UserStoryGen consiste em extrair informações como: título, descrição, verbo principal, usuários e entidades sistêmicas de histórias de usuários, a partir do texto não estruturado. O UserStoryGen usa o texto *big picture* como entrada para processamento de texto e geração automática de histórias do usuário. As histórias do usuário são geradas por meio de uma *Restful API* no formato *JSON* e podem ser exibidas tanto nesse formato, se apenas a chamada da Restful API for usada, como usando uma interface gráfica que mostrará o resultado em uma tabela. A implementação da UserStoryGen teve como objetivo automatizar este processo trabalhoso de extração de histórias de usuários do texto e obteve resultados significativos, principalmente nos testes com dados da indústria. Entre os três grupos de estudos de caso realizados, o terceiro, que utilizou dados da indústria, obteve os melhores resultados com textos que tiveram uma acurácia média de 76%, precisão de 88,23%, *recall* de 78,95% e medida F1 de 83,33%. O segundo grupo de estudos de casos com textos fornecidos por especialistas em Engenharia de Software obteve uma acurácia média de 73,68%, precisão de 85,71% e F1 de 82,76%. O primeiro grupo, utilizando textos de um*white paper* e de um livro teve o pior resultado, com uma acurácia média de 60% e uma medida F1 de 60,87%. Com base nos resultados obtidos com a UserStoryGen, concluímos que é completamente possível atingir o objetivo se pré-identificar e extrair as possíveis histórias de usuário para um determinado texto, e a implementação da abordagem proposta também pode ser melhorada em trabalhos futuros. A UserStoryGen representa um ganho para o Processo de Desenvolvimento Ágil, eliminando o tempo gasto na identificação de Estórias de Usuário, quando a equipe possui um texto com a *big picture* ou um documento textual das funcionalidades para usar como entrada.

**Palavras-chave:** Processamento de Linguagem Natural, Extração Automática, Histórias de Usuário, *Stanford CoreNLP*.

# ABSTRACT

The process of eliciting User Stories required for software development requires both time and dedication, and can present a lot of rework if conversations with stakeholders do not provide cohesive information. The main problem faced is that the client often lacks clarity about what he really wants, and in the state of the art there was no approach or tool that helps transpose what the customer wants into user stories. Thinking on it, we proposed the UserStoryGen approach and tool to simplify all this extensive work, resolving the issue through the use of Natural Language Processing (NLP) techniques with structures and the standard user story description template to automatically generate user stories. UserStoryGen's approach consists of extracting information such as: title, description, main verb, users and systemic entities of user stories from the unstructured text. The UserStoryGen uses *big picture* text as input for text processing and automated generation of the user stories. The user stories are generated through a Restful API in the *JSON* format and can be viewed either in this format, if only the Restful API call is used, as well as using a graphic interface that shows the results through a table. The implementation of UserStoryGen is aimed to automate the laborious process of extracting user stories from text and it obtained significant results, mainly with industry data. Among the three groups of case studies, the third one, that used industry data, obtained the best results with texts that had an average accuracy of 76%, precision of 88.23%, recall of 78.95% and F1 measure of 83.33%. The second group, using texts provided by software engineering specialists obtained an average accuracy of 73.68%, precision of 85.71% and F1 measure of 82.76%. The first group, using texts from a white paper and a book had the worst results with an average accuracy of 60% and a F1 measure of 60.87%. Based in the results obtained with the UserStoryGen, we concluded that it's completely possible to achieve the goal if pre-identifying and extracting the possible user stories for a given text, and the implementation of the proposed approach also can be improved in the future works. The UserStoryGen is a gain for Agile Development Process by eliminating time spent in User Stories identification when the team has a big picture text or a Features textual document to use as input.

**Keywords:** Natural Language Processing, Automatic Extraction, User Stories, Stanford CoreNLP.

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| DINF | Department of Informatics |
| PPGINF | Postgraduation Program in Informatics |
| UFPR | Federal University of Paraná |
| NLP | *Natural Language Processing* |
| NLTK | *Natural Language ToolKit* |
| NER | *Named Entity Recognition* |
| REST | Representational State Transfer |

# List of Symbols

| | |
|---|---|
| $\alpha$ | alpha, first letter of the Greek alphabet |
| $\beta$ | beta, second letter of the Greek alphabet |

# Contents

# 1 Introduction

The software development process is complex and extensive. The initial part of software development is normally defined throughout a specification or text containing the *big picture* that describes the desired software. We know that agile software development produces less documentation, but it doesn't mean that nothing is produced. After interviewing the end user and discussing about product strategies with the all the stakeholders, a minimum of documentation or text should be written by the Business/System Analysts inside the agile team.

According to some practitioners, like Rasmusson (2010), this initial document is called the **big picture**. For Rasmusson (2010), the **big picture** is the correct context for making decisions about the development of the software in question. This specification of the desired software is the basis for extracting the customer's desires in relation to the system's necessary functionalities, business rules and the type of data the system will manage, etc.

> "Teams face thousands of decisions and trade-offs every day. And without the right context or big-picture understanding, it's impossible for them to make the right trade-offs in an informed or balanced way". (Rasmusson, 2010)

The idea of using a **big picture** text is that a heavy documentation is not necessary to understand the main idea of a desired software. A short and concise text can help figure out what will be the features and user stories necessary to deliver the desired software.

Identifying user stories and their requirements is a crucial phase for agile software development. At this stage, after the requirements and **system features** are defined, it is necessary to divide the work into tasks, where each task is considered as the development of a **user story**. To allocate the work within a sprint or iteration in agile development, it's taken into consideration not only what is prioritized by the client, but also what really fits and can be performed within each cycle. For example, if an agile development team defines that each cycle of 2 weeks is a reasonable time to deliver value, then we say that the sprint takes 2 weeks.

In order to accelerate the costly work of the Analysis and Design stage in the construction of software, we are proposing an approach and a tool that simplifies the process of elicitation of user stories. The tool implementing our approach should identify the N user stories found in the **big picture**/software specification text as well as the actors/user types and system entities that the described software can contain.

Among the several works that tried to solve similar problems, none of them worked with texts in natural language without any type of standard for the automatic extraction of the specific case of User Stories. One of the two closest work found was the article from Ghosh et al. (2014), which attempted to make a transcription between semi-formal requirements for requirements in formal language. However the initial texts in semi-formal language already offered an ease in processing, in comparison with the proposed approach input text. The second closest work found was the master thesis from Rane (2017), and it dealt with the automatic generation of Test Cases from previously informed user stories.

In reference to the classification using natural language processing and deep convolutional neural networks, there is the work from dos Santos e Gatti (2014), which aimed to extract feelings from Twitter phrases, in which the authors even created their own neural network. Although similar in the usage of natural language processing usage, these studies don't offer anything related to the automatic generation of software development artifacts.

## 1.1 Problem Statement

A large documentation can say many things, but if it's not accurate enough and the features are described using only these excessive details, mistakes can occur and time can be lost by the development team. As stated by one of the agile principles from Agile Manifest: " the most efficient and effective method of conveying information to and within a development team is face-to-face conversation".

In most cases, the system specification texts provided by the customers or stakeholders do not follow any type of formal language, what makes user stories extraction hard even for humans. In addition, the texts usually contain problems such as: ambiguity, very long sentences, confusing expressions and other difficulties.

The process of features and user stories elicitation necessary to create the software desired by the product owner is normally costly and time consuming. According to this author's work experience, in the agile development process, using SAFe [1] methodology, a group compound by Business Analysts, Product Owners and the users are involved to decide what features will be implemented. This process of user stories elicitation usually takes more then a week. After the initial planning, the whole team is put together to refine and put in a paper the user stories with their respective descriptions, during a period which is called "Planning Iteration".

Depending of the Agile methodology used and also if the software engineering area is inside the company that needs the software or a consultant company, it's common to have not only face-to-face conversations about the desired software, but some documents produced.

When it's a consultant company that will deliver the given software there's a formal document, to regulate the obligations in the services provided, in this case the software that will be delivered. The elicitation of User Stories should be done carefully, since the first documents such as the specification of the software that the customer provides are not exactly the clear picture of the desired features, and this contributes for reworking and an increase of costs during the software development. As tools that allow a previous evaluation of how the user stories are rarely used during software development, errors in the phase of defining the project scope, specification of the software aren't quickly perceived and corrected.

Another problem that arises from the poor elicitation of user stories is the poor complexity score of the stories, which directly affects the time required for development team to finish a group of user stories that forms a system feature. When a user story doesn't contain an accurate description, it causes the team to give the wrong story points measure, because the description can be misunderstood and the team can punctuate either lower or higher than it should be.

The use of a User Stories auto-extraction tool could make all this work easier. The development of a tool like this can be a difficult task of natural language processing, given the lack of standardization factors in text writing. Although it is a complex task, we are proposing an approach and also a tool implementing it to computationally solve the issue with the use of

[1] SAFe and Scaled Agile Framework are registered trademarks of Scaled Agile Inc. Reproduced with permission from © 2011-2017 Scaled Agile, Inc. All rights reserved.

Natural Language Processing techniques and the creation of processing phases specifically for this purpose.

## 1.2 General Objectives

The main objective is to create and implement an approach to automatically extract user stories from unstructured text that contains information about a desired software, such as a short software description or text containing the big picture of the desired software. Each user story extracted should contain the title, description, the user or actor of the user story, the system entity found and the main action of the given user story.

### 1.2.1 Specific Objectives

- Create an approach that solves the issue of automatic extraction of user stories from unstructured text and implement it.
- Investigate how to identify user or actor mentions in the text and their references with other phrases for replacing pronouns with the user references.
- Develop a text processing structure that allows to accurately recognize the main verbs of each multiple verbs sentence.
- Implement an efficient method for splitting sentences with multiple verbs.
- Create a method to identify and extract user stories through both active and passive voice sentences.
- Produce a list of user stories with the attributes: user, title, description, system entity and main verb.
- Plan and execute case studies to evaluate the proposed approach and tool.

## 1.3 Contribution

The development of a tool that implements this approach of User Stories automatic extraction, specifying information such as the title and description of the story, who is the user, the systemic entity and main verb will benefit the process of building software (specifically the Analysis part) and will contribute to the area of Software Engineering.

The benefits of the proposed tool will enhance what has already been developed in other studies that presented tools with the use of *NLP*, with the objective being automatic generation of artifacts for the area of Software Engineering. As the related studies found didn't present a solution for user stories extraction, this work will fill that lacuna and will be valuable for generating user stories as a way to increase productivity in preparing and identifying user stories.

During the development of the proposed tool, several natural language processing issues had to be solved, such as problems of ambiguity, duplicated information and expression failures, different sentence structures and tenses and voices. These issues made the work very challenging since minimizing possible failures required a combination of Natural Language Processing techniques and creating processing steps.

We created phases such as simplification of sentences, creation of structures for elimination of **stop words**, and creation of regular expressions' own structures to generate titles and descriptions of the user stories which correlates with information present in the dependency tree of the sentences.

## 1.4 Document Organization

This work is divided in five chapters. The following chapter present the concepts that ground the development of this work. The third chapter describes the works and reference sources used. The fourth chapter explains how the software that implements the proposed approach was developed, from its design to the implementation. Chapter five presents the verification of the work, the experiments done and their results. The last chapter contains the conclusion of this work. And the last division is the bibliographic references.

# 2 Concepts

## 2.1 Agile Software Development

According to Pressman (2000) “the classic life cycle or the waterfall model, [...] suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and support”. Therefore, we can conclude that traditional models such as the Waterfall show the software development activities partitioned and running in a linear sequence, which usually brings the old situation of problems with maintenance and the user or owner of the product changing his mind afterwards of a complete development cycle that can last for months and cost a lot of money.

> In 2001, the Agile Manifesto was written by the practitioners who proposed many of the agile development methods, [...] the manifesto states that agile development should focus on four core values:

- Individuals and interactions over processes and tools;
- Working software over comprehensive documentation;
- Customer collaboration over contract negotiation;
- Responding to change over following a plan. (Dybå e Dingsøyr, 2008)

In the agile development there is a scheduled period to deliver something, this period is called Iteration or Sprint. Every iteration the team will work on certain tasks to deliver something valuable. Within each iteration the team will perform the analysis, development and testing for each of the tasks allocated or prioritized for the given iteration. In order to transpose agile analysis that cares for accuracy, agile software development uses as many user stories as is necessary to succinctly describe client requirements for small tasks that are easy to manage and deliver within iterations. User stories are one of the main elements of agile development, through which the tasks that are originated from the customer’s requirements are expressed, and through them customer value deliveries are managed within each iteration.

According to Rasmusson (2010), it is not necessary to have everything written in a documentation stack, because “you are going to need a way of doing analysis that is light, that is accurate, and that gives exactly what you need, just when you need it”. Secondly, Rasmusson (2010) says that development should be a solid rock, because all you do not want in an agile team is having to go back and fix buggy code. Therefore, Rasmusson (2010), stresses that “code well-designed, well-tested and completely integrated code as you go”. Thirdly, Rasmusson (2010) points out that “your testing will have to be lockstep with development, because you can’t afford to wait until the end of the project to see whether everything works, you need to maintain the health and integrity of the system from day one of the project”.

There are several common practical forms within the agile development. Among them it is possible to highlight the practice of pair programming, which is the programming in pairs of

developers to enable collaboration between team members in building the code and promoting a continuous improvement in the use of best practices. Continuous integration is another practice, which is tied to the use of automated build and deployment tools, as well as automatic Sonar update and automated test execution tracking through plans within the CI (Continuous Integration) tool. All of these automated processes through continuous integration tools promote easy health check of the software project with the generation of code coverage reports and numbers on the automated tests.

## 2.2 *Features* in agile development

According to Leffingwell e Widrig (2003), "when interviewed about their needs and requirements for a new system, stakeholders typically do not describe any of these things, at least not in terms of the definitions we have so far", and these features definitions are normally expressed by the stakeholders in an ambiguous way, as if describing a problem and not how they wish to solve it within the system, through something that is not so abstract. For Leffingwell e Widrig (2003), we even call high-level expressions on the desired behavior for the system *features* of a system or product.

Leffingwell e Widrig (2003) states that we define a feature as a service that the system comes from to fulfill one or more stakeholder needs. According to Leffingwell e Widrig (2003), to help manage feature information, "we have introduced the construction of feature attributes, or data elements that provide additional information about the feature, and these attributes are used to create a relation of feature information or requirements data with other types of project information". Subsequently the attributes of the features can be useful to prioritize and to manage the state of the features proposed for implementation.

For Leffingwell (2017), from Scale Agile Framework (SAFe ©), "the hierarchy of artifacts that describes the system's functional behavior is: Epic, Capabilities, Features and User Stories" [1], in this order. In agile support and project management tools such as *Rally*, Features are the first level of direct management of the tasks that are prioritized within the iterations, and within each feature we have the stories that are part of the feature in question, that is, the user stories that complete the development of that desired behavior for the system. Therefore, any and all user story created must be linked to a feature.

## 2.3 User stories

Use Cases are extremely important to the Unified Process as well as user stories are important to the Agile process. However, user stories are not the same as use cases. Use Cases describe a complete iteration between the user and the system, including the N features present in the use case and exception paths, user stories treat each feature separately, and this is just one of many differences between stories user and use cases.

About user stories Kamthan (2015) states that "there must be an actual person behind a (user) role in a given user story, [...] and the value provided by a user story to its (user) role must be explicit". For Cohn (2004), user stories are composed of three aspects: a written description of the story used for planning as a reminder, talks about the story that serve to refresh the details of the user story and tests that transmit and document details and that can be used to determine when a user story is concluded.

[1] SAFe and Scaled Agile Framework are registered trademarks of Scaled Agile Inc. Reproduced with permission from © 2011-2017 Scaled Agile, Inc. All rights reserved.

In their work, Patel e Ramachandran (2009) proposed “a new ‘INSERT’ model to improve the quality of user story and to address customer requirements properly and on verifiable way the acronyms INSERT is as”:

- I: Independent,
- N: Negotiable
- S: Small enough to fit into iteration
- E: Estimatable or easy to Estimate
- R: Representation of user functionality (Requirement)
- T: Testable

The first user story template proposed by Connextra in 2001 and popularised by Cohn (2004) is: “As a (type of user), I want (goal), [so that (reason/benefit)]”.

For Rasmusson (2010), the template “As a ... I want to ... so that ...” is widely used in industry as a standard description of user stories and is cited in several sites, books and articles on the agile process. The core thing about this writing pattern for describing user stories is linked to the organization of information, and completeness in the understanding of the main issues in a user story: who wants it? What do you want? And for what? When we define the user, we are already defining an extremely important rule, since each software contains specific rules about the permissions of each type of user, and in some cases not all users can have access to a feature. By specifying who can access a feature and what this functionality will be, we can clarify what the purpose of the user story is. Next, we use the expression “so that” (for that) to explain the motivation for user story.

According to Rasmusson (2010), “what was really important last week can suddenly become not so important this week [. . . ] if all of our stories are intertwined and dependent on one another, making trade-offs becomes hard”. Therefore, user stories take care of one feature at a time, avoiding exception cases, which are usually handled in separate user stories in order to make them independent and easy to manage.

## 2.4 Natural Language Processing

Natural Language can be understood as the natural language that we use in everyday life to communicate, the language that is used in texts published in different social networks, books, magazines, that is, language of communication between humans, independently from the language, English, Portuguese or others. Therefore, the natural language does not follow a common standardization for all phrases.

The difference between natural language and other types of languages for Bird et al. (2009) is: “in contrast to artificial languages such as programming languages and mathematical notations, natural languages have evolved as they pass from generation to generation and are difficult to define with explicit rules”.

Natural language processing is a sub area within the area of Artificial Intelligence that focuses on the application of processing tasks for the treatment of natural language texts, information extraction and several other objectives that can extract value or information from texts in an automated way. To assist with NLP tasks, there are some common libraries that provide texts, writings, and other artifacts that we can use as base texts to validate ideas, to study

natural language processing, and more. One of them is the *corpora*, a collection of writings, which contains bodies of texts like medical journals and presidential speeches, and the other is *lexicon*, a library that contains words and their meanings, both of which return only texts in English.

During the processing of natural language, there are some commonly used tasks and techniques that receive their own nomenclatures, which we will discuss in the next subsections.

### 2.4.1 Tokenizing and Stop Words

When we begin text processing, usually the first task we perform is Tokenization. **Tokenizing** refers to the task of separating text in tokens. According to Harrison (2015b), the best definition for the term **tokenizing** is the separation of sentences and words in the body of a text. Initially, each sentence is separated, that is, for each sentence, a set compounded of words and punctuation is created, then with the token of each sentence, it separates the words within the sentence, generating tokens that can store a word or a punctuation mark.

In summary, with the task of **tokenization** it is possible to separate the text by paragraphs, sentences and words contained in a sentence. The separation by sentence is done by what it's called **sentence tokenizer** and by word through a **word tokenizer**.

Another useful technique for natural language processing is the use of **stop words**. Stop words are the words that are filtered out before or after the text processing. They're words that normally do not represent something meaningful and therefore can be removed from the text without any loss in the sense of the sentence.

### 2.4.2 Stemming and Lemmatizing

*Stemming*, which means to derive, is a data preprocessing technique for natural language processing. This technique allows to see the original value of a word, the root of the word, for example: a word that is in the gerund can be converted into its normal form. If we pass the word "writing" to a function that does *stemming* using any NLP library, the result will be the word "write", which is the root of the initial word.

According to Christopher D. Manning (2009), *Stemming* generally refers to a brute heuristic process that cuts the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes.

Since the *stemming* task retains the original meaning of a word, even though there are different or derived words to qualify something in common, after using stemming it is possible to compare if two sentences have the same meaning, but using different words, and this is quite useful to automatically evaluate if one work or article is a plagiarism of another, for example.

A very similar task to *stemming* is *lemmatizing*. According to Harrison (2015a), "the major difference between these is, as you saw earlier, stemming can often create non-existent words, whereas lemmas are actual words". *Lemmatizing* is a technique that searches the root word from a word "X", but always taking into account the dictionary and therefore we can verify that it is safer to use the *lemma* of a word than a word that passed through the *stemming* process.

According to Christopher D. Manning (2009), *lemmatizing* generally refers to making the processing using "vocabulary and morphological analysis of words, usually with the purpose of removing only inflectional endings and returning the base or dictionary form of a word, which is known as *lemma*". According to Christopher D. Manning (2009), if we compare the results of applying the *stemming* technique against the *token* or word "*saw*", the *stemming* could only

return “s” while lemmatizing technique would try to return *see* or *saw* depending on whether the use of the token was a verb or a noun.

### 2.4.3 *Part of Speech Tagging*

*Part of Speech Tagging* (POS Tagging), which is the action of labeling or putting a *tag* in a part of speech, is a technique that does exactly what it says it does, that is, it extracts a tag representing the syntactic function of each word in a sentence. This means that POS tagging is a task that labels words within a sentence, classifying them as: subject, noun, pronoun, verb, direct object, etc.

In the world of NLP, there is a table that defines tags to represent the syntactic function of words, which is very useful for cases where it is desired to refine sentences and use a new syntactic standard in the construction of them, with the information of part of speech tag it is possible to use NLP functions to normalize sentences.

The 2.1 Table is *Universal Part-of-Speech Tagset*, or a universal set of definitions about Part-of-Speech Tags.

Table 2.1: Universal Part-of-Speech Tagset

| Tag | Meaning | English Examples |
|---|---|---|
| ADJ | adjective | new, good, high, special, big, local |
| ADP | adposition | on, of, at, with, by, into, under |
| ADV | adverb | really, already, still, early, now |
| CONJ | conjunction | and, or, but, if, while, although |
| DET | determiner, article | the, a, some, most, every, no, which |
| NOUN | noun | year, home, costs, time, Africa |
| NUM | numeral | twenty-four, fourth, 1991, 14:24 |
| PRT | particle | at, on, out, over per, that, up, with |
| PRON | pronoun | he, their, her, its, my, I, us |
| VERB | verb | is, say, told, given, playing, would |
| . | punctuation | marks . , ; ! |
| X | other | ersatz, esprit, dunno, gr8, university |

For each NLP library there are tables with different tags, in addition to those in the Universal Part-of-Speech Tagset, table 2.1, and which aim to bring more information about syntactic function of each word and thus improve the accuracy in natural language processing. The table 2.2 table displays the POS tag list of the *NLTK* library.

To illustrate how it works, we’ll consider the three first paragraphs from a George W. Bush presidential speech text, which was cataloged as “2006-GWBush.txt” inside the NLTK library dataset:

> “PRESIDENT GEORGE W. BUSH’S ADDRESS BEFORE A JOINT SESSION OF THE CONGRESS ON THE STATE OF THE UNION
>
> January 31, 2006
>
> THE PRESIDENT: Thank you all. Mr. Speaker, Vice President Cheney, members of Congress, members of the Supreme Court and diplomatic corps, distinguished guests, and fellow citizens: Today our nation lost a beloved, graceful, courageous woman who called America to its founding ideals and carried on a noble dream.”

Table 2.2: *Part of Speech Tag* list of the NLTK library

| Tag | Meaning | English Examples |
|---|---|---|
| CC | coordinating conjunction | |
| CD | cardinal digit | |
| DT | determiner | |
| EX | existential there | "there is" ... think of it like "there exists" |
| FW | foreign word | |
| IN | preposition/subordinating conjunction | |
| JJ | adjective | ’big’ |
| JJR | adjective, comparative | ’bigger’ |
| JJS | adjective, superlative | ’biggest’ |
| LS | list marker | 1) |
| MD | modal | could, will |
| NN | noun, singular | ’desk’ |
| NNS | noun plural | ’desks’ |
| NNP | proper noun, singular | ’Harrison’ |
| NNPS | proper noun, plural | ’Americans’ |
| PDT | predeterminer | ’all the kids’ |
| POS | possessive ending | parent’s |
| PRP | personal pronoun | I, he, she |
| PRP$ | possessive pronoun | my, his, hers |
| RB | adverb | very, silently |
| RBR | adverb, comparative | better |
| RBS | adverb, superlative | best |
| RP | particle | give up |
| TO | to | go ’to’ the store. |
| UH | interjection | errrrrrrrm |
| VB | verb, base form | take |
| VBD | verb, past tense | took |
| VBG | verb, gerund/present participle | taking |
| VBN | verb, past participle | taken |
| VBP | verb, sing. present, non-3d | take |
| VBZ | verb, 3rd person sing. present | takes |
| WDT | wh-determiner | which |
| WP | wh-pronoun | who, what |
| WP$ | possessive wh-pronoun | whose |
| WRB | wh-abverb | where, when |

Below we can see the result of applying the POS tagging task over the mentioned presidential speech, through the figure 2.1:


Figure 2.1: Execution result of the code that does *POS tagging* using NLTK library.

As we can see, the code used a piece of presidential speech to make use of the Part-of-Speech tagging, with the NLTK library. In this case, words like "PRESIDENT", "GEORGE", "THE", "CONGRESS" and others were labeled "*NNP*" (proper noun, singular). The words "diplomatic", "graceful", "courageous", for example, were labeled "*JJ*" (adjective). Words like "lost" and "called" received the tag "*VBD*" (verb, past tense), dates like "2016" an "31", after citing the month "January", were labeled "*CD*" (cardinal digit), and so on each word in the text it received a tag and the punctuation marks received tags identical to them.

There are several ways of extracting part of speech tagging, another way besides using the NLTK library is through the usage of the Stanford library. The Stanford library can be used directly and also has an interface for connection between itself and NLTK. The Stanford NLP POS feature is the only one that has 97% accuracy and can be used as shown in the code snippet below.

```
1 from nltk.tag import StanfordNERTagger
2
3 st = StanfordNERTagger('english.all.3class.distsim.crf.ser.gz')
4 st.tag('Rami Eid is studying at Stony Brook University in NY'.split())
5
6 #Output
7 #[('Rami', 'PERSON'), ('Eid', 'PERSON'), ('is', 'O'), ('studying', 'O'),
8 #('at', 'O'), ('Stony', 'ORGANIZATION'), ('Brook', 'ORGANIZATION'),
9 #('University', 'ORGANIZATION'), ('in', 'O'), ('NY', 'LOCATION')]
```

As you can see, the Stanford NLP POS tagging library has tags that are different from the ones found in the NLTK list and also in relation to the universal POS list tags, for example, in the definition of In the case of the word "Rami", the Stanford library classifies as *PERSON*, but NLTK and the universal POS tag would classify as "*proper noun, singular*". In the case of the words "*Stony Brook University*" that would be classified as "*NNP*" by the universal list and by NLTK, they have a smarter tag when using Stanford, after all, besides being a singular name these words describe an organization as it says the tag of Stanford, and in the case of "*NY*" that knowingly refers to the city of New York, we also can see a better result with Stanford classifying it as "*LOCATION*", while the other libraries and lists classify it as "*NNP*".

### 2.4.4 Chunking and Chinking

After extracting the part of speech tag from each word we can perform other tasks based on the tags, one of them is called chunking, which consists of extracting words with certain tags from within a sentence. Applying the chunking technique, it is possible to group several words that have the same tags defined by the regular expression that will make the chunking.

The regular expression that is responsible for defining the purpose of the chunking task is composed of the tags that are to be extracted and also the known modifiers, which are common characters for any type of regular expression. The asterisk (*) character, for example, has the meaning of zero or more for a certain condition prior to it, and the plus sign (+) means that at least one or more elements corresponding to the preceding expression is required.

The code snippet below shows an example of how to perform the chunking task, using the NLTK library. For this example, a presidential speech from the "2005-GWBush.txt" file was used, and it is possible to identify the regular expression that makes chunking the variable *chunkGram*. The regular expressions can start with "r" and triple quotation marks are usually used. The word "*Chunk*" is there to identify the group of words that contains the sequence of required tags, and inside the key symbol the rule that defines what will be selected, in this case: any adverb that can be RB (adverb) or RBR (adverb, comparative), followed by zero or any VB or VBR (verb at any time), followed by zero or more NNPs (own nouns), followed by at least one or more than one NN (singular noun).

```
1  custom_sent_tokenizer = PunktSentenceTokenizer(train_text)
2  tokenized = custom_sent_tokenizer.tokenize(sample_text)
3
4  for i in tokenized[:1]:
5        words = nltk.word_tokenize(i)
6        tagged = nltk.pos_tag(words)
7        # expression to apply the chunk
8        chunkGram = r"""Chunk: {<RB.?>*<VB.?>*<NNP>+<NN>?}"""
9        chunkParser = nltk.RegexpParser(chunkGram)
10       chunked = chunkParser.parse(tagged)
11
12    for subtree in chunked.subtrees(filter=lambda t: t.label() == 'Chunk'):
13          print(subtree)
```

The code above, which uses the NLTK library in Python, contains three of the NLP techniques already presented. In line 5 the code shows the use of *word tokenizing* to separate each word of the sentence, creating then a list called "words" that in the sequence, in line 6, it becomes the parameter to the function that does the part of speech tagging, that is to generate the tags for each word. In line 8, it's defined the "chunkGram" variable that contains the regular expression for the chunking task. In line 10 all the words of the sentence go through the function that evaluates whether they obey or not the rule requested by the chunking expression, and in the positive case that word sequence of the sentence will receive the label "Chunk". In line 12, the code checks whether or not each statement has label Chunk, if yes, the sentence is printed, otherwise nothing happens.

When executed, the above code will bring the result expressed by the figure below 2.2:

If the need is not to group words in a sentence that follow a rule of syntactic composition, but rather to filter part of a sentence by eliminating certain words that have unwanted part-of-speech tag, then we can use another technique called **chinking**.

With the chinking task you can define how you want to filter a sentence with a regular expression. It's possible to indicate what you want to remove from each sentence and then to

```
(Chunk PRESIDENT/NNP GEORGE/NNP W./NNP BUSH/NNP)
(Chunk ADDRESS/NNP)
(Chunk A/NNP JOINT/NNP SESSION/NNP)
(Chunk THE/NNP CONGRESS/NNP ON/NNP THE/NNP STATE/NNP)
(Chunk THE/NNP UNION/NNP January/NNP)
(Chunk THE/NNP PRESIDENT/NNP)
(Chunk Thank/NNP)
```

Figure 2.2: Result of the NLTK library code example about chunking

process a text to get sentences with words that have part of speech tag other than those specified in the chink. Therefore, it can be said that the chinking technique is the opposite of chunking.

Basically, what defines if the task to be performed is chunking or chinking is the regular expression used in the text processing. In the previous code example, we have a regular expression that defines what we want to find based on the part of speech tags, but if we want to use the same code by changing the regular expression to define what we do not want to see in a sentence, that is already enough to say that we will perform a chinking task, since the correct modifiers will be used.

To define a regular expression that performs chinking, we first say that we want to select everything, for example, and then we begin to report the exclusions based on the part-of-speech tag of the words we don't need. In the regular expression below, we see an example of chinking in which we first select words with any POS tag and then inside the inverted curly bracket symbol "}{" we exclude any verb (VB) and any preposition (IN), any determiner (DT) and any locution TO.

```
for i in tokenized[5:]:
      words = nltk.word_tokenize(i)
      tagged = nltk.pos_tag(words)

   chinkGram = r"""Chink: {<.*>+}
               }<VB.?|IN|DT|TO>+{"""
```

After creating the regular expression of chinking, the execution of the task follows in the same way that we coded to perform chunking, as we can see below:

```
chinckParser = nltk.RegexpParser(chinkGram)
chincked = chinckParser.parse(tagged)

print(tagged)
for subtree in chunked.subtrees(filter=lambda t: t.label() == 'Chink'):
      print(subtree)
```

### 2.4.5 Named Entity Recognition (NER)

According to Bird et al. (2009), "named entities are definite noun phrases that refer to specific types of individuals, such as organizations, persons, dates, and so on". When we're working with natural language processing often it's necessary to get more information than the POS tags brings, and the NER tags are very useful in this sense.

The goal of the NER task or a NER system is to recognize the named entities mentions inside a give text, and provide specific tags that help in the natural language processing. If we have a text with a specific subject and inside this text we have a mention of an actor, with the POS tag the actor name will be labelled as NNP, which is good but when we apply the NER task we will be able to see that the actor also gained the tag PERSON.

Table 2.3: List of commonly used NE types according Bird et al. (2009)

| NE type | Examples |
|---|---|
| ORGANIZATION | Georgia-Pacific Corp., |
| WHO/ PERSON | Eddy Bonte, President Obama |
| LOCATION | Murray River, Mount Everest |
| DATE | June, 2008-06-29 |
| TIME | two fifty a m, 1:30 p.m. |
| MONEY | 175 million Canadian Dollars, |
| GBP | 10.40 |
| PERCENT | twenty pct, 18.75% |
| FACILITY | Washington Monument, Stonehenge |
| GPE | South East Asia, Midlothian |

The table 2.3 contains a common list of named entities tags and their examples, according Bird et al. (2009). The usage of the NER tags shown in the table 2.3 has a positive impact in cases of ambiguity that can occur in sentence construction in any language, as the algorithm will be able to count with an extra validation considering the NER information.

The following figure is a result of applying the Named Entity Recognition task through the site "http://corenlp.run/", which allows the execution of NLP tasks through the *Stanford CoreNLP* library, in this case, more precisely, the *Stanford NER* was used. According to Manning et al. (2014), Stanford CoreNLP is a *framework* with a pipe or sequence of Java (or at least JVM-based) annotation processes, which provides most of the common steps for natural language processing, from tokenization and naming entity recognition to coreference resolution.


Figure 2.3: Example of applying NER task in a sentence through Sanford Core NLP library

## 2.5 Conclusion

This chapter described the concepts of Agile Software Development, features and user stories definitions, as well as the natural language processing techniques that were used in this dissertation. The agile development of software is premised on the delivery of something of value each week or iteration to produce parts of software with good design, code quality, that was tested and also provides value to the customer.

Within the agile world, the tasks that developers should work with are always tied to features. Features are the definitions of a desired behavior for the system, which are implemented through the artifacts called user stories. User stories are smaller and manageable chunks of requirements or customer needs. Each user story has a description, acceptance criteria, and tests that tell you when the story can be considered complete or delivered.

Any text, wrote in any idiom, can be considered as natural language, whether it comes from a book or even a post on Twitter. Any form of human communication can be called natural language, and in the way it's, the natural language has the power to express ideas, opinions, describe facts or even requirements for a desired product or software. Therefore, natural language processing can bring numerous benefits in automating the extraction of relevant information or even processing of non formatted texts.

In this chapter, we presented some Natural Language Processing techniques. We described the **Tokenizing** technique, which separates text into sentences and words, the **Stop Words** technique, which deals with the definition of a list of words that can be removed from the text without loss. We showed the difference between the **Stemming** technique that extracts the root word from a word in any verbal and **Lemmatizing** technique that does the same but never generates a word that does not exist in the dictionary. Other techniques such as **Part-of-Speech Tagging** (POS Tagging), which is responsible for labeling words with tags referring to their syntactic function in the sentence or phrase. It was also seen that with the use of **Chunking** it is possible to group several words that have certain POS tags, unlike the **Chinking** technique that only removes words with certain POS tags and maintains the remainder of the phrase.

# 3 State of the Art

There are some published works on analysis and text processing involving the use of Natural Language Processing with the objective of automatic generation of text obeying a certain standardization.

While some existing researches are dedicated to making text processing to generate structures of standardized texts, some works deal with the application of techniques of Analysis of Feelings to classify texts, and among all analyzed few had as objective to use the classification or processing of texts within the area of Software Engineering and especially in what concerns the generation of artifacts for the agile process of software development.

In the course of this chapter we will present studies closest to the proposed approach and tool, which is to deal with the automatic generation of user stories. To discuss the related works, we divided the studies into three topics.

## 3.1 Search strategy and topics division

The search included both electronic databases and hand searches of annual periodicals, articles, theses and conference proceedings. The following electronic databases were searched:

- IEEE Xplore
- ACM Digital Library
- ISI Web of Science
- SpringerLink (Springer)
- ScienceDirect (Elsevier)

The search string started with: **("automatic generation" AND ("user stories" OR "user story"))** and it brought about of 637 results, but, unfortunately the precision of these results was not good. This search string was changed to **("automatic generation" AND "software engineering" AND ("user stories" OR "user story"))** and with that we reduced the results to around 448 items.

A second search string was created using a specific obligatory criteria of containing "Natural Language Processing" keyword, and the result was: **("automatic generation" AND "software engineering" AND "Natural Language Processing" AND ("user stories" OR "user story"))** and approximately 59 results were found.

We used the following exclusion criteria for the works found:

- Works that don't mention exact information about how the results were obtained or how the method works.

- Works summarizing results of others authors.
- Works with no relation of any automatic generation or classification related with textual information.

To exemplify the exclusions criteria we can mention one of the results obtained with the second search string was the article from Suri et al. (2015). The work Suri et al. (2015) stated to be about "a system for generating agile user stories that includes a processor configured for collecting a plurality of requirements, creating a plurality of content space specification files that includes the plurality of requirements, processing the plurality of content space specification files to generate the user stories". The authors of this article didn't mention how the texts or files are processed or how accuracy was measured, only that one of the key steps is to process "content space" specification files. As no evidence was shown about how this software works, and how are the inputs, it's really hard to compare the results the authors stated to have had.

Among the results found with both search strings we noticed that they can be divided into three topics:

1. Automatic generation or information classification of something valuable for any area but Software Engineering using Natural Language Processing techniques
2. Automatic generation of something valuable for the Software Engineering area using Natural Language Processing techniques, but which is not related with agile artifacts or user stories
3. Automatic generation or classification of agile software development related elements such as user stories or test cases, for example.

In the next three sections we'll present the most relevant works found inside each of the three topics that we mentioned in the same order, starting from the most general automatic generation of value from textual resources until the works closest to our work.

## 3.2 Related works with target out of Software Engineering area

For this topic we selected three of the most valuable works found which had been of a great contribution for the current work in terms of NLP techniques used and also due to their results.

The article Sabira e Uthradevi (2017), proposes the use of the Stanford NLP Tool together with the analysis of feelings to explore behavioral factors and sentimental factors and predict virality of the content of the blog.

The work from dos Santos e Gatti (2014), proposes a **deep convolutional neural network** [1] that extrapolates from the character to the level of the information in the sentence to only then perform the analysis of feelings [2] in short texts, obtained from phrases posted on Twitter. This network was called "Character to Sentence Convolutional Neural Network (CharSCNN)" and uses two convolutional layers and with this "the network extracts resources from the level of

[1] The main characteristics of the convolutional neural networks, besides the multi-layers, are the operations between Matrices, application of filters, borders, and the necessity of several Kernel parameters, such as: size, step/jump, weights and padding.

[2] The importance of the Analysis of Feelings in this type of text is quite useful to evaluate opinions, based on tweets, for example, it is possible to perceive the level of acceptance on any particular subject that people are posting about.

the character to the level of the sentence", which, according to the authors dos Santos e Gatti (2014), "allows you to manipulate words and phrases of any size".

According to the authors, for each sentence CharSCNN performs a process that consists in calculating a score for each feeling label, to mark a sentence, the network uses as input the sequence of words in the sentence and passes it through a sequence of layers where features are extracted, among them the embbedings [3] with increasing levels of complexity. The authors attest that the experiments show the effectiveness of CharSCNN for analyzing feelings of texts from two domains: Revisions of movie sentences and Twitter messages (tweets). "CharSCNN achieves state-of-the-art results for both domains [...] in addition, in our experiments we provided information about the usefulness of pre-training and the contribution of character-level characteristics to detect negations" dos Santos e Gatti (2014).

Shu et al. (2016) proposes an automated method for the construction of ontologies [4]. According to the authors Shu et al. (2016), they used techniques such as machine learning and Natural Language Processing to construct more precise ontologies, the Naive Bayes algorithm for categorizing text and determining the labels of a document. The SVM algorithm was used to calculate the document similarity with documents similar to the cluster they imported and also the Luhn's algorithm of summarization to shorten the text and maintain key terms for the domain ontology. The authors Shu et al. (2016) point out that at the end of the development of their approach "the XML format is used to represent a domain ontology, which has the function of describing the meaning of the term and its lexical relations."

## 3.3 Extraction of Software Engineering relevant information from texts using NLP

In this section we'll present the a set of works which involve text processing to extract relevant information to the Software Engineering area. These studies were chosen using the criteria of avoiding repeated subjects or contents and considering the relevance in terms of achievements, techniques and technologies used. Even if they are not so close to the subject of this dissertation, some of them helped to find the best technologies to use for natural language processing and to understand the current state of the art.

The work Pinquié et al. (2016) proposes a pipeline [5] of natural language processing (NLP) to extract the requirements of prescriptive documents. Second, it shows how machine learning techniques can be used to develop a text classifier that will automatically classify the requirements into disciplines. The proposed contributions are intended to support companies that are willing to power a prescriptive document requirements management tool, normally companies that use Product Lifecycle Management (PLM) tools. According to the authors Pinquié et al. (2016), "the experience with natural language processing showed an average precision of 0.86 and a recall average of 0.95, whereas the SVM requirements classifier exceeds *Naive Bayes* with an accuracy rate of 76% ".

[3] A word embedding is a numerical representation of the word. For a neural network such as *word2vec*, for example, they can use N numbers, for each there is a dimension, and each finds a word, in the space of N dimensions. Once the word embeddings have been trained, they can serve to derive similarities between words.

[4] Ontology, according to Trim (2012), is the semantic context that gives the words within of a sentence, that is, it describes the meaning of terms and their lexical dependencies. Ontology is also an area inside of natural language processing.

[5] The word pipeline can be understood as a pipe or tubular conductor, in the sense of a sequence of processes along a tube

The article Sugawara et al. (2017) proposes a methodology that is based on the ability to understand reading through NLP, and tests this methodology by annotating the understanding of the machine “Test” (MCTest) and analyzing four systems in it, including a neural system. According to the authors Sugawara et al. (2017), "the annotation results showed that answering questions requires a combination of skills and clarified the kinds of capabilities systems need to understand natural language." Therefore, they concluded the set of prerequisite skills they defined are promising for the decomposition and analysis of reading comprehension (RC).

The article Maldonado et al. (2017), states that "it is possible to detect technical debt using source code comments and that the most common types of self-admitted technical debt are project debts and requirements”. Using NLP, the authors studied 10 open source projects: Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and SQuirrel SQL and were able to accurately identify the self-supported technical debt, the current state-of-the-art based on key phrases and phrases. According to the authors, "90 % of the best rating performance was achieved, using only 23% of comments for the project and requirement for self-admission of technical debt, and 80% of best performance using only 9% and 5% Of the comments for design and requirement self admitted technical debt, respectively ".

The article of Bozyiğit et al. (2016), shows a CASE tool called AutoClass that extracts class diagrams and generates C# source code from the requirement specification using NLP techniques. According to Bozyiğit et al. (2016), “results indicate that the proposed system is more accurate in terms of extraction of class diagrams when compared to other studies (...) it has a precision rate that is more than 90%".

Analyzing software reviews and extracting from them items that can improve software development is a difficult task, so that, the article from Panichella et al. (2016), proposes a tool called ARdoc, which "combines three techniques: natural language analysis, text analysis, and sentiment analysis to automatically classify useful comments contained in major application reviews for performing software maintenance and evolution tasks" Panichella et al. (2016). According to the authors, "ARdoc correctly classifies useful feedback for maintenance perspectives in user reviews with high accuracy ranging from 84% and 89%, recall ranging from 84% to 89% and F-Measure ranging from 84% to 89%.

The article Badihi e Heydarnoori (2017) explores crowdsourcing, gamification and natural language processing to automatically generate high-level summaries of Java program methods, and have created an Eclipse plugin to deploy this idea along with a code summary game.

The article of Singh et al. (2016), proposes to combine automated identification and classification of mandatory sentences into non-functional requirements subclasses (NFRs) with the help of rule-based classification technique using thematic papers and identifying the priority of NFR sentences extracted in the document according to its occurrence in several NFR classes. The F1 Measure of 97% is obtained in the corpus PROMISE and F1-Measure of 94% in the Concordia RE corpus. According to Singh et al. (2016), “the results established validates the claim that proposal provides specific and higher results than the previous state of art approaches”.

Ramnani et al. (2017) is an article that proposes a new way of extracting information for the area of cyber security, using natural language processing techniques and a pattern identification structure.

The article from Masuda et al. (2016a), shows “a practical approach to detecting logical inconsistencies by clustering technique in natural language requirements”. The authors state have been used the algorithm *k-means clustering* “to cluster chunks of requirements and develop semantic role labeling rules to derive conditions and action as semantic requirements roles using natural language processing” Masuda et al. (2016a). After implementing their approach, the

authors conducted experiments in three case studies of requirements written in natural English and state the results show that their approach may encounter logical inconsistencies.

For Ye et al. (2016), the Software Engineering knowledge can be found in many documents, even reports of bugs [6], discussions in online forums, but despite being in natural language, is full of domain terms, software entities, and software-specific informal languages, which are not normally correctly labeled by existing Part-of-Speech (POS) tagging techniques. Therefore, the article of Ye et al. (2016), presents a specific POS tagger software, called S-POS, that was developed to process textual discussions in Stack Overflow, which, according to the authors, is a site that "has become an important repository of developer-generated knowledge for software engineering". To develop their approach, the authors, Ye et al. (2016), defined a set of POS tags that is suitable for software engineering, selected a corpus, developed a custom tokenizer, put annotations or labels on data, design features for supervised model training, and demonstrated that the S-POS's marking accuracy exceeds that of Stanford POS Tagger by labeling software engineering texts.

The article from Zolotas et al. (2016), presents a "Model-Driven Engineering (MDE) engine, which supports the rapid design and deployment of Web services with advanced functionality". According to the authors, the system entry consists of textual requirements and graphic storyboards, and is analyzed using natural language and semantic processing techniques, to semi-automatically build the input model for the MDE engine. According to the authors, "the engine subsequently applies model-to-model transformations to produce a RESTful, ready-to-deploy web service". The authors also ensure that their implementation is traceable and that changes in requirements propagate to artifacts and software models generated.

The article from Ghosh et al. (2014), describes the development of a tool called ARSENAL, which promises to automatically generate an intermediate representation between semiformal requirements and formal descriptions that can be used by software tools based on requirements.

The ARSENAL tool performs its function of transcription from semiformal to formal language using a sequence of steps. Initially the text in a semiformal natural language passes through the **Stanford Parser** which is a library of the **Stanford CoreNLP**, a set of libraries for Natural Language Processing written in Java, that are available for free by **Stanford Natural Language Processing Group**. The Stanford Parser is responsible for preprocessing text and tasks such as named entity recognition, between others. The next step is done by the Stanford Typed Dependency Parser (STDP), the STDP parser parses the requirements text and extracts unique entities called mentions, while the dependency parser generates grammatical relations between the mentions. According to Ghosh et al. (2014), the output is a triple set of dependencies typified between the extracted terms, which encode grammar, where each triple set indicates a relation of types: relation name between the governor and dependent terms.

The objective of the initial phases of ARSENAL is to generate an "Intermediate Representation (IR) table", for each Requirement text that is processed.

According Ghosh et al. (2014), the ARSENAL tool was tested on complex requirements of trusted systems in several domains, such as: **FAAIsolette** systems requirements and **TTEthernet**, in which it evaluated its degree of automation and robustness for the perturbation of requirements, and achieved excellent results. However, the tool only does the parse between the requirements already listed in semiformal language into formal language, correcting ambiguities, co-referencing, redundancies and inconsistencies. That is, the tool needs the requirements written in semi-structured language stylized as initial input, what differs ARSENAL from what is being proposed in this dissertation. Other benefits of ARSENAL include:

[6] Bug is a term related to a defect found in a system.

- Example generation of test cases,
- Exploration of putative theorem,
- Traceability to connect implementations to requirements they implement,
- Feedback on quality of requirements and suggestions for improvement to the end user.

The work from Ghosh et al. (2014) was very useful for this dissertation as the authors have presented a good way of working with Natural Language Processing using the Stanford CoreNLP library and Java. The performance obtained with Stanford CoreNLP in their work has inspired us to search and check all the functionalities of this NLP library, which was very helpful during the development. Even the goal of ARSENAL and our goal being different, it was important to know that Stanford can provide dependency graphs, a great accuracy for Part-of-Speech Tagging and other relevant features for Natural Language Processing.

In the table 3.1, we summarized all the works mentioned in this section with their respective authors reference, core techniques/technologies used and their valuable results or achievements.

## 3.4 Automated generation of artifacts for Agile Software Engineering processes using NLP

In this section we'll present the works that had a connection with the automated generation of artifacts that could be useful for the agile methodology, using natural language processing techniques, which would be the domain of research closest to this dissertation.

One of the most relevant results from the second search string was the article from Olaverri-Monreal et al. (2013).

Agile software development teams tend to use TDD, Test Driven Development, and therefore make use of unit testing to help develop code with higher quality and maintainability. According to Zhang et al. (2016), "one of the most useful sources of information for understanding a test is its name". The article from Zhang et al. (2016) suggests a new approach for automatic generation of names describing tests for existing test bodies by combining analysis of natural language program and text generation to create names that summarize the test scenario and the expected result. For the authors, the results show that, compared to alternative approaches, the names generated by the proposed technique are: "significantly more similar to human-generated names and are almost always preferred by developers, are preferred over or are equivalent to the original names of the test in 83% of cases" Zhang et al. (2016). The authors also point out that the proposed technique yields results faster than writing test names manually.

During the agile process of software development it is common to see the use of user stories, to specify the tasks that should be developed, as well as the use of test cases connected to each user story using the standard Behavior Driven Development (BDD). Based on the need to generate Test Cases in BDD format, it was created the article from Masuda et al. (2016b), which deals with the automatic generation of test cases using NLP techniques to select sentences from requirements based on syntactic similarity and then to determine conditions and actions through dependence and case analysis.

According to Rane (2017) "most projects in the industry follow a Behavior-Driven software development approach to capturing requirements from the business stakeholders through user stories written in natural language". For this reason, Rane (2017) proposes the automatic

Table 3.1: Software Engineering area related works list

| Reference | Techniques Technologies used | Valuable results or achievements |
|---|---|---|
| Pinquié et al. (2016) | NLP and machine learning | A text classifier to extract disciplines from requirements |
| Sugawara et al. (2017) | NLP | defined prerequisite skills that are promising for the decomposition and analysis of reading comprehension (RC) |
| Maldonado et al. (2017) | NLP | 90% of the best rating performance achieved, detecting self-admission of technical debt, with only 23% of comments for the project and requirement |
| Bozyiğit et al. (2016) | NLP | AutoClass tool that extracts class diagrams and generates C# source code from the requirement specification, with a precision higher than 90% |
| Panichella et al. (2016) | NLP text analysis sentiment analysis | The ARdoc tool correctly classifies useful feedback for maintenance perspectives in user reviews with both F-Measure and accuracy ranging from 84% and 89% |
| Badihi e Heydarnoori (2017) | crowdsourcing, gamification and NLP | automatically generate high-level summaries of Java program methods |
| Singh et al. (2016) | rule-based classification and NLP | automated identification and classification of mandatory sentences into non-functional requirements subclasses (NFRs), with a F1-Measure between 97% and 94% |
| Ramnani et al. (2017) | NLP | new way of extracting information for the area of cyber security |
| Masuda et al. (2016a) | k-means clustering and NLP | a practical approach to detecting logical inconsistencies by clustering technique in natural language requirements |
| Ye et al. (2016) | NLP | specific POS tagger software: S-POS, developed to process textual discussions in Stack Overflow. Its accuracy exceeds that of Stanford POS Tagger by labeling software engineering texts |
| Zolotas et al. (2016) | NLP | Model-Driven Engineering (MDE) engine subsequently applies model-to-model transformations to produce a RESTful, ready-to-deploy web service |
| Ghosh et al. (2014) | NLP | a tool called ARSENAL, which automatically generate an intermediate representation between semiformal requirements and formal descriptions |

generation of test cases in the context of agile software development using user stories based on natural language and acceptance criteria. The authors said that just the information contained in the user stories wouldn't be enough to generate the test cases through natural language processing, then they used the Test Scenario Description and Dictionary as inputs as well. The authors from Rane (2017) states that they "developed a tool that uses NLP techniques to generate functional test cases from the free-form test scenario description automatically [...] and the tool reduces the effort required to create the test cases while improving the test coverage and quality of the test suite".

Another work that offers benefits to agile processes is the article from Merten et al. (2016) proposes the use of natural language processing and machine learning resources to detect requests of software resources through natural language data of problem tracking systems. According to the authors, Merten et al. (2016), "because problem tracking systems also contain reports of bugs or programming tasks, requests for software features are often difficult to identify". The article compares the traditional features of linguistic machine learning, such as "bags of words", with features as subject-action-object, and evaluates combinations of machine learning resources derived from natural language and features taken from meta data [7] of the problem tracking system. According to the authors Merten et al. (2016), problems with data fields that contain requests for software features can be reasonably well identified, but hardly the exact phrase.

The article from Quirchmayr et al. (2017) proposes a "semi-automatic approach allows to identify and extract atomic software feature-relevant information from natural language user manuals by means of a domain glossary, structural sentence information, and natural language processing techniques". According to the authors, their experiments presented a precision and recall over 94% E 96%, respectively. Besides the implementation of their approach, the authors stated that they produced "corresponding evaluations based on example sections of a user manual taken from industry".

In none of the literature searched anyone had a goal equivalent to the one that is aimed in the proposed approach of extracting user stories information through big picture texts containing the idea of a desired software. The work involving the automated generation of artifacts focused on the agile software development process closer to the proposal of this dissertation refers to the automatic generation of Test Cases from User Stories.

## 3.5 Conclusion

Among all the related works there was no case that is identical to the proposal of this dissertation and also no article was found that generated artifacts such as user stories from unstructured text using Natural Language Processing techniques. One of the closest works found, the article from Ghosh et al. (2014), is a requirement generator of structured language through semi-structured text, that is, the initial text of that work had been already preformatted which reduced the difficulty of the tasks and yet the artifact generated is not the same as the one intended in this dissertation. In regards to works that involve natural language processing tasks for generating useful artifacts to the agile process of software development, the closest found was the article from Rane (2017), which deals with the automated generation of test cases from user stories already defined. Some papers have also dealt with classification and labeling of sentences using techniques of natural language processing or machine learning, but these works are also far from this proposal. This dissertation proposes the automatic generation of user stories from

[7] Meta-data means the data or information about the data itself.

unstructured text that contains a software specification or big picture of the desired software using natural language processing techniques. The development of this approach will bring a gain to the Software Engineering area in a subarea not yet covered by automated processes, at least the ones using natural language processing for the purpose of user stories generation.

# 4 Design and Implementation of User Stories Generation Tool

In this chapter, we'll explain the UserStoryGen approach and the core contribution of this work: a tool for automating the user stories generation. In the next sections we'll present the high-level system design, a drill down of the implementation, describing how the software was built and what are the expected inputs and outputs for each module of the software that implements this approach. The pipeline structure and the author algorithms created to deal with special cases of natural language issues will also be explained in great detail.

## 4.1 Context

In spite of defining user stories can be the job of a Business Analyst inside an Agile team, it's possible to reduce the time of this task by doing it automatically. The purpose of this work is to show that it is possible to create a tool that automatically extracts User Stories from an unstructured text that contains information about the desired software, this can reduce time and cost in an agile development process.

The high-level design idea of the proposed tool is to automatically generate user stories from software big picture text or a software specification in textual format. The output of the proposed tool should stay as close as possible to the existing User Story description used in the industry, and also to extract the user story title, the user type/actor and system entity information. The text language that the tool will accept in the input text is only English.

The complexity to achieve this goal relies on many points, such as co-reference resolution to parse the personal pronouns and get the actual actor name, identifying entities and the main verb, simplifying the phrases that have many verbs, treating the negative phrases and also treating the passive voice sentences which have a more complex structure. To solve all the possible issues while processing the sentences, it was decided to have two pipelines containing all the necessary steps to deal with them and also having a high performance.

Pipelines structure that will be shown in the next sections was the way found to manipulate all the information and sequence of natural language processing tasks. The pipelines helped to order the tasks that should be executed and also the rules that should be applied and validated to achieve the result in an efficient fashion to save time. This structure is also called pipeline due to the idea of having multiple tasks being applied in a loop of all the sentences that are passing throughout the pipe, one after the other until the end of the input text.

## 4.2 Approach

In this dissertation we're proposing an approach called UserStoryGen, which will have its respective implementation by the UserStoryGen tool. The core idea of this approach is to receive a user input in the format of a text file containing any software specification or *Big picture* of the desired software and then iterate over the text executing the text processing until to generate the output, that will be a list of possible user stories.

The UserStoryGen approach illustrated by Figure 4.1 consists in 3 basic steps:

1. UserStoryGen Rest API: receive the original text through a file;
2. First Pipeline: execute a pre-processing phase;.
3. Second Pipeline: receive a semi-processed sentence and a Dependency Graph. Use these inputs plus static lists of Negative expressions and Stop Words to generate the possible User Stories for the given text.

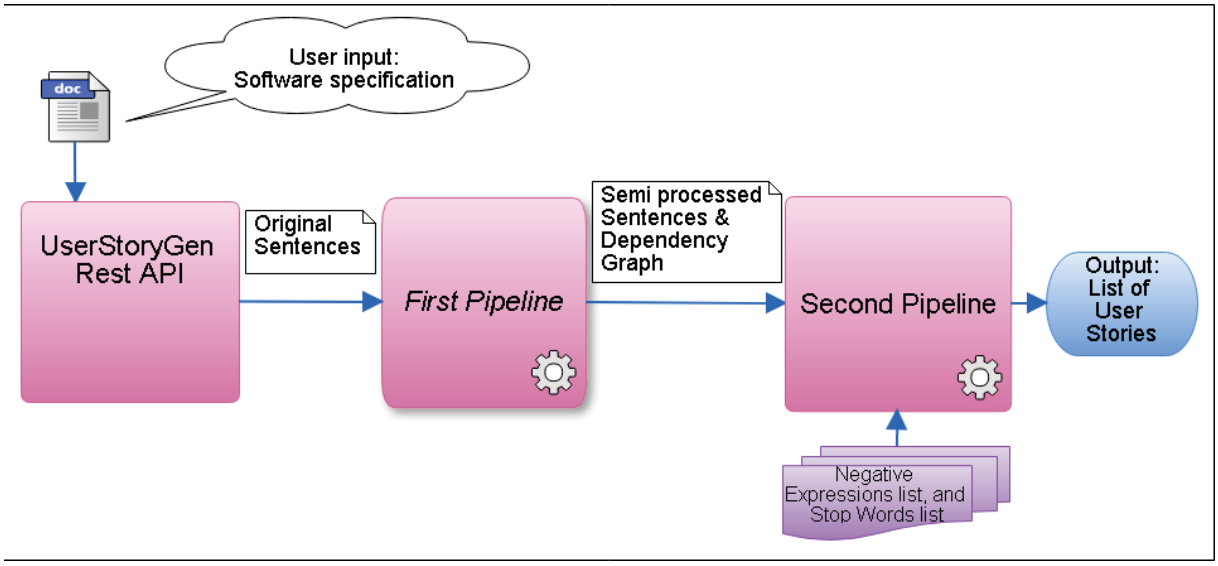



Figure 4.1: Approach diagram

The most difficult part of this approach remains in the text processing phases in which the natural language processing tasks should be correctly splice to result in a good execution time.

## 4.3 Design goals

The goal of implementing UserStoryGen tool is being capable of automatically generate user story information, that are valuable to Agile process, such as user story title, description, user/actor and system entity involved in the context of each user story. All the information that will be processed and generated from the text should help in the Agile process, making easier the work involved in the user stories definition, and all the system entities recognized also can help in the Feature definition, saving time from the business owners, or business analysts who use to work on user stories definitions.

The only input required by the tool is a text containing the information about the desired software, that can be called as software specification and sometimes is described as the software big picture. The other piece of data used during natural language processing phases is static and compound by a list of negative words used for filtering sentences and later recreate them keeping the negative sense, and also a list of punctuation tokens that act like a **stop word** list.

The expected output of this tool is one list of user stories, in which every item will contain:

- The story tile, compound of a verb plus its direct object, just to have a short description of the user story goal;
- The user story description in a format similar to the one quoted by Rasmusson (2010) that will be: "*As a <type of user> I want to be able to <some goal>*", in which we are just omitting the "*so that <some reason>*" piece;
- The main verb;
- The user;
- The system entity involved in the story, which can be found in the complement of each verb.

The output of the system could be seen in either *JSON* format if it was used only the call for the *Restful API*, and in a web page as well.

## 4.4 Design challenges and limitations

One of the most difficult challenges in this tool is to produce the user story information at a high-level of details as the Agile projects use to have to help defining what should be done within a user story, and by whom it should be done. The other part of the challenge is to extract the system entity present in each sentence, what can give more direction about the impact of a specific story in the database.

All the information that it's intended to extract with this tool should help with the user stories definition of the current Agile workflow, and also has a positive impact by helping in the planning, because it's possible to infer how many stories will be needed to develop the requirements exposed in the input document of software specification, and errors or missing information can be identified earlier seeing the list of user stories.

The text passed as the input should be written in English, and at least must avoid incorrect words and try to mention the actors of each action, because even in case of sentences written in passive voice that can be treated, the algorithm created will search for the subject and in case of not finding it, the result won't be as accurate as desired.

The tool is currently supporting most of the cases that could be tricky to handle in the natural language processing, such as negative sentences, multiple verb sentences, sentences with enumeration of actions related to a common subject, active and passive voice, and all kinds of verbal flexing, because the implemented algorithm always transforms the verb in its infinitive form, called *lemma*, to match the user story pattern of description.

Another challenge during the development of this tool was to treat the co-reference resolution between the sentences to replace pronouns with the actual user information, and to decide in which moment it should be better to replace this information in the pipelines execution. After reviewing the steps for the sentences simplification task; that in the same sentence interaction

does all the extractions needed for the output; it stayed clear that this replacement for pronouns should be done before the sentences simplification task. Then, right after transforming the text into a dependency graph, in the first pipeline, the user information is replaced in the same position as the pronoun was, and only after this the sentences can be sent to the second pipeline.

## 4.5 System design

The system design of the proposed tool is described by Figure 4.2, which shows the layers of the system and how they interact one with each other, since the input arrives at the Rest API until the final answer is produced by the Natural Language module.

The user provides the software specification text or the desired software description in textual format as the input, and the proposed software adds a list of stop words to help on text processing.

As Figure 4.2 shows, the text passes through a Rest API [1], that sends the text to a Service class which is responsible to forward it to the pipelines execution. In the first pipeline the two key points that happen are the co-reference resolution with the replacement of pronouns by its root meaning and the generation of the dependency graph with the sentences containing this little adjustment of the pronouns.

Then in the second pipeline an author algorithm simplifies the sentences, taking in consideration the static negative list of words and stop words shown in Figure 4.2 and other checks such as type of voice, etc. Then, for each simplified sentence, the information such as subject, main verb and system entity are extracted and the title is created with the processed information as well as the user story description. By the end of each iteration of a simplified sentence, all this information related to the user story object is saved and added in a list of user stories objects, that is the output of the second pipeline execution.

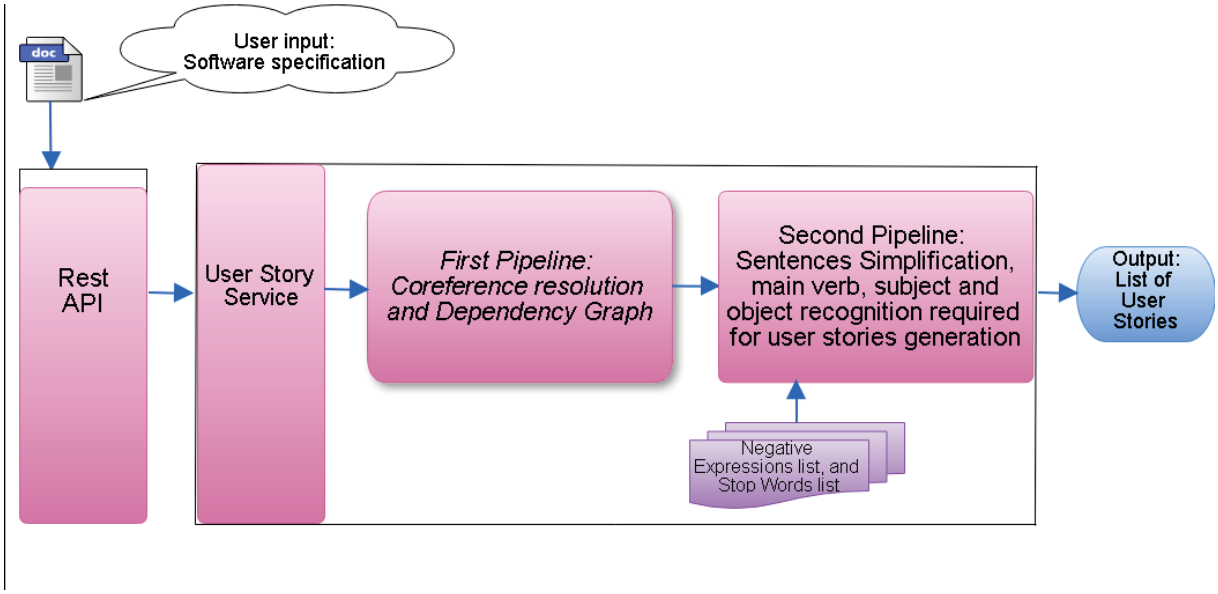



Figure 4.2: High-level system design

[1] "According Fielding (2000), "the Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system". For Masse (2011), "having a REST API makes a web service “RESTful.” According Masse (2011), a REST API consists of an assembly of interlinked resources and this set of resources is known as the REST API’s resource model.

## 4.6 System architecture

The system architecture is compounded by a user interface that does the document digestion, and throughout this user interface the text containing software specification is sent to the back-end to be processed and return a list of user stories.

When the user confirms the information he wants to process, the back-end receives a REST call with the file containing the software specification text inside REST API layer and pass this text to the Service layer. The Service layer is responsible to call First Pipeline, grab the result and send it to the Second Pipeline that will produce the final answer which is the list of User stories, as you can see in the diagram below.

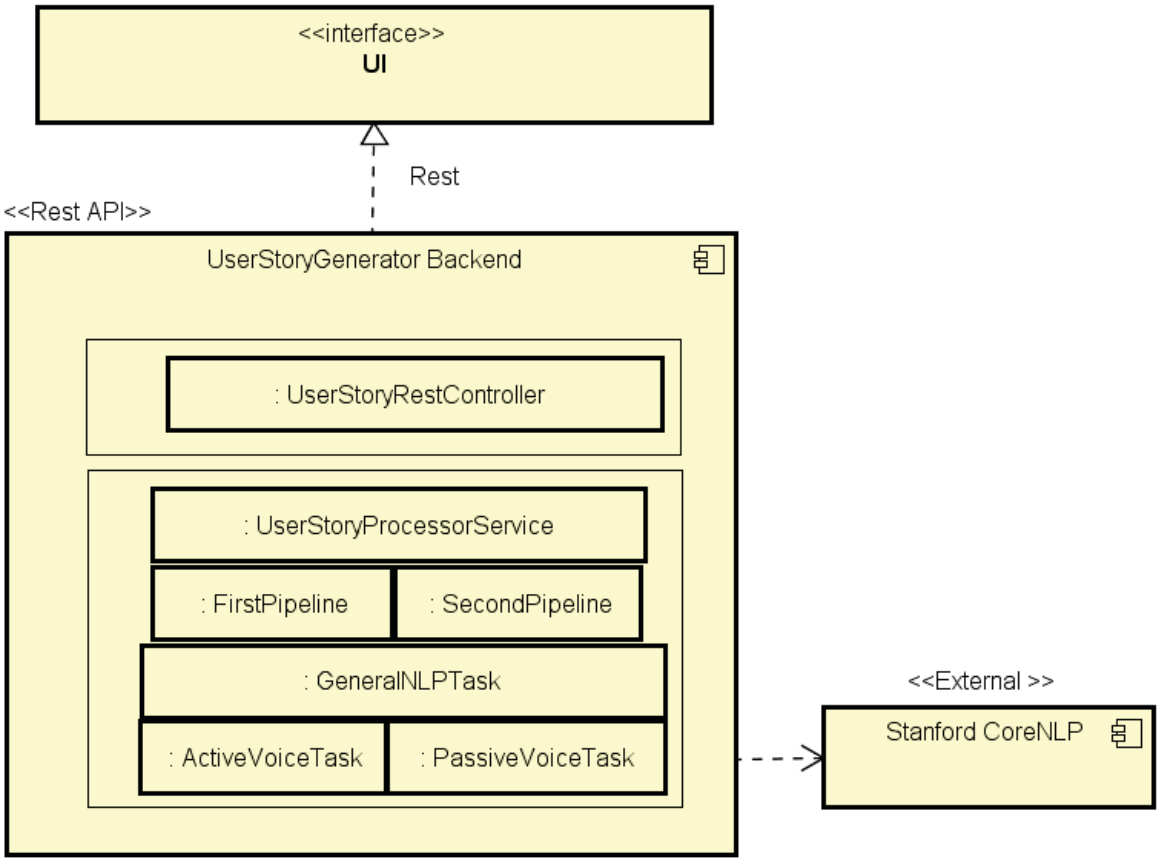



Figure 4.3: System Architecture

As Figure 4.3 shows, the *UserStoryProcessorService* class interacts with the *FirstPipeline* and the *SecondPipeline* class, and these pipeline classes calls *GeneralNLPTask*, *ActiveVoiceTask* and *PassiveVoiceTask* classes to execute specific Natural Language Processing tasks. *GeneralNLPTask* class concentrates all the *regex*[2] filtering methods responsible which aren't specific for one type of sentence voice, like collecting all the subjects from the text, remove stop words and so on.

To finalize the work generating the user story information, extracting the subject, main verb and system entity the *SecondPipeline* class uses the *ActiveVoiceTask* class to handle active voice sentences and the *PassiveVoiceTask* class to deal with passive voice sentences.

In the case of passive voice sentences, the process of generating the user story information uses different techniques to filter every type of information and also to split the sentence when

[2]Regex is a short term for regular expressions.

needed. Because of that, the *PassiveVoiceTask* class has methods completely different from the *ActiveVoiceTask* class.

It's also possible to note in Figure 4.3, that the Service layer has an external dependency with the Stanford CoreNLP library, and this happens because it was the NLP library chosen to help in classifying the Part-of-Speech Tagging, the co-reference mentions and it produces the dependency graph that is used in the Second Pipeline.

## 4.7 Implementation Details

The implementation of UserStoryGen tool has a user interface written using Angular 4, which is a JavaScript Single Page Application framework, and this user interface is responsible for sending the uploaded file to the back-end and later receives the final result, the list of user stories.

The back-end consists in a Restful Service that is the consumer of the input text and also sends the text to other internal module specialized in Natural Language Processing. The NLP module is be responsible for the hard work of treating the text, processing all the information using NLP techniques and producing the User Stories list to send it to the end point that has called it.

The most difficult part developed in UserStoryGen resides in the Natural Language Processing module, because the other parts of this software just aim to grab the text, send to this module and wait for the response to later show to the user. Then, it's possible to affirm that the Natural Language Processing module is where all the things happen, in terms of the real value that is produced within this tool.

It was decided to use a strategy of dismembering the natural language processing tasks during the *NLP* phase graphically described in Figure 4.4. To do this, two pipelines were created and inside them the natural language processing tasks were logically separated to improve the software performance.

The first pipeline, which can be seen in Figure 4.4, has the tasks such as: Sentences Tokenization, Part-of-Speech Tagging, NER (Named Entity Recognition), Coreference Resolution. These tasks appear in violet rectangles and are all tasks that run through Stanford CoreNLP annotators, which produce a dependency graph and a list of mentions as result.

The fifth step in the first pipeline, which appears in blue color, was implemented by the author and aims to identify all the subjects in the texts, that can be either nouns, proper nouns or pronouns and replace the mentions of the subject with the subject itself. To simplify the sentences, it's needed to have the exact information about the subject for which the pronouns point to, to then replace the pronoun with the correct mention of it: the real subject data. After the replacement of pronouns the fifth step returns an up-to-date dependency graph which is sent to the next Pipeline.

In the Second Pipeline in Figure 4.4 it's possible to see that all the tasks appear in blue color figures, as they are all created and implemented by the author of this paper. The tasks are following a central logic that is to find the number of verbs and root verbs of each sentence, because the verbs can determine how to treat each sentence and the validations over they can also discard partial sentences that are created during the natural language processing.

The first task in the Second Pipeline is to **extract the main verb and check how many verbs the sentence has to later split it**, and this is necessary because for one verb sentences the treatment is very easy, but for multiple verbs sentences the algorithm is complex.

The following task is to **check negative words in the sentence and if it's in passive voice to process it in a different way**. When this task is executed, it is checked the presence of

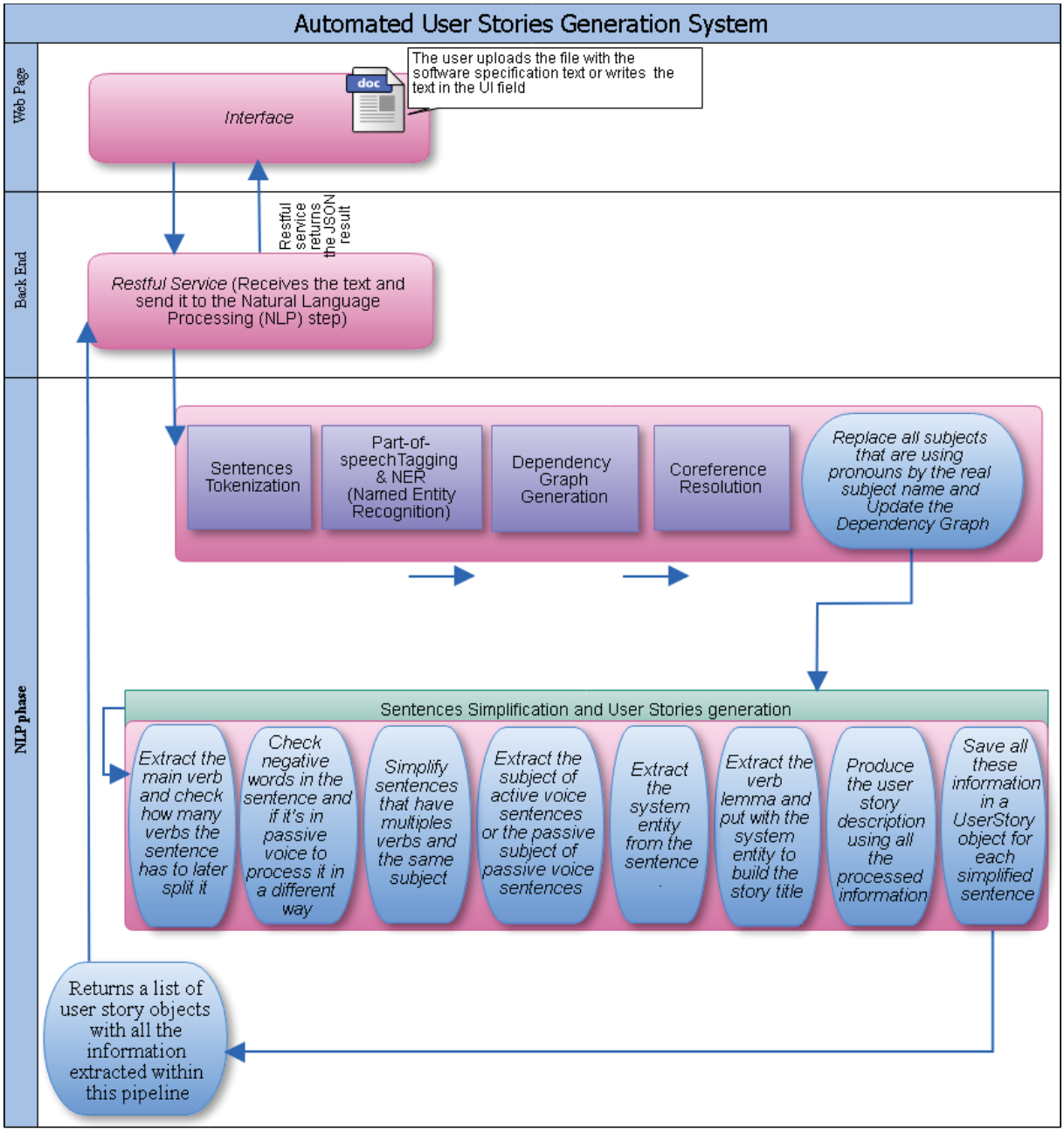



Figure 4.4: Software development process diagram

negative words such as: *no, not, can't, shouldn't*, for example, and if it's a negative sentence it's considered in the simplification phase. Besides this, if the sentence was written in passive voice, then it goes for a different flow to be fully processed and later follows the other common steps that are applied for all sentences.

The third task that appears in the Second Pipeline, shown in Figure 4.4, is to **simplify sentences that have multiple verbs and the same subject** and was the hardest one to implement that will be explained in great details in the following sections. The fourth task is to **extract the subject of active voice or the passive subject of passive voice sentences**. Besides the importance of finding the subject as it's one of the attributes of the *UserStory* object generated by the end of each sentence processed, the subject is used to recreate the sentence to produce the user story description.

The fifth task in the Second Pipeline is to **extract the verb lemma and put with the system entity to build the story title** and with the user story title created it's only missing the sixth task that is to **produce the user story description using all the processed information**. With these last tasks done, it's possible to claim that the *UserStory* object is filled for each sentence. Then the seventh and last task of the Second Pipeline is to **save all these information in a *UserStory* object for each simplified sentence**.

As appears in Figure 4.4, the result of the Second Pipeline is a list of *UserStory* objects that is sent as the final result to the Restful Service, and is forwarded by the Restful Service to the User Interface in a *JSON* format, to be shown to the user.

In the next subsections the pipeline implementation details and author algorithms are explained, and it's possible to see the inputs and outputs of each pipeline.

## 4.8 NLP library and programming language choices

The article called *Comparing the Performance of Different NLP Toolkits in Formal and Social Media Text*, from Pinto et al. (2016), reveals what are the best libraries for treating text from Twitter and *social media*, but, also point to an analysis, evaluating only the libraries performance related to a diversity of texts. The authors point that Rodriquez et al. (2012) and Atdag e Labatut (2013), had compared different *NER* (Named Entity Recognition) tools applied to different text types, using *Stanford CoreNLP*, *Illinois NER*, *LingPipe* and *OpenCalais*, on a set of Wikipedia biographic articles annotated with person, location, organization and date type entities. And, according to them, the library that achieved the best performance was the *Stanford CoreNLP* with overall F1 results of 90%, while *OpenCalais* achieved only 73%.

In the second chapter, many examples of *Natural Language Processing* tasks and techniques were shown, and the most part of them using code examples with the NLTK libraries, because it's the most used library for didactic cases, due to the simplicity of its code among other factors. However, during the researches we identified that the major number of works, mainly the denser ones involving NLP use the *Stanford CoreNLP* library, which is more focused on robust tasks and the ones with greater complexity. This library has both a greater number of features and efficiency compared to the others.

Based on all the possibilities that the *Stanford CoreNLP* library brings, in addition to other Stanford toolkits such as the *Named Entity Recognition* (NER), *Stanford Deterministic Coreference Resolution System*, *Stanford Parser*, among others, and based on the possibility of adding annotations created by third parties, which help us to create our own rules for processing text for our specific purpose, it's possible to say that the Stanford toolkit is still the best choice for the state-of-the-art natural language processing.

The *Stanford CoreNLP* factors such as: free distribution, performance, functionality, and Java library availability that facilitate the abstraction of concepts in an object-oriented implementation were determining for the choice. As *Stanford CoreNLP* is written in Java, we decided to use Java and Spring Boot Framework in the back-end to create our Restful service, and we created the User Interface with the Angular 4 (JavaScript Framework).

### 4.8.1 First Pipeline implementation details

The first tasks which run in the First Pipeline implementation are working throughout the Stanford CoreNLP library and they're respectively: Sentences Tokenization, Part-of-Speech Tagging, NER (Named Entity Recognition), Coreference Resolution. These tasks are solved using the Stanford annotators and pipeline, which returns a dependency graph and a Mentions

list. The dependency graph contains all the relation between the words in the sentence, how they are connected, and also holds all the information like the *POS*, *lemma*, which word is the root of the sentence and so on.

Thus, the next step was basically to resolve the coreference between the subjects using a list of mentions and crossing this information with the list of all subjects found in the whole text. The list, containing all subjects in the text, was extracted by filtering in all the phrases to identify the subjects using the Chunk technique based in a *Regex* to query inside the dependency graph information. To do so, it was created the *GeneralNLPTask* class with a lot of methods to grab information from the dependency graph, because the Stanford CoreNLP doesn't have all the methods that are needed working as it was necessary for this work.

After replacing the pronouns by their related subject name, the algorithm regenerates the dependency graph for the affected sentences and replaces them in the correct node position inside the original dependency graph. This processing step results in sentences well structured in terms of subjects, and also up-to-date dependency graph holding the sentences with all its *metadata* produced by the NLP tasks, which is sent to the second pipeline.

Just to illustrate how is the output from the First pipeline, test the mention classification and the change in the sentence after the update, it was used sentences like these ones:

- "The Payroll Administrator can register all the company payments."
- "But, he can not delete payments information."

These sentences above are clearly talking about the same subject. The first is using explicitly the value "The Payroll Administrator" and the second is using the pronoun "He" to refer to this subject. Using these sentences as the input text example, at the beginning of the processing it was generated the dependency graph and the coreference mentions for these sentences as appears in Figure 4.5.

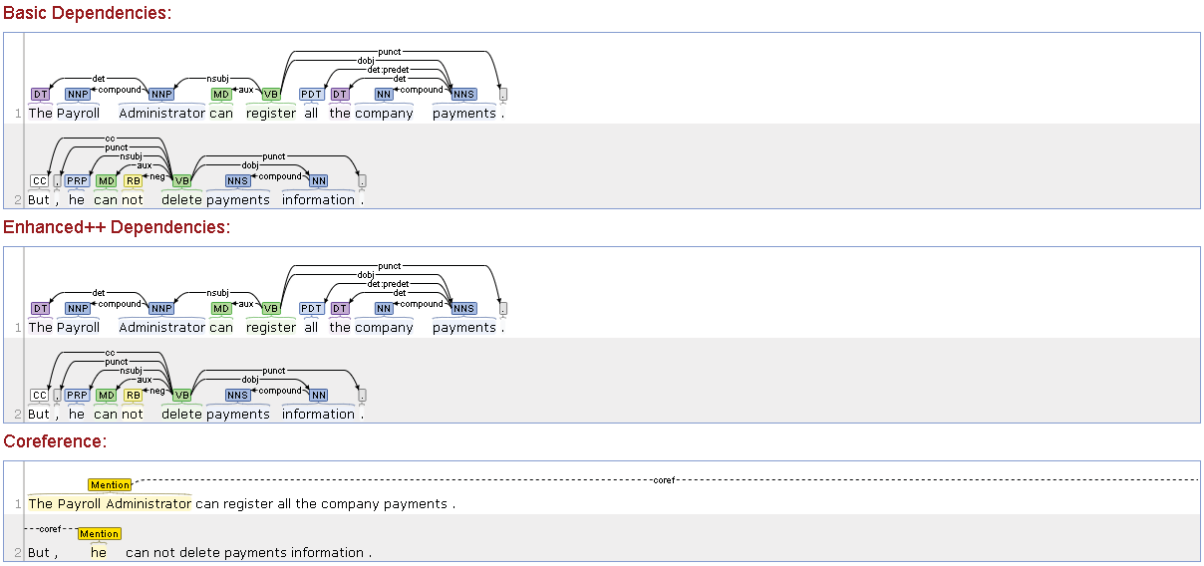



Figure 4.5: Dependency graph and mentions in the beginning of the First Pipeline

As it's possible to see in Figure 4.5, the pronoun "*He*" was identified as the mention of the subject "*The Payroll Administrator*". And that is how the proposed solution acts, replacing this mention of the subject *The Payroll Administrator* when the mention is different from this same value, that should be the case of the pronoun "*He*", of course. Then, the resulting text for the given example is:

- “The Payroll Administrator can register all the company payments”.
- “But, Payroll Administrator can not delete payments information”.

And the dependency graph that is sent to the next pipeline related with this example is the following:

```
(ROOT (S (NP (DT The) (NNP Payroll) (NNP Administrator)) (VP (MD can) (VP (VB register) (NP (PDT all) (DT the) (NN company) (NNS payments)))) (. .)))
dependency graph:
-> register/VB (root)
  -> Administrator/NNP (nsubj)
    -> The/DT (det)
    -> Payroll/NNP (compound)
  -> can/MD (aux)
  -> payments/NNS (dobj)
    -> all/PDT (det:predet)
    -> the/DT (det)
    -> company/NN (compound)
  -> ./. (punct)


(ROOT (S (CC But) (, ,) (NP (DT The) (NNP Payroll) (NNP Administrator)) (VP (MD can) (RB not) (VP (VB delete) (NP (NNS payments) (NN information)))) (. .)))
dependency graph:
-> delete/VB (root)
  -> But/CC (cc)
  -> ,/, (punct)
  -> Administrator/NNP (nsubj)
    -> The/DT (det)
    -> Payroll/NNP (compound)
  -> can/MD (aux)
  -> not/RB (neg)
  -> information/NN (dobj)
    -> payments/NNS (compound)
  -> ./. (punct)
```

Figure 4.6: First Pipeline Output - Dependency Graph

In Figure 4.6, it’s possible to see that the dependency graph creates a dependency tree between the words of the sentence, and the verb node appears as the root of this structure. Every word in the dependency graph has a *POS* declared between the slash and after it, between the parenthesis, appears the relation of the word in the structure of the sentence. For example, the word “Administrator” appears with a *POS* of “NNP”, which means a “Proper noun”, and in the parenthesis appears the information about its role in the sentence which is “nsubj”, what means that this word is part of the subject. Below of the node of the “Administrator” word appears the other two words that are part of this compound subject: "The" and "Payroll".

This dependency graph presented in Figure 4.6 is the output of the First Pipeline that is sent to the Second Pipeline where the processing continues.

### 4.8.2 Second Pipeline implementation details

The Second Pipeline is a module in the proposed tool which contains the algorithm to execute the most complex Natural Language tasks. It uses specific rules of extraction and also author algorithms to process the text in a way that will return the sentences correctly processed to build the user stories information.

After analyzing the texts that could be used as input for the proposed software, some requirements in terms of natural language processing tasks started to appear. For every difficult task, it was tried to use the resources from the Stanford library, but no all of these resources were producing a good result. Then, it was needed to create the algorithm to treat the following cases:

- Compound subject in both active voice and passive voice sentences;
- Sentences with more than one verb node, or when it’s identified one enumeration of actions connected with the same subject, but with different complements;

- Sentences with more than one verb, but with a single action using passive voice;
- Sentences using passive voice and enumeration of actions.
- Sentences with negative words related to the verb, either in active and passive voice;
- Identify and remove words with a sense of connection with the next sentence in the end of a sentence complement, but that don't have a significant meaning in the current sentence;
- Differ what is the subject, the verb, the verb complement and the system entity for a given sentence;
- Recreate every applicable sentence using the user story description pattern.

Figure 4.7 illustrates the tasks that occur during the Second Pipeline in a high-level of details to explain the logic of this approach. And the Sentences simplification author algorithm that has an extensive logic is presented by other activity diagram in Figure 4.8.

To treat all the specific cases and generating user story information, the second pipeline module considers 3 core flows that are illustrated by Figure 4.7:

- Single verb sentences with active voice;
- Multiple verbs sentences with active voice;
- Multiple verbs sentences with passive voice.

Recapitulating the state of the input text in the beginning of the second pipeline, the text received was already treated in the first pipeline and all the pronouns were replaced by the root subject, which is a specific meaning of the pronoun. The previous pipeline sent to the second pipeline a dependency graph that holds the treated text and all the classification for each word in the phrase.

Given that the Second pipeline receives a dependency graph holding the text and its *metadata*, then the algorithm applies all the tasks described in Figure 4.7 in a loop through each of the sentences. This means that every sentence will pass through the process explained in the activity diagram to generate a user story or discard the sentence, from the beginning until the end of the text.

The logic of this pipeline starts with the task of to **check number of verbs**, as it's possible to see in Figure 4.7. This task of getting the number of verbs is done via *chunking* technique, which consists in the use of a regex to filter only the parts of the text that have any variation of a specific Part-of-Speech Tag, which in this case should be equals to *"VB"*.

After knowing the amount of verbs in the sentence, the algorithm described by Figure 4.7 divides the processing tasks into two possible flows: **single verb sentences** and **multiple verbs sentences**. And after that the second flow also is divided into two options: **multiple verbs sentences + active voice** and **multiple verbs sentences + passive voice**. The splitting of the routine in 3 flows was a strategic decision to process the phrases according the common factors and to have a good performance by avoiding tasks that are not applicable to all the cases.

Discussing about the first flow in Figure 4.7, which is the logic to treat "Single verb Sentences". The flow of single verb sentences starts with the task of to **extract the subject or compound subject via dependency graph**. The logic for this task in terms of the algorithm is to go through the dependency graph and find the root node with the relation equals to "nsubj".

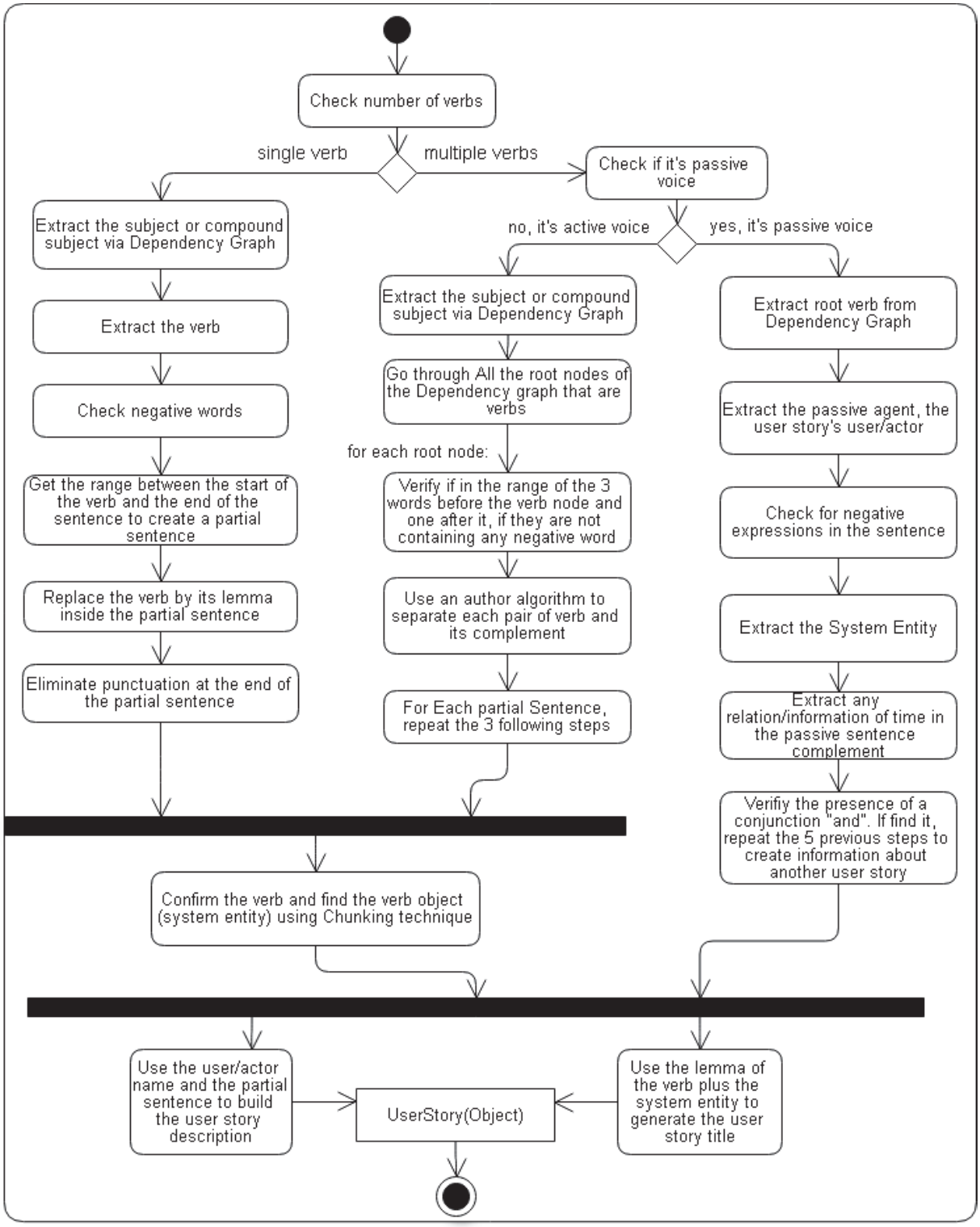



Figure 4.7: Second Pipeline Activity Diagram

Then, with this node found, it is necessary to navigate into its leaves and finding others nodes with their words if they exist, because it's possible to find either simple or compound subjects. After collecting the words that represent the subject, this information is updated in a variable that could be part of a new user story information.

The following task in Figure 4.7 is to "extract the verb". The verb is extracted via *chunking* technique using the following *regex*:

```
[{ tag:/VB.*/ } ]
```

Then Figure 4.7 shows the next task that is to **check negative words**. The goal of this task is to find any negative word with some relation with the verb in the analyzed sentence. To find negative words in the sentence, it's necessary to get the range of words which considers the three words before the verb and one after it, and find in this range of words, if there's any match with one of the negative words in the static list created by the author. This verification result helps to later create the user story title with right syntax, as well as user story description.

The following task in Figure 4.7, is to **get the range between the start of the verb and the end of the sentence to create a partial sentence**, and it is something simple to do as the verb was already extracted. This range of words extracted in this task is added into a *String* and named as partial sentence. Then, this partial sentence is used in the next task which is to **replace the verb by its lemma inside the partial sentence**.

After it comes the task of to **eliminate punctuation at the end of the partial sentence**. This task removes some words and punctuation at the end of the sentence that are not relevant in the text. With these changes, the partial sentence helps in the creation of the user story description during the next steps.

The next step in Figure 4.7 is to **confirm the verb and find the verb object (system entity) using *chunking* technique**, that is a shared task among two flows. The importance of this task resides in finding the system entity, which is the object of the verb, and is one of the attributes of the user story object. To extract the system entity and confirm the verb it's used a *regex* expression to get the root node and the object directly connected with it, this object is the system entity, and after getting the node of the verb object the strategy was to navigate in this node tree, to check if the node is a compound noun or not. If for some reason the regex doesn't find the matches, then it's used the dependency graph to extract the information looping through it until to find a relation equals to "dobj" (direct object), and in the case of not finding it, the other acceptable relations are: "nmod" (nominal modifiers of nouns or clausal predicates) and "ccomp" (clausal complement). After finding one of these nodes it's navigated through it to get the complete value of all its leaves. Then, the system entity information and the verb *lemma* are saved inside the user story object.

The following tasks in Figure 4.7 are common for the three flows, and also finalize the process of filling the information inside the user story object generated by the iterations of each sentence that passes on the second pipeline. They are:

- To "use the user/actor name and the partial sentence to build the user story description". In this case the description is generated with the template: "As a [user/actor] I want to be able to [partial sentence]", for positive sentences or for negative sentences: As a [user/actor] I should not be able to [partial sentence];

- To "use the lemma of the verb (verb in the infinitive form) plus the system entity to generate the user story title".

The result of the two previous tasks is that the user story title and description are done and they are saved in the user story object, what finalizes this flow of a single verb sentence.

Now, it's explained with more details how works the second flow that appears in Figure 4.7, the case of multiple verbs and active voice sentence. The first task for this flow is to **extract the subject or compound subject via dependency graph**, and it implies in going through the dependency graph and navigate into the node with the "nsubj" relation, getting the words inside

this node and its leaves if they exist. Then the subject words collected in this task should be saved in a variable to fill the user information in the possible user story to be generated.

The next task that appears in Figure 4.7 is to **go through all the root nodes of the dependency graph that are verbs**. This task introduces that it's necessary to navigate through the dependency graph, looking for nodes that are verbs, and that for each of those nodes there will be a group of tasks to be performed to simplify and to split the sentence.

The following task that appears in Figure 4.7, is to "verify if in the range of the 3 words before the verb node and one after it, if they are not containing any negative word", and given that we are going through the verb nodes, it's necessary to know if the partial sentence that is built for each verb has a positive or negative sense. As it's done in the first flow, the static list of negative words is used to check if there's any match with the mentioned range of words.

The descendants of each verb node gathered from dependency graph using the Stanford library were not coming in the correct way. During the tests the Stanford classification for descendants of the verb were failing and grouping more words than expected, instead of grab only one verb and its complement, sometimes the library was classifying one verb plus its complement and connecting also with other verbs and complements inside a single node of descendants. Then, it was necessary to implement an author algorithm to correctly separate each group of verb and its complement. The logic of this author algorithm can be seen in Figure 4.8.

The step of to **use an algorithm written by the author to separate each pair of verb and its complement**, in Figure 4.7 is explained with more details by the activity diagram in Figure 4.8. The sentence splitting process represented by Figure 4.8 starts with an activity node remembering that "for each verb node found in the dependency graph" the following actions are performed:

- To get the index of the verb word considering the whole sentence;
- To get the verb lemma;
- To Get the boolean information about negative words presence.
- Get the next word after the verb. With the verb plus its next word create a String parameter called *searchString*.

And finally to "add these information to an object *WordNode*". With that, it's possible to understand that for each verb node that was passed through it's created and filled a *WordNode* object, and this object is added to a list of *WordNode* objects. The following diagram node says to "get the list of WordNode objects and a list of *Strings* containing all the words in the sentence", because these are the arguments for the method that splits the sentence.

The method that splits the sentence gets both arguments, the list of *WordNode* objects and a *String* with all words in the initial sentence. Then the logic implemented is basically as says the next diagram node: "considering the position of each verb, go through the whole sentence and get each verb plus all the words before the next verb until the end of the main sentence". To do this, it was used classic operations with Strings in Java, using the information of the verb word and the previous word in the sentence from *WordNode* objects it's possible to get the correct ranges. After this operation the verb is replaced by its lemma, using the information contained in the *WordNode* object.

"By the end of the list of partial sentence built, it's passed to another method that checks if every partial sentence has a (dobj) node that is the complement of the verb", as it's possible to see in Figure 4.8. Following this diagram, after the decision gateway, for positive cases when the partial sentence has a node with the "dobj" relation it does nothing else with this partial sentence.

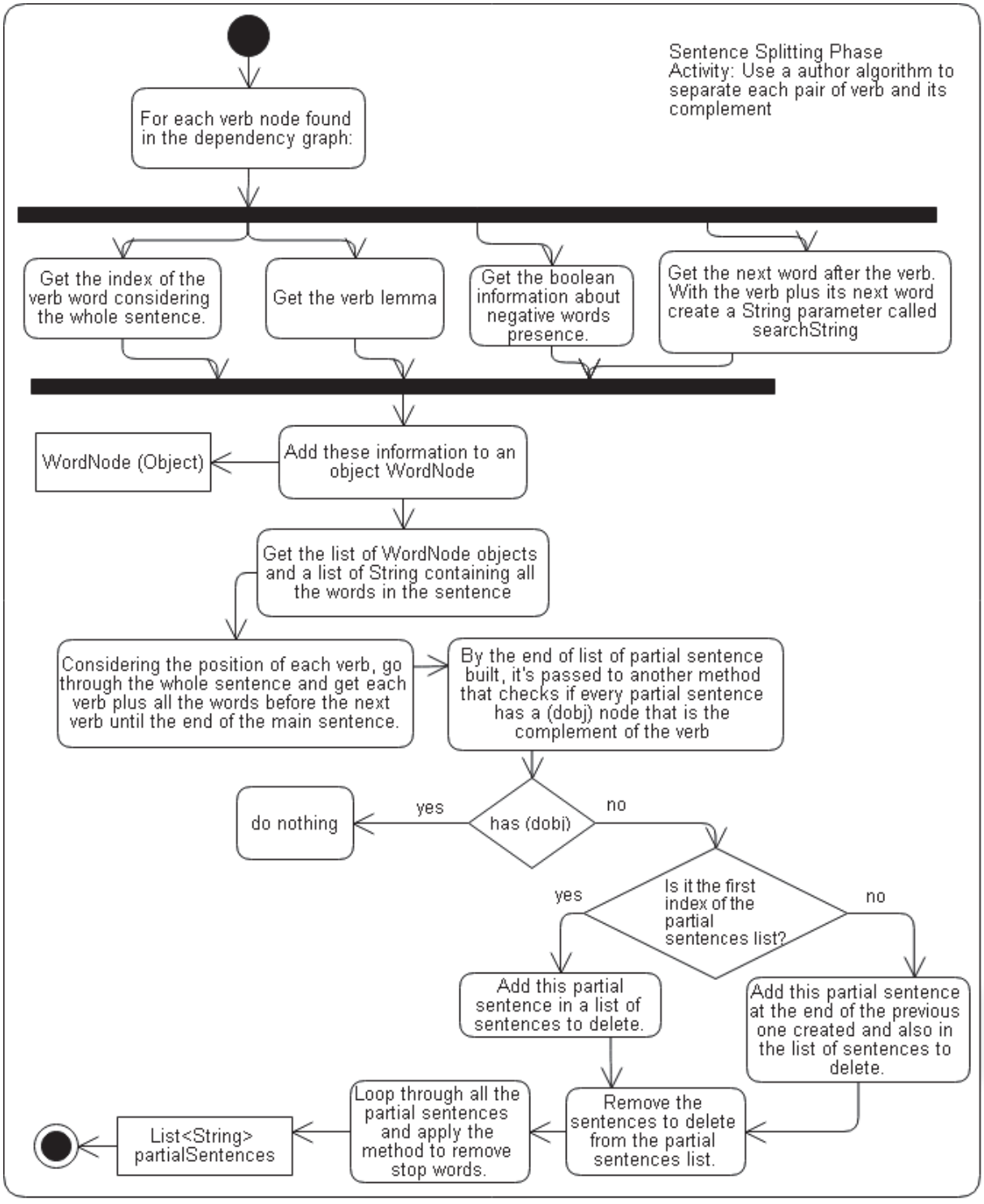



Figure 4.8: Second Pipeline - Sentence Splitting Task Activity Diagram

But, in case of not finding a node with the "dobj" relation, it's possible to see that there is another decision gateway which checks if "it is the first index of the partial sentences list". If yes, the action is just "add this partial sentence in a list of sentences to delete". Or else, in negative case the action is: "add this partial sentence at the end of the previous one created and also in the list of sentences to delete", as it's possible to see in Figure 4.8.

The next action which appears in Figure 4.8 is to "remove the sentences to delete from the partial sentences list". This action ends the process of splitting sentences and the next actions

are just to “loop through all the partial sentences and apply the method to remove stop words”. In this case, *stop words* means: conjunctions, punctuation and other non meaningful words at the end of the String. Thus, the result of this method is a list of partial sentences where each one is compounded by a verb plus all the words that are part of its complement.

The goal of these last five tasks is to give the last processing over the partial sentences generated, to eliminate the ones that have a verb but no object connected with the verb if the given partial sentence is the first piece of the original sentence. Otherwise, to put this partial sentence at the end of the previous one created, because it’s probably the case of a verb complement not correctly classified by the dependency graph. All the *WordNode* objects used for sentences splitting comes from the root nodes of the dependency graph. If the the dependency graph created from the Stanford CoreNLP library has some errors in the definition of the root nodes, these tasks can mitigate those errors while using the information dependency graph to split a sentence.

With this, the flow of sentences splitting process ends and the partial sentences goes to main diagram in Figure 4.7, and the next task according the diagram node is to “for Each partial Sentence, repeat the 3 following steps”:

- To “confirm the verb and find the verb object (system entity) using Chunking technique”.
- To “use the user/actor name and the partial sentence to build the user story description”. In this case the description is generated with the template: “As a [user/actor] I want to be able to [partial sentence]”, for positive sentences or for negative sentences: As a [user/actor] I should not be able to [partial sentence];
- To “use the lemma of the verb (verb in the infinitive form) plus the system entity to generate the user story title”.

The result of the two previous tasks is that for each partial sentence is created one user story object completely filled, with title and description, plus the system entity and actor/user already filled at the beginning of the flow. Then, the flow of multiple verb sentences ends by producing a list of user stories and the elements of this list are added to the final list of user stories that is returned at the end of the Second Pipeline.

Now, it’s discoursed about the third flow: multiple verbs plus passive voice sentence. To fall into this flow Figure 4.7 presents a gateway of decision, asking whether the sentence is passive or not to decide the path to follow. To check the presence of the passive voice in a sentence, it’s used the technique of to search for a node with the relation equals to “nsubjpass” in the dependency graph, this node means that the sentence has a passive subject, then it is classified as passive voice.

The third flow starts with the task of to **extract root verb from dependency graph**. This task is something common among the 3 flows, and again, it was done via navigation over the dependency graph structure to identify the root node containing the verb.

The third flow follows with the task of to **extract the passive agent, the user story’s user/actor**. According with the Stanford typed dependencies manual: “An agent is the complement of a passive verb which is introduced by the preposition “by” and does the action” De Marneffe e Manning (2008). The extraction of passive agent is also done going through the dependency graph and searching for a node with a relation equals to “nmod:agent”. Just to illustrate how it works, a piece of code used to do this:

```
1 if (edge.getRelation().toString().equals("nmod:agent")) {
2   List<Pair<GrammaticalRelation, IndexedWord>> pairs =
3   parse.childPairs(edge.getDependent());
4
```

```
5     if (!pairs.isEmpty()) {
6        pairs.forEach(p -> {
7        if(!p.second().word().equals("by"))
8           subjectBuilder.append(p.second().word()).append(" ");
9        });
10    }
11    subjectBuilder.append(edge.getDependent().word());
12 }
```

An alternative case for finding the passive agent happens when it's located below the node of the object of the verb. This situation can occur when the agent of the passive voice sentence is not introduced using the preposition "by", and it was solved with the following code:

```
1  if (edge.getRelation().toString().equals("nsubjpass")) {
2     passiveVoiceEntity = edge.getDependent().word();
3     List<Pair<GrammaticalRelation, IndexedWord>> subjPairs =
4
5     parse.childPairs(edge.getDependent());
6     subjPairs.forEach(s -> {
7        if (s.first.toString().equals("amod") ||
8
9        s.first.toString().equals("nmod:poss")) {
10          subjectBuilder.append(s.second().word()).append(" ");
11       }
12    });
13 }
```

The code above tests in its first line if the current edge relation is equals to "nsubject", which is considered the object of the verb and also the system entity in the context of the user stories. In the second line the value of this node is assigned to a variable which will hold the system entity. In the third line the list called "subjPairs is created, using the instance of the dependency graph named as "parse", the function "childPairs" is called to get the leaves below the current node. Then, in the line 6 of the code above, a loop starts for each internal leaf of the list to verify if the relation is equals to "amod" or "nmod:poss", and if so, the word in this node is appended to the *StringBuilder* object called "subjectBuilder" which holds the value of the subject found. This little verification has enabled to get the passive agent of sentences which didn't mention exactly the agent but mentioned two possessive pronouns separated by slashes: "his/her", because these tokens together are classified as "amod". The other possible situation is a single possessive pronoun, "his", for example, which is identified as "nmod:poss" and correctly identified by the exposed code.

Then, the next task that appears in Figure 4.7, is to **check for negative expressions in the sentence**, and again, it was used the static list of negative words. We created this list of negative words to check for any match with the negative words and with this to classify the sentence as negative or positive if there was no match considering the words of the sentence in the range of the 3 words before the verb and one after it.

The next diagram activity in Figure 4.7 points to the task of to **extract the system entity**. In this case the extraction of the system entity is different from the active voice, now it's not the object of the verb that should be found, but the passive subject. To find the system entity, it is used the technique of going through the dependency graph and searching for a root with the relation equals to "nsubjpass".

Just to exemplify, this code below is used to check for the passive subject, that is our system entity:

```
1  if (edge.getRelation().toString().equals("nsubjpass")) { ... }
```

After extracting the system entity, the following task demonstrated in Figure 4.7 is to **extract any relation/information of time in the passive sentence complement**. As the structure of the passive voice sentences is completely different from the active voice ones that are only splice to get the complement, the task of identifying the complement is implemented based on getting some specific and common items used in the grammatical structure as complement, for example, words that express relation of time or duration. Then, to search for this type of phrasal complement, the dependency graph is navigated to search for a node with a relation equals to "nmod:tmod".

Then it comes the task of to "verify the presence of a word with coordination relation (e.g: and), and if finding it, repeat the 5 previous steps to create information about another user story". According to De Marneffe e Manning (2008), "A coordination is the relation between an element of a conjunct and the coordinating conjunction word of the conjunct, [...] this is also called as "cc", and dependent on the root predicate of the sentence". A coordination can indicate the begin of other action description in the text. For example, the usage of an element with coordination relation can be seen in the sentence: "Her work can be saved to her account and shared with other users". In this example, the word "and" has the relation equals to "cc" (coordination).

If a word of coordination is found, then the same 5 previous tasks are executed going through the remaining nodes of the dependency graph in a separated loop to grab all the information related to the other action, which derives other user story.

And finally, Figure 4.7 points that the next step is to execute those 2 common tasks that are used for all the flows:

- To **use the user/actor name and the partial sentence to build the user story description**.

- To **use the lemma of the verb (verb in the infinitive form) plus the system entity to generate the user story title**.

In this case, for the passive voice, the description is generated with the template: "As a [user/actor] I want to be able to [partial sentence]", for positive sentences or for negative sentences: As a [user/actor] I should not be able to [partial sentence]. And the partial sentence for passive voice is the result of the concatenation of the verb plus the system entity plus the complement. In the end of these steps it's possible to see that the *UserStory* object was completely filled, and that's how it ends the flow of multiple verbs plus passive voice sentences.

Just to exemplify what happens with a passive voice sentence that passes throughout the second pipeline, imagine that it is the initial sentence that passed through the First Pipeline and at the start of the Second Pipeline was classified as a passive voice sentence:

- The salaries will be updated by the Payroll Administrator every year.

In the first task the root verb found in the dependency graph for that sentence is the verb **updated**. After the second task the user/actor recognized is **the Payroll Administrator**. When evaluated if it's a negative sentence in the third task, it was verified that it's a positive sentence. The system entity recognized in the fourth task is **the salaries**. And during the fifth task, it is extracted the complement: **every year**. Then the result in the end of the flow was the *UserStory* object with the following content:

- Title: Update the salaries;

- Description: As the Payroll Administrator I want to be able to update salaries every year.
- User/Actor: The Payroll Administrator
- System entity: salaries;
- Main action: update.

At the end of the Second pipeline, after all the sentences have been passed through the algorithm that transforms them into *UserStory* objects, all the *UserStory* objects are added in a list which is returned to the previous level: the *UserStoryService* class. And, the *UserStoryService* class returns the list of *UserStory* objects to the Controller class which returns this list to the User Interface using the *JSON* format.

In the next section, more examples are shown about the sentences transformation into *UserStory* objects, using sentences that activate the others two flows of the algorithm and that were used during the tool development.

## 4.9 Pipelines inputs and outputs

During the Pipelines execution the input sentences suffer changes in their structures. These changes start in the First Pipeline and they get more complex in the Second Pipeline until to generate the *UserStory* objects.

It was explained in the previous section how the pipelines tasks work, what they change in the sentences, how they can extract and manipulate the information to build the text of the user story title and description. In this section, we take a deep look in every input and output of each pipeline task.

To exemplify the process that occurs with sentences we use some sentences extracted from one IBM white paper called *Payroll System*:

- *"The Payroll Administrator maintains employee information. The Payroll Administrator is responsible for adding new employees, deleting employees and changing all employee information such as name, address, and payment classification (hourly, salaried, commissioned), as well as running administrative reports. "*

The first tasks that run in the first pipeline generate the dependency graph and a list of mentions for the text above. After that, it is searched in the text if there's any subject which uses a pronoun or any other word to mention the root name of the subject. As these two sentences just use the same way to mention the subject, the algorithm keeps with the same words and the dependency graph generated in the beginning.

Just to have an idea about how is the dependency graph information returned from these sentences, follows the graphic illustration of the dependency graph in Figure 4.9:

Figure 4.10 shows the programming version of the dependency graph from the first sentence, in the way it's generated to be used in the First Pipeline:

This sentence, which contains only one verb will pass in the first flow:

- *"The Payroll Administrator maintains employee information."*

In the first flow, the first task extracts the user/actor from the first sentence by searching for a node with "nsubj" in the dependency graph and navigating in their leaves to get all the words inside this node. Then the user/actor extracted is: **the Payroll Administrator**.

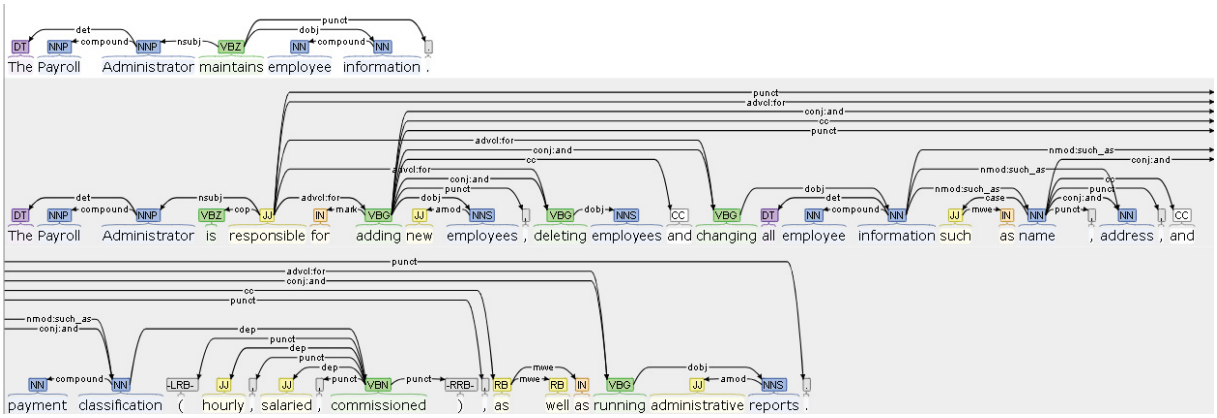



Figure 4.9: Dependency Graph of Payroll Administrator text

```
(ROOT (S (NP (DT The) (NNP Payroll) (NNP Administrator)) (VP (VBZ maintains) (NP (NN employee) (NN information))) (. .)))

-> maintains/VBZ (root)
  -> Administrator/NNP (nsubj)
    -> The/DT (det)
    -> Payroll/NNP (compound)
  -> information/NN (dobj)
    -> employee/NN (compound)
  -> ./. (punct)
```

Figure 4.10: Dependency Graph of the first sentence

After that, it's checked for negative words, and no one is found. Then, it's extracted the verb **maintains**, using the *chunking* technique.

In the following task, it's created the partial sentence, getting the part of the sentence that starts with the verb and get all the words until the end of the sentence. And in the next task, the verb is replaced inside the partial sentence by its *lemma*, which is the infinitive form of the verb. And, finally, it's eliminated the punctuation at the end of the partial sentence.

In the next task, it's confirmed the current verb, that now is a verb in the infinitive form. And also the system entity is extracted from the partial sentence. Then it's identified that the main action is the word **maintain** and the system entity is equals to **employee information**.

The following task is to generate the user story description, using the pattern "As a [user/actor] I want to be able to [partial sentence]". And the description generated is: **As the Payroll Administrator I want to be able to maintain employee information**. The title is generated using the lemma of the verb plus the system entity, what results in: **maintain employee information**. Then, with all the information filled inside the *UserStory* object this flow ends and the loop follows with the next sentence starting again in the beginning of Figure 4.7.

The second sentence has multiple verbs and active voice, then it follows in the second flow. Figure 4.11 shows the dependency graph generated for the second sentence, that is used for most of the tasks in the Second Pipeline.

The first thing that happens in the second flow with the second sentence is to **extract the subject or compound subject via dependency graph**. In this step the subject which is our user/actor of our user story is recognized as **The Payroll Administrator**.

In the second task of the second flow, the dependency graph that appears in Figure 4.11 is navigated into its root nodes. And in the next task, it's verified the presence of negative words, but, no negative word is found.

Then the author algorithm which splits the sentence is executed and builds the partial sentences. One of the partial sentences generated is: *"I can be responsible for"*. This part of the sentence was actually connecting the enumeration of actions in the initial sentence, and after

```
(ROOT (S (NP (DT The) (NNP Payroll) (NNP Administrator)) (VP (VBZ is) (ADJP (JJ responsible) (PP (IN for)
        (S (VP (VP (VP (VBG adding) (NP (JJ new) (NNS employees))) (, ,)
        (VP (VBG deleting) (NP (NNS employees))) (CC and)
        (VP (VBG changing) (NP (NP (DT all) (NN employee) (NN information)) (PP (JJ such) (IN as)
                (NP (NN title) (, ,) (NN address) (, ,) (CC and) (NN payment) (NN classification)))
                (PRN (-LRB- -LRB-) (NP (NP (JJ hourly)) (, ,) (NP (NP (JJ salaried)) (, ,) (VP (VBN commissioned)))) (-RRB- -RRB-)))))
                (, ,) (CONJP (RB as) (RB well) (IN as))
        (VP (VBG running) (NP (JJ administrative) (NNS reports)))))))) (. .)))

-> responsible/JJ (root)
  -> Administrator/NNP (nsubj)
    -> The/DT (det)
    -> Payroll/NNP (compound)
  -> is/VBZ (cop)
  -> adding/VBG (advcl)
    -> for/IN (mark)
    -> employees/NNS (dobj)
      -> new/JJ (amod)
    -> ,/, (punct)
    -> deleting/VBG (conj:and)
      -> employees/NNS (dobj)
    -> and/CC (cc)
    -> changing/VBG (conj:and)
      -> information/NN (dobj)
        -> all/DT (det)
        -> employee/NN (compound)
        -> address/NN (nmod:such_as)
          -> such/JJ (case)
            -> as/IN (mwe)
          -> title/NN (compound)
          -> ,/, (punct)
          -> ,/, (punct)
          -> and/CC (cc)
          -> classification/NN (conj:and)
            -> payment/NN (compound)
        -> classification/NN (nmod:such_as)
        -> hourly/JJ (dep)
          -> -LRB-/-LRB- (punct)
          -> ,/, (punct)
          -> salaried/JJ (appos)
            -> ,/, (punct)
            -> commissioned/VBN (acl)
          -> -RRB-/-RRB- (punct)
    -> ,/, (punct)
    -> as/RB (cc)
      -> well/RB (mwe)
      -> as/IN (mwe)
    -> running/VBG (conj:and)
      -> reports/NNS (dobj)
        -> administrative/JJ (amod)
  -> deleting/VBG (advcl)
  -> changing/VBG (advcl)
  -> running/VBG (advcl)
  -> ./. (punct)
```

Figure 4.11: Dependency Graph of the second sentence

passing through the task of to **validate that partial sentence has "dobj" otherwise eliminate it**, it is eliminated.

The partial sentences that contain a node with a relation equals to “dobj” are kept in the pipeline, and they are:

- *"I can add new employees."*
- *"I can delete employees."*
- *"I can change all employee information such as name, address, and payment classification (hourly, salaried, commissioned)."*
- *"I can run administrative reports."*

This static piece of text: **I can**, was added to the sentence just to make them have a structure for helping to extract the main action and the system entity.

Then, the next task is to loop through the partial sentences and for each one repeating the steps:

- Confirm the verb and find the verb object (system entity) using chunking technique;
- Use the user/actor name and the partial sentence to build the user story description
- Use the lemma of the verb plus the system entity to generate the user story title.

And the result is that each partial sentence derives a user story object with the information as follows:

1. **User story tile**: add new employees.

   **User story description**: As the Payroll Administrator I want to be able to add new employees.

   **User/Actor**: the Payroll Administrator.

   **Main action**: add.

   **System Entity**: employees.

2. **User story tile**: delete employees.

   **User story description**: As the Payroll Administrator I want to be able to delete employees.

   **User/Actor**: the Payroll Administrator.

   **Main action**: delete.

   **System Entity**: employees.

3. **User story tile**: change employee information.

   **User story description**: As the Payroll Administrator I want to be able to change all employee information such as title, address, and payment classification (hourly, salaried, commissioned).

   **User/Actor**: the Payroll Administrator.

   **Main action**: change.

   **System Entity**: employees.

4. **User story tile**: run administrative reports.

   **User story description**: As the Payroll Administrator I want to be able to run administrative reports.

   **User/Actor**: the Payroll Administrator.

   **Main action**: change.

   **System Entity**: administrative reports.

After the generation of all these *UserStory* objects, they are added in a list that is the result of the Second Pipeline, and passed to the upper levels until to be sent in the *JSON* format to be shown in the User Interface as demonstrated in Figure 4.12 below.

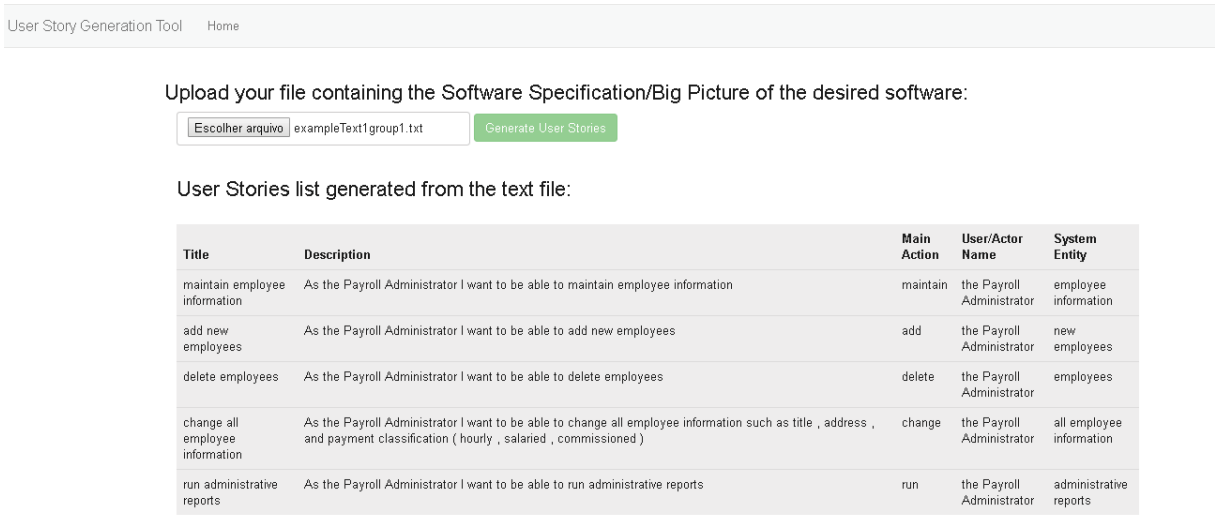



Figure 4.12: Demonstration Result Screen

## 4.10 Conclusion

In this chapter we presented UserStoryGen approach and tool. During each section it was shown the proposed solution using top-down approach, that introduced first the definition of the UserStoryGen approach, design goals, challenges and limitations, then the System Design and architecture, and finally went deep in the proposed solution, showing the Implementation Details. It was explained the decisions for using an architecture with pipelines and all the tasks that happens inside each pipeline. It was explained the inputs and outputs of each task inside the pipelines until to generate the user story information for each sentence that passes through the natural language processing phase. This chapter approached how it was treated every complex task in the NLP context to work with different formats of sentences, active and passive voice, negative sentences, splitting sentences with multiple verbs and other tasks necessary to extract information to generate the list of user story objects that is the final output from this work. The example texts used during the development of UserStoryGen demonstrated that this tool can handle the 3 flows explained: single verb sentences, multiple verbs sentences with active voice and multiple verbs sentences with passive voice.

# 5 Evaluation of UserStoryGen Tool

During the previous chapter it was explained how the UserStoryGen tool was built and every logic involved in how it works. In this chapter, we'll explain the evaluation of the UserStoryGen. In the following sections, we'll present the methodology used to evaluate the UserStoryGen tool, the case studies and their results. The results produced by the natural language processing done within this work will be visually analyzed, and the methodology to calculate the rates will be explained. We'll show the rates of success and failure among different types of texts used during the validation. Lastly, after all the case studies made, the obtained results will be discussed.

## 5.1 Methodology

This section outlines the proposed methodology to evaluate the efficiency, accuracy, and performance of the UserStoryGen tool. The methodology implies in using three groups of texts from different sources, and for each group of texts, the case studies results are measured.

The first group of input sources is a group dedicated to book and white paper texts, which contain a software description, or a big picture information about a desired software, or even to be more simplistic, these texts present a brief idea of a software and what is expected to find in it in terms of functionalities.

The second group of texts contains the ones collected from people who work in software engineering area, like developers and system analysts that collaborated as specialists, and the texts that they gave to help in testing is basically:

- Short description/big picture of a desired software as input.
- Big picture or textual description of a desired software, in a text that should have at least 12 lines.

Every text given by the specialists also has a list of expected user stories in their vision.

The third group of texts has texts provided by a company that shared the information with terms of secrecy, preventing the disclosure of the shared texts, as they contain the company real data about a software which is a product of this company. These texts help in having real data from the industry. The texts provided by the company are:

- A collection of introduction texts extracted from user journey documents.
- Description of system features, which was written by a Product Owner. System features are described by Leffingwell (2017) from SAFe © team as: **A Feature is a service that fulfills a stakeholder need.**[1]

[1]SAFe and Scaled Agile Framework are registered trademarks of Scaled Agile Inc. Reproduced with permission from © 2011-2017 Scaled Agile, Inc. All rights reserved.

To evaluate the results, the company also shared a list of user stories that were done within this product. The purpose of the UserStoryGen evaluation is to measure and have a numerical idea about how efficient this tool is. Then, as these texts are under terms of secrecy, this group of texts will have only their rates and numbers from the case studies results exposed.

Before starting the case studies, it was expected a efficiency rate above 70% as the tests during the UserStoryGen development were showing good results, and the algorithm written by this author was also based on many different input texts that were formulated by this author and also grabbed in books and white papers.

## 5.2 Example of evaluation

With the two pieces of the project up and running, the UserStoryGen was started to evaluate by adding the first input text on the screen. Using the button **Chose File** and selecting the "exampleText.txt" on the screen as shown in the Figure 5.1.

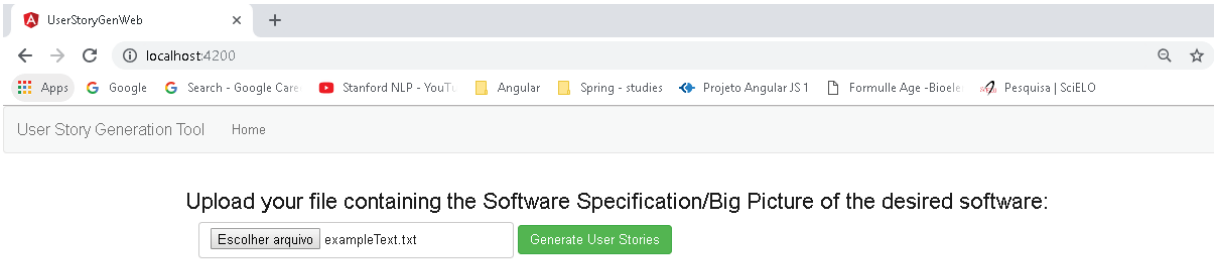



Figure 5.1: Group 1 first case study - proposed system home page

The file "exampleText.txt" contains the following text, from an IBM white paper called **Payroll System**:

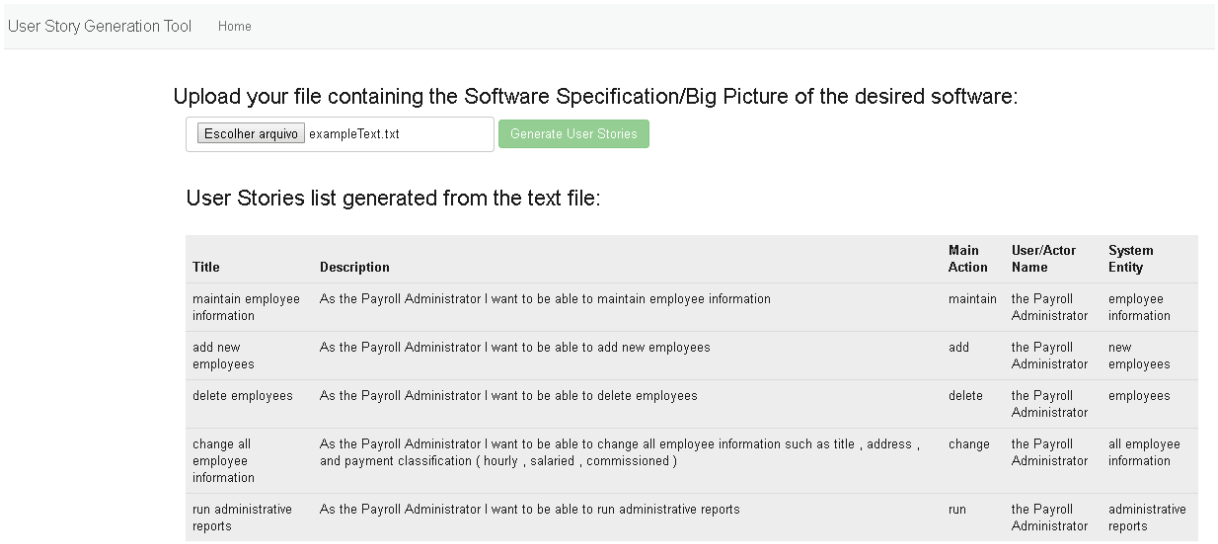
- The Payroll Administrator maintains employee information. The Payroll Administrator is responsible for adding new employees, deleting employees and changing all employee information such as title, address, and payment classification (hourly, salaried, commissioned), as well as running administrative reports.

Then the **Generate User Stories** button was pressed, and after the processing of the text by the Back-end of the project, the result was displayed on the screen as shown in the Figure 5.2.

The result shown in the Figure 5.2 contains a table with a list of 5 user stories built from the input text. As demonstrated, for each relevant verb found in the **big picture** text, it was created a user story. In the table of user stories in the Figure 5.2, it was generated a user story for the first paragraph, and for the second sentence which originally had 5 verbs, it was generated 4 more user stories.

For the first sentence it was generated one user story as expected. According to the author work experience in the industry, it's normal to have big user stories with a generic title and description which group other users stories, as is the case of the user story generated for the first sentence. Its content is generic and the user stories generated in the sequence can be considered as children of this one.

Going deeper in the result of the second sentence of the input text, it's possible to conclude that the 5 verbs in the this sentence are: "is", "adding", "deleting", "changing" and




| Title | Description | Main Action | User/Actor Name | System Entity |
|---|---|---|---|---|
| maintain employee information | As the Payroll Administrator I want to be able to maintain employee information | maintain | the Payroll Administrator | employee information |
| add new employees | As the Payroll Administrator I want to be able to add new employees | add | the Payroll Administrator | new employees |
| delete employees | As the Payroll Administrator I want to be able to delete employees | delete | the Payroll Administrator | employees |
| change all employee information | As the Payroll Administrator I want to be able to change all employee information such as title , address , and payment classification ( hourly , salaried , commissioned ) | change | the Payroll Administrator | all employee information |
| run administrative reports | As the Payroll Administrator I want to be able to run administrative reports | run | the Payroll Administrator | administrative reports |

Figure 5.2: Group 1 first case study - result screen

“running”. The first verb: “is” was just connecting the subject with the enumeration of possible actions that this subject should be able to do in the described system. Because of that, the piece "is responsible for" was considered as a connection element to initiate actions enumeration, and as this partial sentence doesn’t have an object connected with the verb “is”, then this piece of text was discarded and counted as a true negative example.

The other verbs in the second sentence have their object and complement, and they’re connected to the same subject, who is **the Payroll Administrator**, what resulted in valid partial sentences which originated 4 user stories. For all the 4 verbs that originated the lasts user stories, it can be noticed that the original verbs in the gerund form were replaced by their *lemma* word (infinitive or root form), in the process of creation for either the title and description of the user stories as well as the main action attribute which is set exactly to this value.

The system entities extracted from the text for each user story are all correct, and they represent the object directly connected with the given verb. The main action that is the identified relevant verb lemma plus the system entity are building the attribute title in each user story, and as it’s possible to see in the Figure 5.2 all the results are correct.

To evaluate the result produced by the UserStoryGen tool, a list of expected user stories was generated, also considering the document that resolves the Payroll System problem that shows the identified use cases and the use case considering only that piece of text. For that piece of text, the IBM document contains two use cases as the answer, the first one have 3 alternative flows and its title is: **maintain employee information**. The other use case is **Create Administrative Report**, but using the text words it should be **run administrative reports**.

Even if user stories and use cases are not the same, they reflect what should be delivered to the user in different ways. The idea of what is a value for the user can be captured from a use case and transformed in one or multiple user stories. And, this is what the author of this work did, while defining the list of expected user stories for that piece of text.

Considering the list of expected stories created by this author, it was noticed that for this short text of two paragraphs the hit success was of 100%. The UserStoryGen has identified and created 5 correct user stories that are present in the IBM result document. The answer of IBM is exactly the 4 user stories generated through the second sentence, so these user stories represent the true positives, and the partial sentence that was eliminated during the processing and didn’t originate a user story was considered as a true negative example.

## 5.3 Case studies

The case studies were done separately for each of the 3 groups described in the previous section. In this section and its subsections we explain the case studies of each group and for the ones that it's possible, we show the inputs and outputs for further discussion about the results in the next section.

### 5.3.1 First Case Studies Group: texts from books and white papers

The first group of case studies were done using academic texts as inputs, these texts were basically texts found in a book and a white paper.

The first case study was done using the whole Payroll System Problem text produced by IBM. To measure the result, a list of expected user stories was created considering the all the use cases given by IBM as the result for the problem, but only the use cases that can be manually found in the text, and also converting their ideas in terms of user values from use cases to user stories. The list of expected user stories was created with user stories titles and descriptions using the same words found in the text, differently from what was seen in the IBM answer that changed the words of the text that have a user perspective and put in a technical perspective.

The first case study text contains 38 lines and 600 words, what makes the processing very hard, as the sentences have some ambiguity issues and multiple relations between them. These factors do not influence much on the processing time, but in the accuracy of the result. The results of the first case study from the first group of case studies will be shown and discussed in the next section.

The answer to the Payroll System Problem are the following use case tiles and sub-flows titles:

- Create Administrative Report. In the words of the text: "Run Administrative reports".
- Create Employee Report.
- Login
- Maintain Employee Information. This use case has a basic flow that includes three sub items that in the world of user stories would be considered as separated user stories:
  - Add an Employee
  - Update an Employee
  - Delete an Employee
- Maintain Purchase Order. The basic flow includes three sub items and each of them can be considered as user stories.
  - Create a Purchase Order
  - Update a Purchase Order
  - Delete a Purchase Order
- Maintain Timecard
  - Basic Flow: "This use case starts when the Employee wishes to enter hours worked into his current timecard."

  - Submit Timecard

- Run Payroll
- Select Payment Method

The sub-flows described in the Payroll System Problem answer that are defined as Alternative Flows and that are not mentioned in the problem text was not considered to assert the answer of the UserStoryGen tool, only the flows that would be possible to infer reading the text were considered. Hence, two items were not considered:

- Login.
- Delete a Purchase Order.

Another important rule used to make the assertions against the expected answer and the result given by the UserStoryGen was to create a different expected user stories list using the same words found in the text and also considering some user stories that are not present in the definition of use cases, of course because use cases are different from user stories. The main difference between use cases and user stories is that user stories aim to separate each task, and create a user story for it. Then, the list of user stories consider the tasks of the users, which can be manually inferred from the text.

This is the user stories title list used to test the UserStoryGen results:

- Run administrative reports
- Have employee report
- Maintains employee information:
  - Add new employees,
  - Delete employees
  - Change all employee information
- Enter purchase orders
- Can only access and edit my own time cards and purchase orders
- Submit purchase orders
- Record time card
- Change employee preferences
- Submit time cards
- Run automatically payroll
- Choose employee method of payment
- Pay each employee
- Pay the employee 1.5 times his or her normal rate for those extra hours

- Pay hourly workers every Friday.
- Tell what date the employees are to be paid
- Receive a commission
- Have a Windows-based desktop interface
- Have a web-based interface
- Determine commission rate for each employee
- Query totals
- Create various reports
- Work with the existing Project Management Database
- Access but not update information stored in the Project Management Database

The second case study of the first case studies group used a big picture text extracted from the Rasmusson (2010), The Agile Samurai book presents a big picture text and also the user stories extracted from this text. The previous case study did not have the expected output in terms of user stories and it was needed to make a conversion between the expected result document and the desired format, but this third case study has the perfect data mass: a big picture text and the user stories generated through it.

This is the text from Rasmusson (2010):

> "First, I want the website to be a place for the local scene. Somewhere the kids can come and check out upcoming events—surf competitions, lessons, things like that.
>
> Second, I need a place to sell merchandise. Boards, wet suits, clothes, videos, and things like that. But it's gotta be easy to use and look really good.
>
> Third, I've always wanted a webcam pointing at the beach. This way, you don't have to come down to check out the conditions. You can just open your laptop, go to the website, and see whether it's worth getting up. This also means the website has to be fast."

The answer for this big picture text present in The Agile Samurai Book, from Rasmusson (2010), is this list of user stories:

- List upcoming sales. In the text words: **sell merchandise**.
- Display local surf report
- List upcoming events and competitions
- View webcam of beach
- Show lesson packages and rates
- Used board and equipment section
- Website must be super-fast / Real-time updates

- Design should look really good

After mentioning this list, Rasmusson (2010) says that the user story: **website must be super-fast** and the user story **design should look really good** can appear a little vague and ambiguous, but they are still user stories.

> "Stories like these, we call constraints. They aren't your typical user stories that we can deliver in a week. But they are important because they describe characteristics our customers would like to see in their software. Sometimes, we'll be able to translate these into testable stories." Rasmusson (2010)

### 5.3.2 Second Case Studies Group: texts from Software Engineer Specialists

In the second group of case studies it was used 2 texts provided by women that work as Development Tech Leader and Test Engineer, both with a great experience in the Software Engineer area and different backgrounds. Their different perspective in software development area helps on having different text styles and check if the UserStoryGen algorithms are capable to generalize and produce good results with different input texts.

The specialists also provided the answer for the texts, that is, the user stories that are expected as a perfect output to the given texts. Basically, this group of case studies has the case study one and two using texts that represent the **big picture** idea of two desired software.

The text of the first case study was written by a Development Tech Leader that has a long experience working with user stories in agile teams. She's graduated in Computer Science, and who has 11 years of experience in the development area. This is the first case study's text:

> " The user of the application is able to track his/her performance when running or riding his/her bike via the GPS.
>
> His/her performance can be saved to his/her account and shared with other friends from his/her social networks.
>
> The user cannot delete any entries, once they are saved to the account. The user has the ability to create a report with all the activities by date range, or by type (running or biking)."

The expected user stories generated from this text of the second case study will be:

- Track his/her performance
- Save performance
- Share performance
- Cannot delete any entries
- Create a report

The text of the first case study from the second case studies group contains some key points of difficulties to be processed, such as: multiples verbs in the sentences, one negative sentence and also a passive voice sentence which has two important verbs for the user stories generation. And even with these factors, the UserStoryGen tool has produced a great result, as displayed in Figure 5.3.

The box bellow contains the result in JSON format that is possible to download from the UserStoryGen tool, so that each item can be visualized in the way that it's coming from the Rest API.

User Stories list generated from the text file:

| Title | Description | Main Action | User/Actor Name | System Entity |
|---|---|---|---|---|
| track his/her bike | As the application user I want to be able to track his/her performance when running or riding his/her bike via the GPS | track | the application user | his/her bike |
| share performance | As His/her I want to be able to share performance with other friends from his/her social networks | share | His/her | performance |
| save performance | As His/her I want to be able to save performance | save | His/her | performance |
| Can not delete any entries | As the user I should not be able to delete any entries , once they are save to the account | delete | the user | any entries |
| create a report | As the user I want to be able to create a report with all the activities by date range , or by type ( run or biking ) | create | the user | a report |

Figure 5.3: Result of the first case study from the second group of case studies


```
[  {
      "title": "track his/her bike",
      "description": "As the application user I want to be able to track
       his/her performance when running or riding his/her bike via the GPS",
      "systemEntity": {
         "name": "his/her bike"
      },
      "mainAction": "track",
      "user": {
         "name": "the application user"
      }
   },
   {
      "title": "share performance",
      "description": "As His/her I want to be able to share performance with
       other friends from his/her social networks",
      "systemEntity": {
         "name": "performance"
      },
      "mainAction": "share",
      "user": {
         "name": "His/her"
      }
   },
   {
      "title": "save performance",
      "description": "As His/her I want to be able to save performance",
      "systemEntity": {
         "name": "performance"
      },
      "mainAction": "save",
      "user": {
         "name": "His/her "
      }
   },
   {
      "title": "Can not delete any entries",
      "description": "As the user I should not be able to delete any
       entries , once they are save to the account",
      "systemEntity": {
         "name": "any entries"
```

```
        },
        "mainAction": "delete",
        "user": {
           "name": "the user"
        }
    },
    {
        "title": "create a report",
        "description": "As the user I want to be able to create a report
         with all the activities by date range , or by type ( run or biking )",
        "systemEntity": {
           "name": "a report"
        },
        "mainAction": "create",
        "user": {
           "name": "the user"
        }
    }]
```


As shown in Figure 5.3 and in the **JSON** file content, all the expected user stories were successfully generated. The result presented a perfect match between the expected user stories titles, and the descriptions are also good. Even for the passive voice sentence, the user stories were correctly generated, the sentence was correctly splice, the **mainAction** attribute holds the correct verb, the **systemEntity** attribute is correct and only the **user** attribute was not so great, because **his/her** possessive pronoun could not be recognized as the same user extracted for the others user stories.

In this example of output it's possible to notice that the technique of using the root node verbs to help on sentence splitting really produces a good result. The first sentence of this text which contains the verbs: **is**, **able**, **track**, **running** and **riding**, but the root node verb detected and validated was **track**. This was only possible because the structure of the pipeline validates several conditions that were explained in the previous chapter, and one of them is that even if more than one verb is recognized as a root node, the condition of having an object directly associated with the given verb helps to eliminate false positive elements. The first sentence: " The user of the application is able to track his/her performance when running or riding his/her bike via the GPS " has two verbs with complement: **track** and **riding**. The second verb and its complement can be considered as a partial sentence that acts as a true negative element, that in this case were correctly classified and didn't generate a user story as expected.

The second text of the second group was written by a Test Engineer, with graduation in Information Systems and 4 years of experience with Testing and 2 years of experience in the Test Automation field. This is the text of the second case study:

> " The bakery system will maintain data from customers, products, service supplier, employees as well as generate administrative reports.
>
> The maintenance feature includes adding, deleting, and updating all customer, users and employees information that will be: name, address, date of birth and for employees payment classification too. And for the users of the bakery system, it will be saved the personal information, an id and password too.
>
> The products maintenance feature will be about to create product in the system with information such as name, value for sale, date of sale, value of purchase, quantity, value off when applicable. This feature will include, deleting, and updating the product information as well.

> The administrative reports will generate data from sales, crossing information with the date and value of sales. It will also include an alert function when items and products are bellow the minimum amount defined by administrative users.
>
> Finally, the system also provides a cash flow that will be the feature that provides information for administrative reports."

The expected user stories provided for this text, and their title and description are:

- Create product
- Update the product information
- Delete product
- Maintain data from service supplier.
- Add customer, users and employees information
- Delete customer, users and employees information
- Update customer, users and employees information
- Include an alert function
- Generate data from sales
- Provide a cash flow
- Generate administrative reports

The result screen after the test with the second text can be seen in Figure 5.4. This text presented several points of difficulties which impacted in an accuracy rate of 57,14% and both Precision, Recall F1 Measure rates of 70%.

The result shown in Figure 5.4 are discussed with more details in the next section.

### 5.3.3 Third Case Studies Group: real product texts provided by a company

The third group of case studies is compounded by a collection of introduction sections from User Journeys documents, *Product Feature* texts that are for sure the most important type of text provided by the company, and a commercial proposal text from the same product that belongs the **Product Features** texts.

As the company shared its data with secrecy terms, the data will not be shown in this work, but the numeric results of the case studies with enterprise data will be presented and discussed.

The first type of texts: introduction part of User Journey documents, are grouped in a single file and processed at once, as the introductions are short and each only basically refers to the usage of a **Product Feature**. Then, this type of text produces one file that is used in the first case study of the second case studies group.

The Second type of text: **Product Features** descriptions have 2 texts that are put together into a single file to be used in the second case study of the third group.

| Title | Description | Main Action | User/Actor Name | System Entity |
|---|---|---|---|---|
| maintain data | As the bakery system I want to be able to maintain data from customers , products , service supplier , employees | maintain | the bakery system | data |
| generate The administrative reports | As the bakery system I want to be able to generate The administrative reports | generate | the bakery system | The administrative reports |
| update all customer | As the maintenance feature I want to be able to update all customer , users and employees information that will be : name , address , date of birth and for employees payment classification | update | the maintenance feature | all customer |
| create product | As the products maintenance feature I want to be able to create product in the system with information such as name , value for sale , date of sale , value of purchase , quantity , value off when applicable | create | the products maintenance feature | product |
| update the product information | As the products maintenance feature I want to be able to update the product information | update | the products maintenance feature | the product information |
| generate data | As the administrative reports I want to be able to generate data from sales | generate | the administrative reports | data |
| information with the and value sales date | As the administrative reports I want to be able to cross information with the date and value of sales | information | the administrative reports | with the and value sales date |
| include a alert function | As the products maintenance feature I want to be able to include a alert function when items and products are | include | the products maintenance feature | a alert function |
| bellow the minimum amount | As the products maintenance feature I want to be able to bellow the minimum amount defined by administrative users | bellow | the products maintenance feature | the minimum amount |
| provide a cash flow | As the system I want to be able to provide a cash flow that will be the feature that provides information for The administrative reports | provide | the system | a cash flow |

Figure 5.4: Result of the second case study from the second group of case studies

The person who wrote the documents used in this group of case studies is a Development Team Leader and also plays the role of Product Owner inside an agile team, he's graduated in the computer area and has several years of experience in the software engineering area.

The results of each case study are compared with the user stories that this product had during its implementation and they will be also checked with the Product Owner of the project and the company from which the documents belong to.

## 5.4 Results

For better discussing the numbers about the UserStoryGen tool performance and to understand how it was measured, Table 5.1 gives more description about what was considered a true positive, false positive, true negative and false negative in the relation to each sentence and the user story produced from it or not. These measures will be used in the formula of accuracy and point to how good is the performance of the UserStoryGen.

Table 5.1: Classification of sentences that will correctly originate user stories

| Sentence/Partial Sentence | **Should Originate User Story** | **Shouldn't originate User Story** |
|---|---|---|
| **Originates a User Story** | True Positive (**TP**) Correct result | False Positive (**FP**) Incorrect result. |
| **Doesn't originate a User Story (Correctly)** | False Negative (**FN**) Incorrect Result | True Negative (**TN**) Correct Result |

Where:

- TP = The sentence or partial sentence processed by the UserStoryGen that originates a user story and which really should originate a user story.
- FP = The sentence or partial sentence that after processed by the UserStoryGen originates a user story, but, shouldn't do it.
- TN = The sentence or partial sentence processed by the UserStoryGen that doesn't originate a user story and this is the expected behavior.
- FN = When the sentence or partial sentence processed by the UserStoryGen doesn't originate a user story, but, it should do it.

Then, with the values for the each item, they are used in the accuracy Equation 5.1 to produce one of the metrics.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

Other metrics that were calculated are: Precision, Recall and based on these two it is calculated the F Measure.

The **Precision** rate is the percentage of selected items that are correct and it's calculated with Equation 5.2:

$$Precision(P) = \frac{TP}{TP + FP} \tag{5.2}$$

The **Recall** is the percentage of correct items that are selected, and it's calculated with Equation 5.3:

$$Recall(R) = \frac{TP}{TP + FN} \tag{5.3}$$

After calculating the result of the equation of **Precision** P (5.2) and **Recall** R (5.3), with the values of **P** and **R** it's possible to calculate the **F Measure**:

$$F = \frac{1}{\alpha\frac{1}{P} + (1.\alpha)\frac{1}{R}} = \frac{(\beta^2 + 1)PR}{\beta^2 P + R} \tag{5.4}$$

To calculate the F Measure it'll be used the balanced F1 measure formula 5.5, with $\beta$ = 1, that is $\alpha$ = 1/2 as well as the values obtained for Precision = (P) and Recall = (R):

$$F1Measure = \frac{2PR}{P + R} \tag{5.5}$$

Table 5.2 below are the results for each item and accuracy rate calculated for each case study.

Table 5.2: Case Studies Results

| Case Studies Group | Experiment | **TP** | **FP** | **TN** | **FN** | Accuracy rate | Precision | Recall | F1 Measure |
|---|---|---|---|---|---|---|---|---|---|
| **G**roup 1 | Text 1 | 18 | 14 | 4 | 8 | 50% | 56,25% | 69,23% | 62,07% |
| **G**roup 1 | Text 2 | 3 | 4 | 1 | 1 | 44,44% | 42,86% | 75% | 54,55% |
| **G**roup 2 | Text 1 | 5 | 0 | 1 | 0 | 100% | 100% | 100% | 100% |
| **G**roup 2 | Text 2 | 7 | 2 | 1 | 3 | 61,54% | 77,77% | 70% | 73,68% |
| **G**roup 3 | Text 1 | 8 | 1 | 1 | 1 | 81,81% | 88,88% | 88,88% | 88,88% |
| **G**roup 3 | Text 2 | 7 | 2 | 3 | 3 | 71,43% | 87,5% | 70% | 77,77% |

Looking at Table 5.2 we can infer that there are big differences between the results in the case study group 1. The first text is the largest one with 38 lines and the second is short with only 8 lines. But, besides factor of the size of the texts, they are the ones with less connection with the reality. While the first text that was extracted from an IBM white paper is used for study purposes and sometimes repeats a lot some information, the second one was taken from a book and represents a voice of a child asking for a software. The lack of clarity and concise information cause a lot of troubles for processing it correctly.

The text of the first case study of the group one correctly generated 18 user stories, that are the true positives, but unfortunately, it presented 14 false positives. The number of 8 false negative represents the user stories that should be generated or correctly generated but they were not. The false negatives are compounded by 4 user stories not generated and others 4 user stories that were not generated correctly, which means that the sentence which should generate a user story has generated it with some errors. The metrics for the text of the first case study were: 50% of accuracy rate, 56,25% of Precision, 69,23% of Recall and a F1 Measure of 62,07%.

In the text 2 of the first group of case studies there were 3 user stories true positives, 4 user stories considered as false positive, one true negative and one false negative which was a expected user story that was generated with errors as it's possible to see in the JSON below:

```
{
   "title": "mean website fast have",
   "description": "As This I want to be able to mean the website have
      to be fast",
   "systemEntity": { "name": "website fast have" },
```

```
6     "mainAction": "mean",
7     "user": { "name": "This" }
8  }
```

As shown in the **JSON** file piece above, the **title** attribute was generated with some words in the wrong order, and even if the description text is something possible to understand, the **mainAction** attribute is holding the wrong verb, what means that the classification of the root nodes from the dependency graph is not right for this case. Another thing that had a negative impact in the third text of the first case studies group was the way the author used to refer to the user of the system, always using the first person of the singular to talk about the features he wanted to have in the software instead of writing from the user perspective and naming the users of the application. All of these characteristics of this text made hard the processing and UserStoryGen tool has presented an accuracy rate of 44,44%, precision of 42,86%, a recall of 75% and 54,55% as the F1 Measure for this text, which are the worsts metrics obtained between all the case studies results.

The text of the first case study from the second group of case studies appears with 5 true positive, or 5 user stories correctly generated as expected, 1 true negative correctly discarded from the text and no false positive or false negative, which is a perfect result and gives to this case study 100% of Accuracy rate, precision, recall and F1 Measure.

The text of the second case study from the second group of case studies had the number of 7 true positives (user stories generated which should be generated according to the business rules), 2 false positives, the user stories generated which shouldn't be, 1 true negative correctly discarded, and 3 false negatives according to Table 5.2. From this number of 3 false negatives there was one user story which should be generated, but was not correctly generated due to errors in the processing and others 2 that was not even generated. The second case study of the second case study group has an accuracy rate of 61,54%, a precision of 77,77%, a recall of 70%, and an F1 measure of 73,68%.

The text two of the second case studies group has failed to generate 3 user stories, one of them was generated with errors and others 2 were not even generated, that is the number of **false negative** in this table. This failure to generate these stories consisted in an issue to treat the case of multiple verbs when they don't have a complement for all the verbs, like in the phrase: "This feature will include, deleting, and updating the product information as well. ", just the verb **updating** presented a complement, and this made impossible the generation of a user story for the action of **deleting the product information**. The only user story generated for this specific sentence was the one with the title **update the product information**.

The results for the second group of case studies were greater in comparison with the first group. With F1 measure of 100% and 73,68% for the text 1 and 2 respectively, the second group of text is closer to the reality texts than the first group case studies texts. The texts of the second group have two great characteristics: they are short and direct to the point while describing the idea of the desired software.

The third group of case studies which had only real texts from a company product was surprisingly the group with the best average of results. The results from the text one and two of the last case study group were not only good, but, they have the closest rates in comparison with other group texts. The first text of the third case studies group has presented 81,81% of accuracy or success rate in Table 5.2, what is a great result in terms of correct classification of what should be a user story, and also correct generation of the user story information. This text was a collection of introductory sentences from some User Journey documents, and its both precision, recall and F1 Measure of 88,88% is really good considering that it's a real text from the industry.

The second text of the third case studies group which contained Features descriptions appears in Table 5.2 with 71,43% of accuracy, a precision of 87,5%, recall equals to 70% and F1 Measure of 77,77%. This result can be considered a good one taking into consideration that it had 7 user stories correctly generated as expected, 3 sentences correctly discarded, the 3 true negative, and only 2 false positive. The 3 false negative number was something that could be lower if it wasn't one user story with bad generation which was counted as a false negative.

The general rates based on the case studies groups, plus the error rate will be shown in Table 5.3 below:

Table 5.3: Results by Case Studies Group

| Case Study Group | **TP** | **FP** | **TN** | **FN** | Accuracy or success rate | Precision | Recall | F1 Measure | Error rate |
|---|---|---|---|---|---|---|---|---|---|
| **G**roup 1 | 26 | 18 | 6 | 9 | 60% | 53,85% | 70% | 60,87% | 50% |
| **G**roup 2 | 12 | 2 | 2 | 3 | 73,68% | 85,71% | 80% | 82,76% | 26,31% |
| **G**roup 3 | 15 | 2 | 4 | 4 | 76% | 88,23% | 78,95% | 83,33% | 24% |

Table 5.3 shows that the group 1 with texts from books and one white paper has the worst result with the accuracy average of 64%, the precision average of 59,09%, 74,28% of recall average, F1 Measure of 65,82% considering the hits of all the 3 texts and an error rate of 45,76%. It's well known that the bad results of the second and third texts of this group decreased the overall average as that text was too long and presented a lot of false positive elements, but the third text of the first group also presented some difficulties in by not being concise in the explanation of the desired software.

According to Table 5.3, the second group of texts and the third one had similar averages for all the factors analyzed. While the group two presented 73,68% of accuracy rate, 85,71% of precision, 80% of recall, and F1 Measure of 82,76%, the third group had a little better result with 76% of accuracy rate, 88,23% of precision, a recall rate of 78,95% and an F1 Measure of 83,33%, which is just a little better than the same metric value of the second group.

It's important to consider that the groups two and three had texts better written than the first group, and texts with the focus on the user actions which were shorter than the group one texts and also very concise. That's why they have a low error rate of respectively 26,31% and 24%, while the group one presented an error rate of 45,76% as shows Table 5.3. As it's possible to see in Table 5.3 the third group of case studies has the better average of results for all the metrics analyzed.

## 5.5 Conclusion

The results presented in this chapter suggest that concise texts with the description of the software focused on the user perspective have the greatest result in comparison with long texts with a large explanation and low user reference. The metrics result showed that texts with a high user reference and which were closest to the reality, the third group of case studies with real texts from the industry, had the best average of accuracy per group of case studies.

With an average accuracy of 60%, precision of 53,65%, a recall of 70% and a F1 measure of 60,87%, the results of the first case studies group were considered the lowest in comparison with the other groups of case studies. The first text that was the biggest one analyzed considering all the case studies had a lower result with an accuracy of 50% and F1 measure of 62,07%. This

first text had a high number of false positives, user stories that shouldn't be generated, but, they were. The 14 false positives are the consequence of a many sentences that were there in the text only to explain something, sometimes in a redundant way, what had increased this negative number.

The third text of the first group of case studies had the lowest rates of the first case studies group: 44% of accuracy, 42,86% of precision, 75% of recall and an F1 measure of 54,55%. This bad result can be related to the fact that the speech centralized in the first person, missing the focus on the user had caused several issues for the user story generation. At the same time that this text generated 3 expected user stories, it identified based on the actions mentioned in the text other 4 possible user stories which were not correct, and this number of false positives decreased the metrics.

Among the hardest factors for the text processing, it was noticed during the development of the UserStoryGen tool and also during the testing, that the author sentences splitting algorithm was one of the key points to success or error in the process as a whole. When the dependency graph doesn't identify correctly the verbs that are root nodes, it becomes very hard to make a correct analysis and to split the sentence correctly. Another hard case: the verbs that don't have complement right after them, in sentences of multiple verbs is something very hard to process, and as this case brought a lot of errors, it's certainly something to fix in the next versions of the UserStoryGen.

When a sentence has an enumeration of actions and each verb has its complement, the UserStoryGen brought excellent results. The functions provided by *Stanford CoreNLP* that uses dependency graph to determine the start and end of each part of the sentence were not used in this sentence splitting process, due to the bad results during the development of the UserStoryGen tool. The responsibility for these good results is given to the fact that the author sentence splitting algorithm breaks the sentence considering some basic rules:

- All the verbs root nodes and its position in the sentence are collected at the first moment;
- Each verb plus all the words before the next verb are grabbed to build each partial sentence;
- To be consider a valid partial sentence, it's verified if the partial sentence has an object connected to the verb. If it doesn't have an object and it's the first partial sentence evaluated, it's simply discarded, otherwise the partial sentence is added to the previous partial sentence created;
- By the end of this process, each partial sentence passes through a method which eliminates stop words from the end of each partial sentence.

The excellent results of 100% of accuracy and F1 measure obtained with the first case study of the second case studies group showed that the proposed approach has a high performance splitting the passive voice sentences with multiple verbs, and also a great performance in the identification of each element that compound the user stories.

The second group of case studies which had texts from specialists in the Software Engineering area, presented an average accuracy of 73,68% and an F1 measure of 82,76%, and the third case studies group, which contained texts from a company product, had an even greater accuracy average of 76% and F1 measure of 83,33%.

The texts provided by a company presented the best average for Accuracy, precision, F1 Measure rates and the lowest error rate of 24% while the first group with texts from books and a white paper had the worst accuracy average with 64%, F1 measure of 65,82% and an error rate of 45,76%.

# 6 Conclusion

In this work we proposed the UserStoryGen approach and tool for extracting user stories from unstructured text of a given software big picture or software specification. UserStoryGen tool implemented the approach using Natural Language Processing techniques and had some key objectives to produce a valuable result.

Studying natural language processing techniques and based on all the problems that should be solved during the implementation of UserStoryGen, it was created a structure of pipelines of processing, which are processing structures that have the NLP tasks strategic positioned to build the user stories and keep a good performance.

The goal of creating a way for identifying user or actor mentions in the text and its reference with other phrases for replacing pronouns with the user reference was successfully achieved in the implementation of the tool through the First Pipeline structure. Besides to solve the coreference issue, the First Pipeline implementation had tasks such as removal of specific stop words to make easy the processing and also the generation of the dependency graph.

In the Second Pipeline structure, it was solved all the remaining goals related to the implementation of UserStoryGen. In the beginning of the Second Pipeline, it's executed the task of the identification of the number of verbs to simplify the treatment for each case: single verb sentence or multiple verb sentence. Later, it's achieved another goal that it's the identification of active or passive voice for a given sentence with multiple verbs, which allows separated processing for each case using the dependency graph of the given sentence. Then, the next step in the second pipeline structure is to recognize which are the main verbs of each sentence for better splitting them through the information obtained from the dependency graph, and these tasks complete two other specific goals.

After the sentence splitting in the case of a multiple verbs sentence, or even after the single verb recognition, each sentence or splice sentence is passed through methods which extract the main verb, the system entity, the user or actor information and also build the title and description of each user story. The list of the user stories with these attributes is also another specific objective that was successfully achieved.

The last specific goal was achieved in this work during the testing of the built tool. One of the specific goals was to have a success rate at least greater than the error rate, but the results were even greater than expected mainly for the case studies with texts provided by the industry.

The evaluation of UserStoryGen tool was done using 3 groups of case studies, the first group containing a white paper and a book text, the second group with texts provided by specialists in the Software Engineering area, and the last group of case studies was done using texts provided by a company, which were Product feature texts and a collection of introduction texts taken from User Journey documents. Between the three groups of case studies, the worst accuracy rate or success rate was 60% obtained by the first group. The first group of case studies had a precision of 53,85%, recall of 70%, F1 measure of 60,87% and the worst error rate of 50%.

The second group of case studies had a better result in comparison to the first group, its first case study text had a great result with 100% for both accuracy, precision, recall and F1

measure, and zero errors, the second text of the second group had lower metrics result with 61,54% of accuracy rate, a precision of 77,77%, 70% of recall and an F1 measure of 73,68%. The average of the results of the group two were: 73,68% of accuracy, 85,71% of precision, 80% of recall, an F1 measure of 82,76% and an error rate of 24%.

The third group of case studies had the best average between the metrics of all the groups, with an average accuracy of 76% between the two texts tested, an average precision of 88,23%, recall of 78,95% and an average F1 measure of 83,33%, this group presented the lowest error rate with 24%. Between the two texts analyzed in this group there were some differences in the results, with the first text having an F1 measure of 88,88% while the second text presented only 77,77% for the same metric, but the differences were not too large in comparison with the other groups results per case study text.

UserStoryGen tool had presented a good initial result in the user stories extraction, and all the knowledge gained from the implementation and testing of the tool can be used in future works to improve even more the accuracy and precision of the results. The initial idea was to build a tool that could have user intervention before generating the user stories in order to fix any possible issue with the automatic generation. Due to time reasons, no all the initial ideas were applied and there were the mentioned points of improvement needed in the phase of sentence splitting to have better results. All the lessons learned treating the active and passive voice sentences, as well the limitations of the Stanford Core NLP which motivated self development of many functions in terms of natural language processing tasks can be helpful to achieve another level in the precision of the user stories extracted.

## 6.1 Future works

In the future works there are some possibilities for improvements in the implementation of the tool and also extending some capabilities of the proposed approach.

Among the future works, there is the possibility of improving the sentence splitting and the separation of a verb plus complement pairs, especially in the case of enumeration of verbs that mention a common object just at the end of the sentence connected only with the last verb. This change will have great impact, for each partial sentence which will be correctly treated with this additional implementation, one new true positive will be increased and one false negative will be decreased from text metrics.

To improve the identification of system entities is another desirable future work. This improvement can be done by clustering the system entities found, based in the *lemma* of their core word.

To start saving the information in a database before returning the list of user stories throughout the endpoint is also a good option which can enable other improvements such as: allow the user to edit the results and save it into the back-end. This functionality will enable the user to save the perfect result and export the results is also another welcome feature.

Another feature that it's possible to implement after to start saving all the generated results, and saving the changes from the user is to learn with the results edited by the user of the tool. After some time of usage the database of the proposed tool will have some data and parameters to start learning from the user final modifications over the generated user stories. Hence, applying the knowledge of the user decisions about the correctness of the user stories generated can help in decreasing the number of false positives, what positively affects all the tool metrics.

# References


Atdag, S. e Labatut, V. (2013). A comparison of named entity recognition tools applied to biographical texts. *CoRR*, abs/1308.0661.

Badihi, S. e Heydarnoori, A. (2017). Crowdsummarizer: Automated generation of code summaries for java programs through crowdsourcing. *IEEE Software*, 34(2):71–80.

Bird, S., Klein, E. e Loper, E. (2009). *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.".

Bozyiğit, F., Aktaş, Ö. e Kılınç, D. (2016). Autoclass: Automatic text to oop concept identification model. *International Journal of Computer Applications*, 150(10):29–34.

Christopher D. Manning, P. R. . H. S. (2009). Stemming and lemmatization. Available on `https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf`. Accessed in 05/10/2017.

Cohn, M. (2004). *User Stories Applied: For Agile Software Development*. Addison-Wesley signature series. Addison-Wesley.

De Marneffe, M. C. e Manning, C. D. (2008). Stanford typed dependencies manual. Available on `https://nlp.stanford.edu/software/dependencies_manual.pdf`. Accessed in 11/05/2017.

dos Santos, C. N. e Gatti, M. (2014). Deep convolutional neural networks for sentiment analysis of short texts. Em *COLING*, páginas 69–78.

Dybå, T. e Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Inf. Softw. Technol.*, 50(9-10):833–859.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Tese de doutorado, University of California, Irvine - USA. AAI9980887.

Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N. e Steiner, W. (2014). Automatically extracting requirements specifications from natural language. *CoRR, abs/1403.3142*.

Harrison (2015a). Lemmatizing with nltk. Available on `https://pythonprogramming.net/lemmatizing-nltk-tutorial/`. Accessed in 01/09/2017.

Harrison (2015b). Tokenizing words and sentences with nltk. Available on `https://pythonprogramming.net/tokenizing-words-sentences-nltk-tutorial/`. Accessed in 01/08/2017.

Kamthan, P. (2015). A comparison of use cases and user stories. Em *Encyclopedia of Information Science and Technology, Third Edition*, páginas 6949–6955. IGI Global.

Leffingwell, D. (2017). Features and capabilities. Available on `https://www.scaledagileframework.com/features-and-capabilities/`. Accessed in 5/04/2017.

Leffingwell, D. e Widrig, D. (2003). *Managing Software Requirements: A Use Case Approach Second Edition*. The Addison-Wesley object technology series. Addison-Wesley.

Maldonado, E., Shihab, E. e Tsantalis, N. (2017). Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J. e McClosky, D. (2014). The Stanford CoreNLP natural language processing toolkit. Em *Association for Computational Linguistics (ACL) System Demonstrations*, páginas 55–60.

Masse, M. (2011). *REST API Design Rulebook*. O'Reilly Media, Sebastopol.

Masuda, S., Matsuodani, T. e Tsuda, K. (2016a). Detecting logical inconsistencies by clustering technique in natural language requirements. *IEICE Transactions on Information and Systems*, 99(9):2210–2218.

Masuda, S., Matsuodani, T. e Tsuda, K. (2016b). Syntactic rules of extracting test cases from software requirements. Em *Proceedings of the 2016 8th International Conference on Information Management and Engineering*, páginas 12–17. ACM.

Merten, T., Falis, M., Hübner, P., Quirchmayr, T., Bürsner, S. e Paech, B. (2016). Software feature request detection in issue tracking systems. Em *Requirements Engineering Conference (RE), 2016 IEEE 24th International*, páginas 166–175. IEEE.

Olaverri-Monreal, C., Hasan, A. E. e Bengler, K. (2013). Semi-automatic user stories generation: To measure user experience. Em *Information Systems and Technologies (CISTI), 2013 8th Iberian Conference on*, páginas 1–6. IEEE.

Panichella, S., Di Sorbo, A., Guzman, E., Visaggio, C. A., Canfora, G. e Gall, H. C. (2016). Ardoc: App reviews development oriented classifier. Em *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, páginas 1023–1027. ACM.

Patel, C. e Ramachandran, M. (2009). Story card based agile software development. *International Journal of Hybrid Information Technology*, 2(2):125–140.

Pinquié, R., Véron, P., Segonds, F. e Croué, N. (2016). Requirement mining for model-based product design. *International Journal of Product Lifecycle Management*, 9(4):305–332.

Pinto, A., Gonçalo Oliveira, H. e Oliveira Alves, A. (2016). Comparing the performance of different nlp toolkits in formal and social media text. Em *OASIcs-OpenAccess Series in Informatics*, volume 51. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Pressman, R. (2000). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, Boston, USA, 5 edition.

Quirchmayr, T., Paech, B., Kohl, R. e Karey, H. (2017). Semi-automatic software feature-relevant information extraction from natural language user manuals. Em *International Working Conference on Requirements Engineering: Foundation for Software Quality*, páginas 255–272. Springer.

Ramnani, R. R., Shivaram, K., Sengupta, S. et al. (2017). Semi-automated information extraction from unstructured threat advisories. Em *Proceedings of the 10th Innovations in Software Engineering Conference*, páginas 181–187. ACM.

Rane, P. P. (2017). *Automatic Generation of Test Cases for Agile using Natural Language Processing*. Tese de doutorado, Virginia Tech.

Rasmusson, J. (2010). *The Agile Samurai: How Agile Masters Deliver Great Software*. Pragmatic Bookshelf ©2010.

Rodriquez, K. J., Bryant, M., Blanke, T. e Luszczynska, M. (2012). Comparison of named entity recognition tools for raw OCR text. Em Jancsary, J., editor, *Proceedings of KONVENS 2012*, páginas 410–414. ÖGAI. LThist 2012 workshop.

Sabira, A. e Uthradevi, K. (2017). Topic behavioral study of microblog content. *Software Engineering and Technology*, 9(3):53–56.

Shu, Y., HaiLun, Y., XiangRun, Y. e Ye, W. (2016). An automated method for constructing ontology. Em *Software Engineering and Service Science (ICSESS), 2016 7th IEEE International Conference on*, páginas 538–541. IEEE.

Singh, P., Singh, D. e Sharma, A. (2016). Rule-based system for automated classification of non-functional requirements from requirement specifications. Em *Advances in Computing, Communications and Informatics (ICACCI), 2016 International Conference on*, páginas 620–626. IEEE.

Sugawara, S., Yokono, H. e Aizawa, A. (2017). Prerequisite skills for reading comprehension: Multi-perspective analysis of mctest datasets and systems. *AAAI*, páginas 3089–3096.

Suri, N., Subramanian, S. e Sproule, W. D. (2015). Automated story generation. Published by Google Patents, US Patent 9,161,007.

Trim, C. (2012). Ontology-driven nlp. Available on `https://www.ibm.com/developerworks/community/blogs/nlp/entry/ontology_driven_nlp?lang=en`. Accessed in 05/01/2017.

Ye, D., Xing, Z., Li, J. e Kapre, N. (2016). Software-specific part-of-speech tagging: An experimental study on stack overflow. Em *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, páginas 1378–1385. ACM.

Zhang, B., Hill, E. e Clause, J. (2016). Towards automatically generating descriptive names for unit tests. Em *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, páginas 625–636. IEEE.

Zolotas, C., Diamantopoulos, T., Chatzidimitriou, K. C. e Symeonidis, A. L. (2016). From requirements to source code: a model-driven engineering approach for restful web services. *Automated Software Engineering*, páginas 1–48.