HENRIQUE VARELLA EHRENFRIED

# EXTRAÇÃO DE POSSÍVEIS CASOS DE USO DE TEXTOS ESCRITOS USANDO LINGUAGEM NATURAL

Proposta de Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Andrey Ricardo Pimentel

CURITIBA

2016

**UNIVERSIDADE FEDERAL DO PARANÁ**

HENRIQUE VARELLA EHRENFRIED

# EXTRAÇÃO DE POSSÍVEIS CASOS DE USO DE TEXTOS ESCRITOS USANDO LINGUAGEM NATURAL

CURITIBA

2016

HENRIQUE VARELLA EHRENFRIED

# EXTRAÇÃO DE POSSÍVEIS CASOS DE USO DE TEXTOS ESCRITOS USANDO LINGUAGEM NATURAL

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Universidade Federal do Paraná como requisito à obtenção do título de obtenção do grau de Bacharel em Ciência da Computação, pela seguinte banca examinadora:

Orientador: Prof. Dr. Andrey Ricardo Pimentel
Departamento de Informática, UFPR

Prof. Dr. Laura Sánchez García
Departamento de Informática, UFPR

Prof. Dr. Silvia Regina Vergilio
Departamento de Informática, UFPR

Curitiba, 16 de dezembro de 2016

*"Não encontre um problema, encontre uma solução"*

*Harry Ford*

# RESUMO

Este trabalho implementa uma abordagem baseada no processamento de linguagem natural para realizar a elicitação automática de casos de uso a partir de um texto de requisitos. Nesta abordagem o texto é utilizado para criar uma árvore de dependências de palavras para facilitar a identificação dos atores e casos de uso. Entretanto o usuário precisa interagir com o resultado da aplicação, pois as saídas da ferramenta são os possíveis casos de uso e cabe ao usuário definir quais são os verdadeiros casos de uso.

**Palavras-chave:** caso de uso, elicitação automática, linguagem natural.

# ABSTRACT

This work implements an approach based on natural language processing to perform automatic use case elicitation from a requirements text. In this approach the text is used to create a dependency tree to facilitate the identification of actors and use cases. However the user needs to interact with the result of the application, because the outputs of the tool are the possible use cases and it is up to the user to define what are the real ones.



**Keywords:** Use case, automatic elicitation, natural language.

# SUMÁRIO

# CAPÍTULO 1

# INTRODUÇÃO

Todo projeto de software começa com uma especificação. Esta especificação contém dados chave para o projeto que são relativos aos principais *stakeholders*, às funcionalidades esperadas do sistema, ao processo que os usuários do sistema precisam executar e à forma eles precisam executá-los. Portanto é importante que a fase de projeto de software analise criteriosamente qualquer tipo de especificação que for passada pelo cliente. Como uma das formas mais comuns de passar conhecimento adiante é por meio da escrita, muitos clientes acabam escrevendo textos contendo todas as informações relacionadas ao software.

Desta forma um projetista de software deve ler este documento para extrair as informações relevantes para a construção do software. Este processo é cansativo, já que requer que o mesmo texto seja lido inúmeras vezes para evitar a perda de detalhes mínimos. Para tentar auxiliar os projetistas de software, pesquisas, como a que criou o software UMGAR [7] têm sido realizadas para que softwares interpretem as especificações e gerem diagramas UML da arquitetura do software, fazendo com que o texto seja lido apenas para remover dúvidas e explicar requisitos não funcionais [7][18].

## 1.1 PROBLEMA

Diversos problemas aparecem na avaliação dos requisitos. O primeiro é a falta de padronização do texto, como textos são escritos usando linguagem natural, o escritor pode se apropriar de figuras de linguagem que não são facilmente processadas nem por seres humanos, tampouco por máquinas. Outro problema, ainda relacionado com os textos escritos, é o fato de frases muito compridas serem difíceis de serem processadas por pessoas, causando dúvidas que programas bem desenvolvidos para interpretar textos conseguem resolver.

Textos podem ainda causar o problema de domínio, onde o projetista de software pode

não compreender o domínio da aplicação e por isso não identificar corretamente alguns componentes dos artefatos que o software deve ter.

Do ponto de vista computacional há o problema de coletar a semântica dos textos utilizando apenas algoritmos de processamento de linguagem natural, para tanto usualmente são utilizados algoritmos de inteligencia artificial, como, por exemplo, redes neurais ou redes neuroevolutivas. Estes algoritmos podem ser treinados para identificar a semântica do texto, permitindo desta forma que haja uma melhora na precisão da interpretação do texto. Esta melhora pode impactar na elicitação automática dos requisitos.

## 1.2 OBJETIVOS

A proposta deste trabalho é a criação de um software que dado um texto não estruturado, sugira os possíveis casos de uso presentes neste texto. Para chegar a este software será preciso processar linguagem natural e associar o resultado do processamento do texto com os conceitos de casos de uso. Além disso o software precisará de um ambiente de interface e interação para o usuário consistente para que possa ser facilmente utilizado pelo usuário.

Com esta funcionalidade e esta interface consistente é esperado que este trabalho auxilie os profissionais da área no entendimento do problema e na modelagem do software para que a qualidade do software seja aumentada[9].

## 1.3 JUSTIFICATIVA

O desenvolvimento deste software possibilitará que seus usuários criem diagramas de casos de uso mais facilmente, já que atualmente as ferramentas disponíveis criam apenas destacam palavras e frases chave [12][18] ou exigem muita interação do usuário[7]. Para que a solução proposta funcione é necessário entrar com o texto de especificação e escolher os casos de uso que são realmente casos de uso. Desta forma profissionais da área de engenharia de requisitos poderão prototipar rapidamente um diagrama de casos de uso.

Outra contribuição é a melhora no entendimento do problema, uma vez que o software

separa as sentenças por ator, é mais fácil compreender o texto original.

## 1.4 APRESENTAÇÃO DO DOCUMENTO

Este trabalho está divido em mais quatro capítulos. O capítulo seguinte apresenta os principais conceitos envolvidos com este trabalho. O capítulo três descreve como a ferramenta foi construída e quais tecnologias foram utilizadas para construí-la. O capítulo quatro exemplifica o funcionamento do software construído e analisa o resultado obtido. Por fim o último capítulo conclui este trabalho apresentando as conclusões obtidas, a contribuição deste trabalho e os trabalhos futuros.

# CAPÍTULO 2

# REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta os conceitos teóricos que serão utilizados neste trabalho. Na Seção 2.1 são apresentados os conceitos básicos de requisitos e os tipos de requisitos mais comuns. A Seção 2.2 explica o que são e como funcionam os caso de uso. Já a Seção 2.3 introduz os conceitos utilizados no trabalho para analisar texto. A Seção 2.4 discute o estado da arte da elicitação automática de artefatos a partir de textos não estruturados. Por fim, a Seção 2.5 sintetiza este capítulo e introduz o próximo capítulo.

## 2.1 REQUISITOS

Basicamente requisitos de software são propriedades que o software deve conter para poder resolver um problema de negócio que o software se propõe a fazer. Estas propriedades podem ser classificadas de várias formas [4]. As principais classificações são:

- Requisitos de processos e produtos
- Requisitos funcionais e não funcionais

### 2.1.1 Requisitos de processos e produtos

Um requisito de produto é uma necessidade ou uma restrição do software que será produzido [4]. Por exemplo, o software deve verificar se o Cadastro de Pessoa Física (CPF) existe e se é válido. Enquanto que um requisito de processo é uma restrição de como o software deve ser desenvolvido [4]. A título de exemplo, o software deve ser desenvolvido usando SCRUM.

### 2.1.2 Requisitos funcionais e não funcionais

Um requisito funcional descreve as funções que o software vai executar [4], por exemplo, cadastrar cidadão. Os requisitos funcionais também são conhecidos como "características"do software.

Por sua vez, os requisitos não funcionais são aqueles que agem para limitar a solução. Eles usualmente dizem respeito a como deverá ser implementado o software em termos da qualidade do software. Isto significa que são requisitos que especificam o desempenho, a manutenibilidade, a segurança, a confiabilidade ou qualquer outro atributo similar [4].

## 2.2 CASOS DE USO

Antes de apresentar o conceito de casos de uso é importante saber que um *stakeholder* é algo (sistema, instituição, objeto, etc.) ou alguém (gerente, operário, etc.) que está relacionado com o sistema de alguma forma. O conceito de *stakeholder* é importante para o caso de uso, pois é ele que interage com o sistema. Os *stakeholders* que participam mais ativamente de um caso de uso é chamado pela engenharia de requisitos de "ator"[6].

Um caso de uso é um contrato entre um *stakeholder* e o comportamento do sistema, definido por um ou mais requisitos funcionais. Desta forma um caso de uso descreve o comportamento do sistema em várias condições de operação [6].

Para elicitar os casos de uso devem ser realizados alguns passos [14]:

1. A fronteira do sistema deve ser estabelecida
2. Os atores devem ser identificados
3. Para cada ator devem ser definidos seus objetivos
4. Definir um conjunto de casos de uso que cumpra os objetivos dos atores

O passo 1 serve para definir qual problema o sistema deverá resolver e quais são os *stakeholders*, de forma que não será construído um sistema com mais funcionalidades que o necessário.

Os passos 2 e 3 por sua vez devem ser executados para identificar os atores que interagirão com o software e o que cada um destes atores precisa ser capaz de fazer no sistema.

Com os passos 2 e 3 concluídos, seus resultados devem ser formalizados. É justamente nisso que consiste o passo 4, na formalização dos dois passos anteriores.

Casos de uso são fundamentalmente texto, entretanto podem ser expressos em formato de diagrama [6]. O diagrama mais comum para representar casos de uso é o diagrama de casos de uso da UML (Unified Modeling Language) [3].

Os diagramas de caso de uso são criados na fase de concepção do software e aprimorados durante a fase de elaboração [17]. Estes diagramas serão mais tarde utilizados para definir a arquitetura do software.

Na UML um ator é representado por boneco palito com seu nome abaixo, um caso de uso é representado por uma elipse com uma brevíssima descrição do caso de uso dentro da elipse [3]. Por fim as relações entre casos de uso e atores são representadas com linhas interligando um ator com um caso de uso [3]. A Figura 2.1 exemplifica um caso de uso.

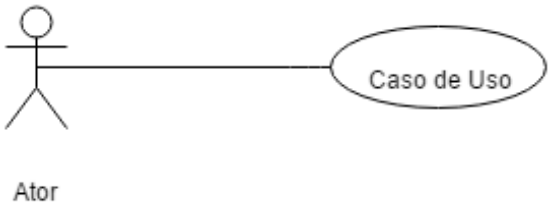


Figura 2.1: Diagrama de Caso de uso no padrão UML

A UML ainda permite casos especiais, não utilizados neste trabalho, de extensão e inclusão. Após o diagrama gráfico da UML montado, usualmente é realizada a criação de um documento onde todos os eventos de todos os casos de usos modelados no diagrama são descritos e explicados. Este documento é conhecido na engenharia de requisitos como "especificação de casos de uso".

Usualmente um caso de uso começa com um ator primário, que é o *stakeholder* principal do caso de uso, iniciando uma interação com o sistema [6]. Em seguida o sistema responde de alguma forma. A partir de então, uma sequência de interações pode ou não começar a ocorrer, inclusive com a interação de outros atores. Cada sequência de intera-

ção é chamada de "evento". Cada caso de uso tem um conjunto de eventos, que juntos, o descrevem completamente, tanto os casos de sucesso quanto os casos de erro[3].

A aplicação desenvolvida neste trabalho sugere possíveis casos de uso e seus atores para que possam ser representados no diagrama gráfico da UML.

## 2.3 ANÁLISE AUTOMÁTICA DE TEXTO

A análise de texto pode ser realizada por análise de máquinas de estados, análise estrutural, técnicas heurísticas e técnicas de aprendizagem de máquina [13]. A estratégia que foi utilizada neste trabalho consiste na análise estrutural das sentenças que compõem o texto. Assim como seres humanos fazem, a ideia por trás desta técnica é classificar as palavras em classes como "substantivo"e "verbo", para depois gerar uma árvore sintática, ou como também é conhecida, uma árvore de dependência.

Uma árvore de dependência é um grafo que contém as relações entre as palavras da sentença [16]. Desta forma, este grafo é uma possível representação computacional de uma análise sintática, onde cada arco representa uma relação, como "sujeito"e "verbo auxiliar". No meio computacional cada implementação de árvores de dependência é diferente, já que há várias formas de representar um grafo e várias formas de atribuir uma relação entre dois vértices de um grafo. Portanto a implementação que foi utilizada neste trabalho será explicada mais adiante na Seção 3.1.1.

Uma abordagem alternativa para realizar uma análise automática de texto consiste em classificar as palavras do texto e criar uma árvore de dependência. A partir da árvore de dependência, algoritmos que exploram certos atributos das palavras e sua posição na sentença realizam o processamento desejado[11]. Esta forma de processar o texto é possível porque a Língua Inglesa, idioma base deste trabalho por ter sua estruturação gramatical mais simples, esta estrutura é usada para que sentenças possam ser criadas, de forma que algumas palavras possam ser trocadas dentro de um texto sem perder a corretude do texto, apenas, em alguns casos, o sentido [2].

## 2.4 ELICITAÇÃO AUTOMÁTICA DE CASOS DE USO E OUTROS ARTEFATOS

A obtenção de conhecimento a partir de textos não estruturados têm sido estudada há vários anos como mostram as publicações [7] [12] [18], como o. Algumas pesquisas revelam resultados interessantes como o *Textractor* [12] que coleta informações de reclamações de clientes e de ações de reparo feitas engenheiros para então processar estes dados por meio de vários algoritmos. Após este processamento, são avaliados os resultados em conjunto com especialistas na área e o resultado que o software obteve em precisão foi de 91.7%.

Outro caso de sucesso na elicitação automática de artefatos, este na área de engenharia de software, é o UMGAR [7] que utiliza várias técnicas de pré-processamento como a reescrita de sentenças para simplifica-las. Em seguida são analisadas sentenças resultantes para extrair atores, classes, atributos, métodos e associações.

Há, ainda, um estudo realizado para facilitar a especificação de requisitos, chamado de *Dowser tool* [18], esta abordagem classifica as palavras e partes da sentença para então extrair termos específicos de frases nominais. Segundo os pesquisadores, esta técnica deve ajudar os seres humanos na extração de requisitos.

## 2.5 CONCLUSÕES

Embora os conceitos de requisitos e casos de uso já sejam bem definidos e consolidados, ainda é preciso avançar muito na área de processamento de linguagem natural para automatizar a criação de artefatos. Este trabalho visa a contribuir justamente neste espaço. Para tanto foi criado um software que processa texto para extrair os possíveis casos de uso presentes no texto. A seguir será detalhada a arquitetura do software, seu processo de concepção e sua forma de uso.

# CAPÍTULO 3

# DESCRIÇÃO DA FERRAMENTA

Neste capítulo será explicada a arquitetura do software na Seção 3.1, em seguida a Seção 3.2 apresenta o modo como a ferramenta foi concebida. Por fim a Seção 3.3 explica como instalar e utilizar a ferramenta enquanto que o Seção 3.4 sintetiza o capítulo.

## 3.1 ARQUITETURA

A ferramenta GPCU é dividida em duas partes, há o módulo de processamento escrito em Python e que contém toda a lógica de processamento do texto recebido como entrada. Há ainda um módulo gráfico escrito em HTML/CSS/Javascript que responde por toda a parte gráfica da aplicação. Basicamente a arquitetura do sistema pode ser representada pela Figura 3.1, a qual demonstra uma visão geral do software, desde seu início até a finalização de um processamento.

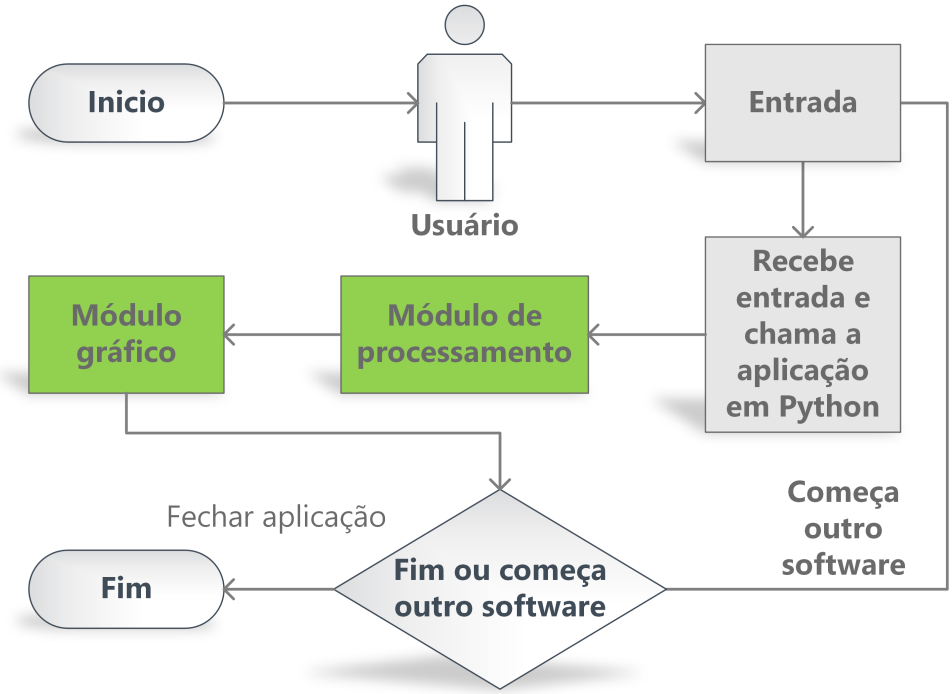


Figura 3.1: Arquitetura do software gerador de possíveis casos de uso

A Figura 3.2 mostra, com detalhes, como é a interação entre os módulos do sistema e o usuário. Assim que o usuário inicializa a aplicação ele é indagado quanto a forma da entrada de dados para o sistema, isto é, se será realizada a partir de uma arquivo texto ou se será utilizado o próprio sistema para a inserção da descrição do software para análise.

Feito isso, o texto recebido como entrada alimenta o no módulo de processamento que começa sua execução separando as sentenças do texto. A separação de sentenças é feita por pontos (".") e cada sentença é armazenada em uma posição de um vetor de sentenças. Para separar as sentenças foi utilizado o método *split* da linguagem Python. Realizada esta etapa, cada sentença recebe um tratamento de dados da biblioteca SpaCy a qual gera informações de sintaxe e morfologia para cada palavra. Após este tratamento a biblioteca cria a árvore de dependência da sentença. Com as árvores de dependência geradas a aplicação processa cada árvore de dependência para extrair o(s) sujeito(s). Já em posse do sujeito, ele é processado para que não seja nenhum pronome, substituindo o pronome pelo último sujeito que não é pronome encontrado. Depois de processado, o sujeito será utilizado para encontrar seus verbos, para tanto usa-se o arco de chegada do sujeito e encontra-se seu verbo. Este verbo é a ação que o ator executa. Entretanto ainda é preciso saber mais detalhes sobre a ação, então todas as palavras após o verbo são consideradas como sendo o complemento da sentença. Desta forma, o software possui os três elementos básicos da sentença (sujeito, verbo e complemento) e com eles é possível fazer algumas inferências:

1. Esta sentença, possivelmente é um caso de uso

2. Caso esta sentença seja um caso de uso, seu ator será o sujeito desta sentença

3. Caso esta sentença seja um caso de uso, o caso de uso será a combinação entre o verbo e o complemento da sentença

Tendo estas inferências em mente, um arquivo *JSON* é criado com os dados obtidos através do processamento do texto e é a saída do módulo de processamento. O controle da aplicação retorna para o módulo gráfico o qual lê o arquivo *JSON* gerado. Neste momento, um algoritmo agrupa todos os sujeitos iguais que foram retornados pelo arquivo *JSON* e

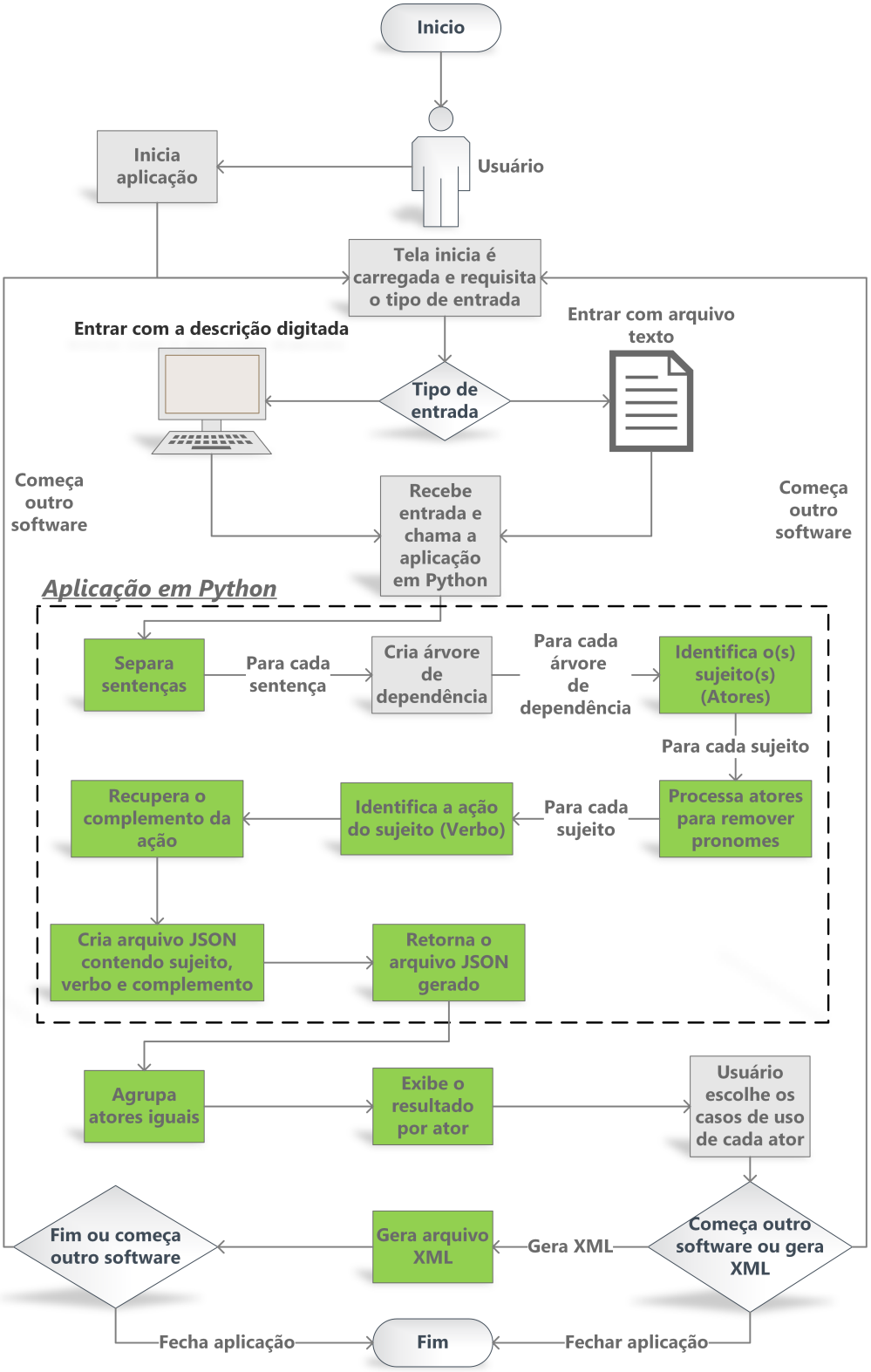


Figura 3.2: Arquitetura do software gerador de possíveis casos de uso

exibe o resultado desta operação. O usuário pode então escolher dentre os possíveis casos de uso os que realmente o são. Terminada a seleção, há ainda a possibilidade de criar

um arquivo *XML* ou começar a análise de outro texto. Caso seja escolhida a análise de outro texto, todo o processo recomeça. Caso o usuário decida gerar o arquivo *XML*, este arquivo é gerado e ele poderá processar outro texto ou terminar a aplicação.

A Figura 3.2 exibe também os módulos (em verde) que foram programados pelo autor deste trabalho. Os outros módulos ou foram programados por outra pessoa ou representam elementos do ambiente.

### 3.1.1 Módulo de processamento

Como especificado no início deste capítulo, o módulo de processamento foi escrito em Python. Este código foi dividido em em três classes:

1. Text
2. Sentence
3. MainClass

A classe Text contém o texto a ser processado, o modelo gramatical do inglês, tanto da biblioteca spaCy, quanto da biblioteca NLTK e uma lista com todas as sentenças processadas pela biblioteca spaCy, o que fornece uma série de *Tokens* (palavras) e as relações entre esses *Tokens* gerando uma árvore de dependência no estilo mostrado pela Figura 3.3 para cada sentença do texto. Após este processamento, cada sentença é adicionada em um vetor de sentenças.

Nesta árvore, cada vértice é uma palavra com sua classe gramatical no contexto da sentença e cada arco direcionado possui a relação que há entre o vértice de chegada em relação ao vértice de saída. Desta forma, no exemplo da Figura 3.3 o arco que liga a palavra *use* com a palavra *client*, que contém a relação *nsubj*, indica que a palavra *client* é um sujeito nominal da palavra *use*.

Ainda no contexto da Figura 3.3, a palavra *use* é chamada de raiz da sentença, porque ela não possui nenhum arco de entrada. Note que pela explicação realizada anteriormente,

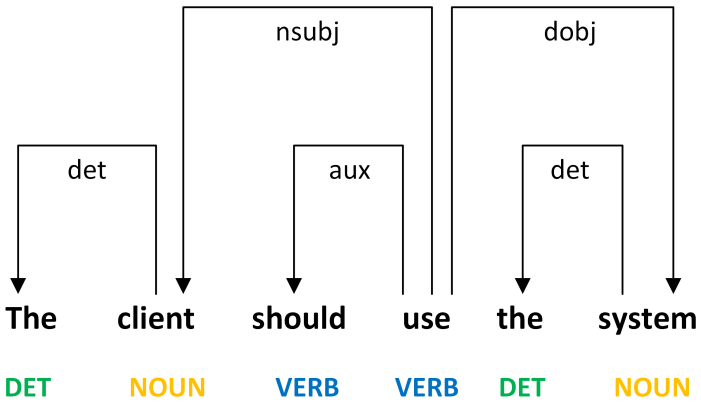


Figura 3.3: Árvore de dependência criada pela biblioteca spaCy

uma árvore de dependência pode ter mais de uma raiz. Boa parte dos processamentos feitos das sentenças utilizados nesta ferramenta utiliza o pressuposto de estes vértices raíz existirem e a partir deles as informações pertinentes ao utilizador serão encontradas e derivadas. Desta forma, no exemplo da Figura 3.3, se queremos todo o sujeito da sentença, basta acessar o vértice raiz (*use*), navegar pelo arco cuja marcação é *nsubj* e pegar toda subárvore do vértice resultante (*client*). Tal subárvore é composta pelos vértices *The* e *client*, vértices estes que configuram nosso sujeito corretamente e, por conseguinte, o nosso ator.

A classe Sentence por sua vez possui os dados derivados de cada árvore de dependência. As informações que são extraídas do texto são: O(s) sujeito(s), o(s) verbo(s) e o complemento da sentença. Para tanto a classe utiliza um conjunto de métodos criados que percorrem a árvore, recuperam os dados pertinentes e os processam para serem melhor interpretados pelo usuário da aplicação.

Por fim, a classe MainClass é a classe de controle que possui uma instância da classe Text e um vetor de elementos da classe Sentence. Além destes dados, a MainClass também é responsável por gerar a saída da aplicação, seja ela na linha de comando de forma simples, tratada ou ainda como um arquivo *JSON*. Neste trabalho a saída em formato *JSON* é a saída padrão para permitir uma interface melhor com a outra parte do software responsável por gerar a interface gráfica. Isto será mais detalhado na Seção 3.1.2.

### 3.1.2 Módulo gráfico

Para a criação da interface gráfica pelo módulo gráfico, foram utilizadas as seguintes bibliotecas:

1. MaterializeCSS[21]

2. ElectronJS[15]

A biblioteca Materialize é uma biblioteca de CSS a qual implementa o Material Design definido pelo Google [20]. Estas diretrizes de design visam melhorar a aparência das aplicações e melhorar de uma forma geral a interface de interação entre os seres humanos e o computador, tudo isso mantendo a simplicidade. Suas principais diretrizes incluem a metaforização de materiais, o uso de cores contrastantes, o foco nas atividades que podem ser realizadas e os movimentos que servem para focar a atenção do usuário e manter a continuidade [20].

A biblioteca ElectronJS é um framework que permite a utilização de HTML, CSS e Javascript para a criação de aplicações para computadores de arquitetura x86. O que está ferramenta faz é a criação de um browser (Chromium) o qual executa apenas a "página"que criamos para ela executar. As principais vantagens deste tipo de ferramenta incluem a capacidade de importar qualquer biblioteca desenvolvida para aplicações web, a portabilidade entre sistemas operacionais (desde que rodem em uma plataforma x86 ou x86-64) e a rapidez com que a aplicação pode ser criada.

O GPCU apresentado neste documento utiliza Javascript não só para criar a aplicação, mas também para melhorar a visualização dos dados pelo usuário, pois é por meio dele que é possível o agrupamento das informações de um mesmo ator, a seleção pelo usuário dos casos de uso que realmente são casos de uso e a exportação dos casos de uso selecionados para um arquivo *XML*.

Embora será detalhada a forma de uso da aplicação na Seção 3.3, vale a observação de como é a interface da aplicação. Ela é completamente definida em um arquivo HTML, chamado *index.html*, o qual recebeu as melhorias de interface providas pela biblioteca de CSS Materialize. Neste arquivo, são definidos três regiões. A primeira região compreende

a seleção do nome do programa Python no sistema do usuário. Esta seção é necessária para manter a portabilidade da aplicação, visto que dependendo do sistema onde a aplicação irá executar, a linguagem Python de versão 3 pode possuir o nome python ou o nome python3.

A segunda região compreende a entrada dos dados, que pode ser de duas formas: ou o usuário seleciona um arquivo com extensão *.txt* ou ele escreve a especificação diretamente no aplicativo. Para ambos os casos há a necessidade de pressionar o botão *Continue*.

Finalmente, com o texto inserido e processado, a terceira região do software aparece. Nela o usuário pode escolher um ator e selecionar apenas os casos de uso que lhe interessam e, após feito isso, exportar estes casos de uso para um arquivo *XML*.

## 3.2 PROCESSO DE CRIAÇÃO

A ideia básica da lógica desta aplicação surgiu da observação de que comumente, em textos de especificação de software, a maioria dos atores são sujeitos de frases e os casos de uso de cada ator são, geralmente, a junção do verbo ao qual o sujeito responde e o complemento deste verbo.

Após esta observação foram definidas as etapas de criação da aplicação, que consistiu de quatro fases:

1. Pesquisa
2. Estudo das bibliotecas
3. Implementação da lógica
4. Implementação do ambiente de interação humano-computador

Na fase de pesquisa, foram investigadas formas de tratar um texto a fim de ser possível identificar cada palavra corretamente em sua classe gramatical e sua semântica no contexto em que se encontra no texto. Para tanto, várias bibliotecas foram encontradas, dentre elas as que se destacaram foram a CoreNLP[1], NLTK[8] e spaCy[10]. Destas as que geram árvores de dependência são a CoreNLP e a spaCy. A biblioteca NLTK possui

módulos úteis como o módulo wordnet, que foi utilizado neste trabalho para permitir a transformação de palavras do singular para o plural e vice versa. Foi escolhido trabalhar com a biblioteca spaCy por que ela é uma das bibliotecas que fazem a geração da árvore de dependência mais rapidamente [5] e atualmente, após melhorias realizadas pelos desenvolvedores da biblioteca, esta biblioteca mantém uma boa taxa de precisão na geração da árvore de dependência, conforme mostra a Figura 3.4[11].

| SISTEMA | LINGUAGEM | PRECISÃO | VELOCIDADE(PPS) |
|---|---|---|---|
| spaCy | Cython | 91,8 | 13963 |
| CoreNLP | Java | 89,6 | 8602 |
| ClearNLP | Java | 91,7 | 10271 |
| MATE | Java | 92,5 | 550 |
| Turbo | C++ | 92,4 | 349 |

Figura 3.4: Comparativo entre spaCy e seus concorrentes[10]

Após ter escolhido as bibliotecas NLTK e spaCy, a segunda fase do desenvolvimento se inciou, a fase de estudo das bibliotecas. Nesta etapa, foram realizados alguns pequenos experimentos sobre os comportamentos das bibliotecas, ou seja, sobre como inicializar as bibliotecas, o que cada função espera como parâmetro, o que retorna e como usá-las para obter o resultado almejado. Esta etapa foi razoavelmente curta, já que serviu apenas como aprendizado básico da API que seria utilizada.

Então foi possível começar a terceira etapa, que implementou o GPCU. A ideia do algoritmo que resolve o problema é transformar o texto em várias árvores de dependência e, para cada uma delas, identificar os sujeitos, verbos e complementos para então derivar os atores e casos de uso. A arquitetura e o funcionamento do software foram melhor detalhados na Seção 3.1.

Após terminada a lógica do software foi implementada uma interface gráfica para a utilização do software. Está interface facilita o uso da aplicação por usuários não familiarizados com interfaces de comando, além de melhorar a visualização dos possíveis casos de uso obtidos.

## 3.3 FORMA DE USO

### 3.3.1 Forma de instalação do software

Para começar a utilizar a aplicação, é necessário possuir conexão com internet e realizar o download da aplicação na página do GitHub[1] e instalar as dependências da aplicação:

1. NodeJS
2. Python 3.X
3. Pip
4. Bibliotecas

Para instalar o item 1, o usuário precisa visitar a página de download do NodeJS[2], e seguir as instruções lá contidas. Feito isso é preciso obter o software do item 2. Para tanto, basta seguir as instruções contidas na página de download do Python[3] referente à versão 3.X. Com as linguagens instaladas, falta então instalar o gerenciador de bibliotecas do Python, o Pip. Para instalar o Pip citado no item 3, é preciso que o usuário baixe o código em Python [4] e rode-o com a instalação do Python 3.X.

Agora que o ambiente básico para o GPCU ser executado está configurado, é necessário instalar as bibliotecas para que a aplicação funcione corretamente. Começando pelas bibliotecas de Python, as bibliotecas que são utilizadas podem ser obtidas da seguinte forma:

```
pip install spacy
pip install inflection
pip install nltk
python -m spacy.en.download --force
```

Para instalar a última biblioteca do Python, é necessário inicializar um terminal no Unix ou prompt de comando no Windows, e entrar com o comando python ou python3,

[1] https://github.com/HenriqueVarellaEhrenfried/TG-Henrique-V-Ehrenfried
[2] https://nodejs.org/en/download/
[3] https://www.python.org/downloads/
[4] https://bootstrap.pypa.io/get-pip.py

depende de como seu sistema estiver configurado. Feito isso você deverá ter em sua tela o ambiente do Python. agora entre com o seguinte código em Python:

```
import nltk
nltk.download()
```

Faça download dos módulos **wordnet** e **wordnetic**. Após esta instalação todo o ambiente necessário para a aplicação Python funcionar estará configurado.

Então será preciso configurar a interface gráfica, isto pode ser feito abrindo o diretório do aplicativo em um terminal ou prompt de comando e digitando *npm install*. Terminada a instalação, todo o ambiente para que a interface gráfica funcione estará instalado. Como já o ambiente do Python já está configurado então é possível executar o aplicativo. Para executar, basta digitar o comando, em um terminal ou prompt de comando no diretório da aplicação, *npm start*.

Este comando carregará a aplicação que já poderá ser utilizada, como será descrito na Seção 3.3.2.

### 3.3.2 Forma de utilização do software

Esta aplicação tem duas formas de ser utilizada:

1. Sem interface gráfica
2. Com interface gráfica

Para executar a aplicação sem interface gráfica, é preciso acessar o diretório da aplicação com um terminal ou prompt de comando e digitar:

```
python TG.py "Software requirement" > saida.json
```

Esse comando irá criar um arquivo *JSON* contendo todos os possíveis casos de uso seguindo o padrão de código mostrado no código 3.1:

```
[
    {
        "id":<numero>,
        "actor":"<texto>",
```

```
        "action":"<texto>",
        "complement":"<texto>"
    },
    ...
    {
        "id":<numero>,
        "actor":"<texto>",
        "action":"<texto>",
        "complement":"<texto>"
    },
]
```

Código 3.1: Modelo de saída *JSON*

Alternativamente, o software poderá ser utilizado por meio da interface gráfica. Neste caso, o usuário precisa iniciar a aplicação conforme explicado no final da Seção 3.3.1. Após iniciada, a aplicação carregará a interface, que é mostrada na Figura 3.5

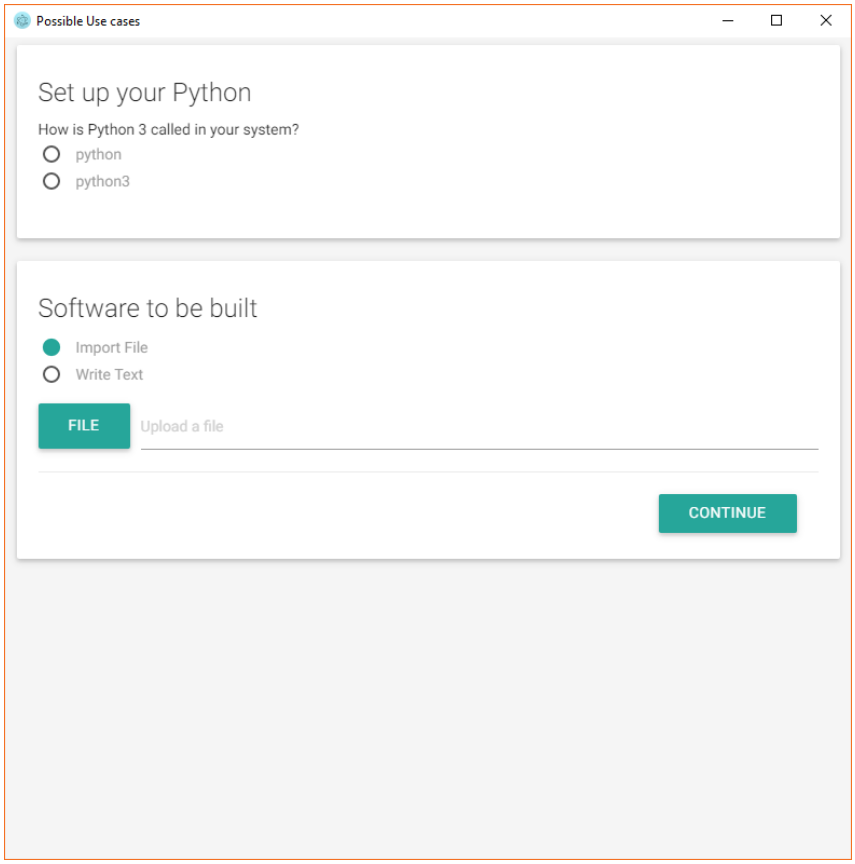


Figura 3.5: Tela inicial da aplicação

Esta interface requisita do usuário três itens: Na primeira seção do software é requisitado como o sistema chamará o Python de versão 3.X. Já na segunda seção, o software requisita como serão inseridos os dados na aplicação, se será via arquivo de texto (no formato *.txt*), ou se o usuário digitará o texto na aplicação. Por fim, ainda na segunda seção, o software requisita o texto que será processado. Preenchido corretamente todos os campos, quando for pressionado o botão *CONTINUE* o software consumirá a entrada de texto e a processará, criando uma terceira seção para mostrar os resultados obtidos, como mostra a Figura 3.6

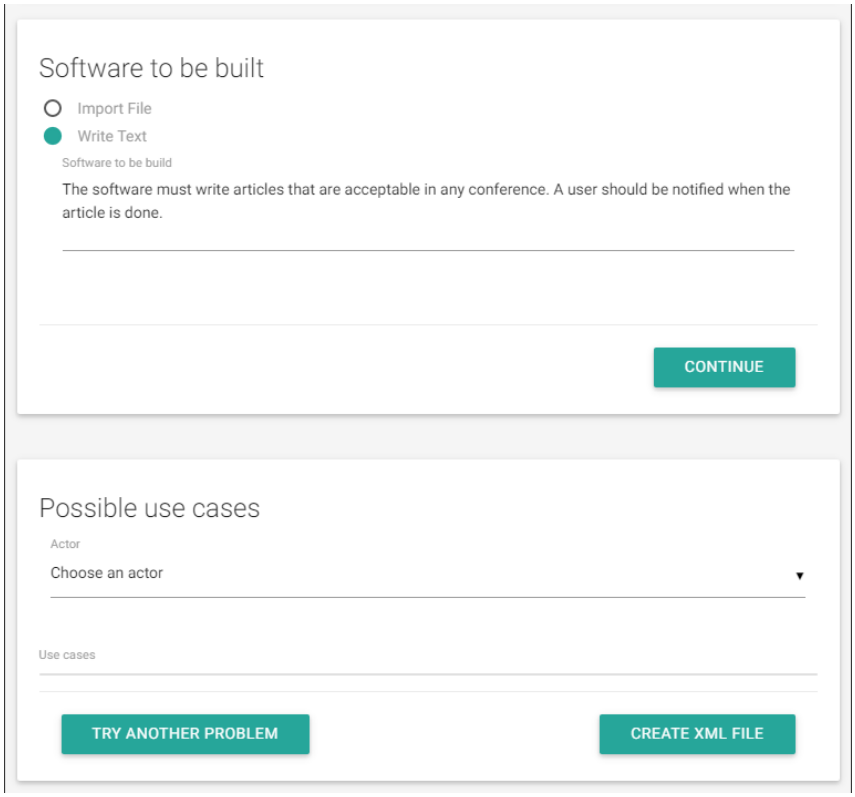


Figura 3.6: Texto após ser processado

Após o texto ser processado o usuário pode selecionar um dos possíveis atores que são listados no menu *Actor* para poder visualizar os casos de uso assosciados aom mesmo. A Figura 3.7 mostra um exemplo.

Quando um ator é selecionado, seus possíveis casos de uso são exibidos em uma lista de opções passíveis de seleção, permitindo destacar os casos de uso pertinentes, como a Figura 3.8 exemplifica.

Cada possível caso de uso listado pode ser selecionado para ser promovido a um caso de

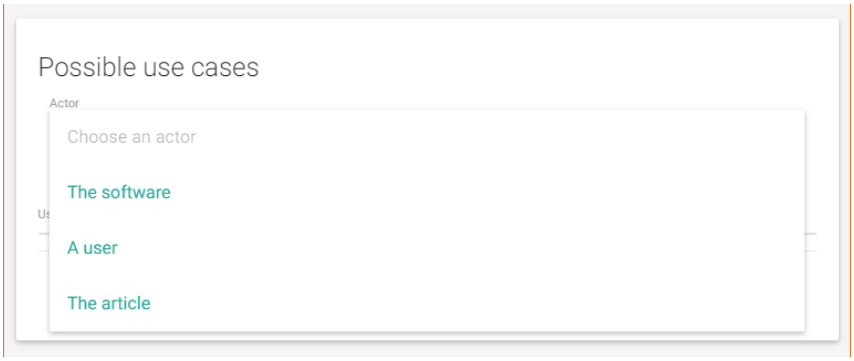


Figura 3.7: Menu de seleção de ator

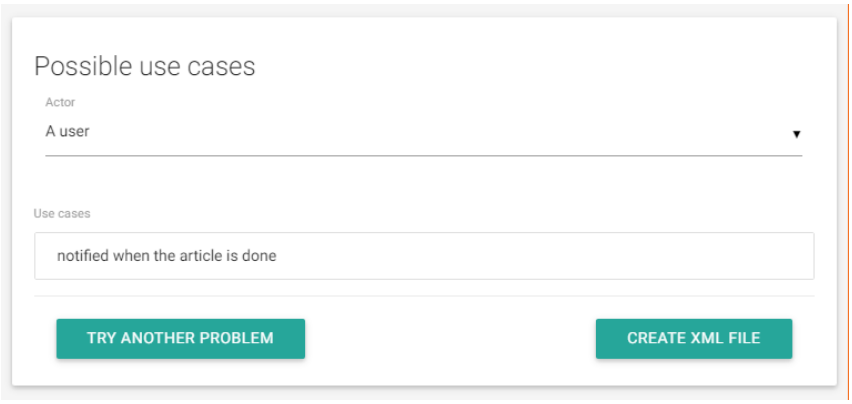


Figura 3.8: Aplicação com o ator selecionado

uso, como a Figura 3.9 exemplifica. Com os casos de uso selecionados é possível pressionar o botão *CREATE XML FILE* para gerar um arquivo XML contendo todos os atores e seus respectivos casos de uso, ou então, a qualquer momento, pode ser selecionado o botão *TRY ANOTHER PROBLEM* para reiniciar o texto de especificação de software.

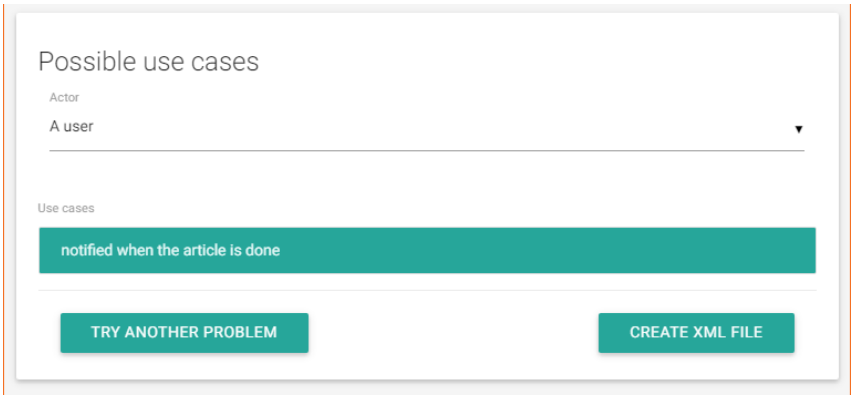


Figura 3.9: Caso de uso selecionado

Quando o usuário requisita a geração de uma arquivo XML, uma nova janela é aberta para que ele escolha o lugar em que deseja salvar o arquivo e o seu nome. Após escolher

o caminho e o nome, um arquivo com a estrutura mostrada no Código 3.2 é salvo no diretório especificado.

```
<Software>
  <Actor>
    Actor1
  </Actor>
  <UseCase>
    something 1
  </UseCase>
  <Actor>
    Actor2
  </Actor>
  <UseCase>
    something 2
  </UseCase>
  <UseCase>
    something 3
  </UseCase>
    ...
  <Actor>
    the alarm
  </Actor>
  ...
</Software>
```

Código 3.2: Problema para criação de diagrama de caso de uso

## 3.4 CONCLUSÃO

Neste capítulo foi explicado o funcionamento do módulo de processamento e do módulo gráfico. Basicamente, o módulo gráfico recebe uma entrada e a envia para o módulo de processamento. No módulo de processamento o texto é processado utilizando uma técnica de processamento de linguagem natural e a saída deste processamento compõe um arquivo *JSON*. Este por sua vez será utilizado pelo módulo gráfico para a composição da resposta.

A resposta que o módulo gráfico produz é composta pelo agrupamento de atores iguais com seus respectivos possíveis casos de uso.

No final deste capítulo foi documentado como o software pode ser instalado e como é sua forma de utilização. No capítulo que segue será exposta a aplicação em execução com um problema real. Depois de executado o problema, os resultados obtidos serão avaliados.

# CAPÍTULO 4

# EXEMPLO PRÁTICO

Neste capítulo será mostrado um caso real de uso da ferramenta, e todas as etapas inerentes a sua utilização.

## 4.1 UTILIZANDO A FERRAMENTA

Para exemplificar o uso do GPCU, ele será executado com um texto que especifica um despertador que tem como especificação a seguinte descrição [19]:

```
The clock shows the time of day. Using buttons, the user can set the hours
    and minutes fields individually, and choose between 12 and 24-hour
    display. It is possible to set one or two alarms. When an alarm fires,
    it will sound some noise. The user can turn it off, or choose to 'snooze
    '. If the user does not respond at all, the alarm will turn off itself
    after 2 minutes. Snoozing means to turn off the sound, but the
    alarm will fire again after some minutes of delay. This 'snoozing time'
    is pre-adjustable.
```

Código 4.1: Problema para criação de diagrama de caso de uso

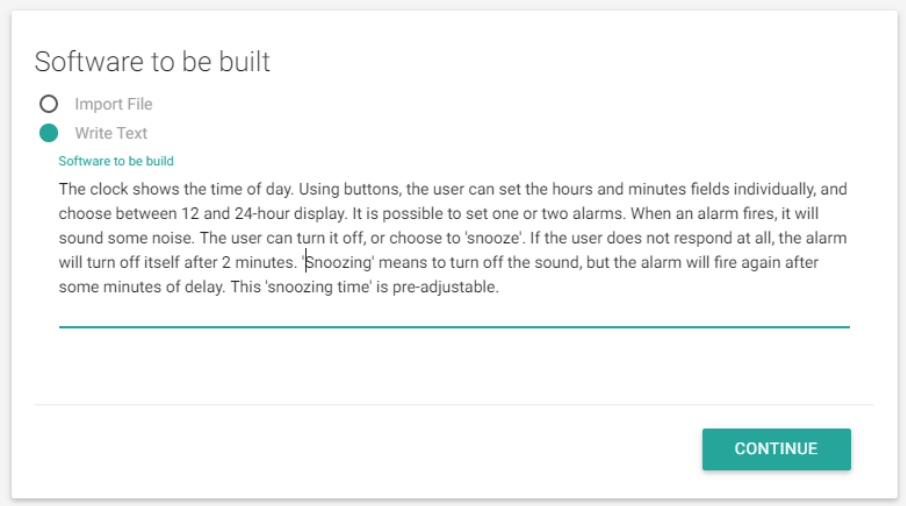


Figura 4.1: Exemplo de entrada de texto

Então, após iniciarmos a aplicação entramos com o texto, assim como mostra a Figura 4.1 e pressionamos o botão *CONTINUE*

O pressionamento deste botão faz com que o texto seja processado conforme foi descrito na Seção 3.1. O resultado da execução resulta no *JSON* mostrado no código 4.2.


```
[
    {
        "actor": "The clock",
        "action": "shows",
        "id": 1,
        "complement": "the time of day"
    },
    {
        "actor": "the user",
        "action": "set",
        "id": 2,
        "complement": "the hours and minutes fields individually , and
           choose between 12 and 24-hour display"
    },
    {
        "actor": "the user",
        "action": "is",
        "id": 3,
        "complement": "possible to set one or two alarms"
    },
    {
        "actor": "the user",
        "action": "sound",
        "id": 4,
        "complement": "some noise"
    },
    {
        "actor": "The user",
        "action": "turn",
        "id": 5,
```

```
        "complement": "it off , or choose to ' snooze '"
    },
    {
        "actor": "the user",
        "action": "respond",
        "id": 6,
        "complement": "at all"
    },
    {
        "actor": "the alarm",
        "action": "turn",
        "id": 7,
        "complement": "off itself after 2 minutes"
    },
    {
        "actor": "the alarm",
        "action": "fire",
        "id": 8,
        "complement": "again after some minutes of delay"
    },
    {
        "actor": "This ' snoozing time '",
        "action": "is",
        "id": 9,
        "complement": "pre - adjustable"
    }
]
```


Código 4.2: Arquivo *JSON* criado pelo software com possíveis casos de uso

Este *JSON* é tratado como descrito na Seção 3.1 e produz o resultado descrito nas Figuras 4.2 e 4.3

A Figura 4.2 exibe todos os possíveis atores encontrados no texto que foi inserido na ferramenta proposta por este trabalho.

Cada possível ator possui ao menos um possível caso de uso. O resultado completo

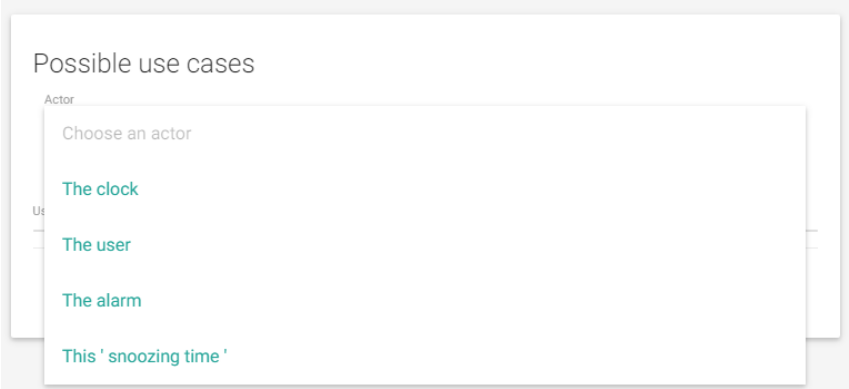


Figura 4.2: Atores encontrados pela ferramenta

pode ser visto na enumeração a seguir. Cada ator exibe seus possíveis casos de uso exatamente da mesma forma que é mostrado na figura 4.3, a qual exibe todos os possíveis casos de uso encontrados para o ator *User*.

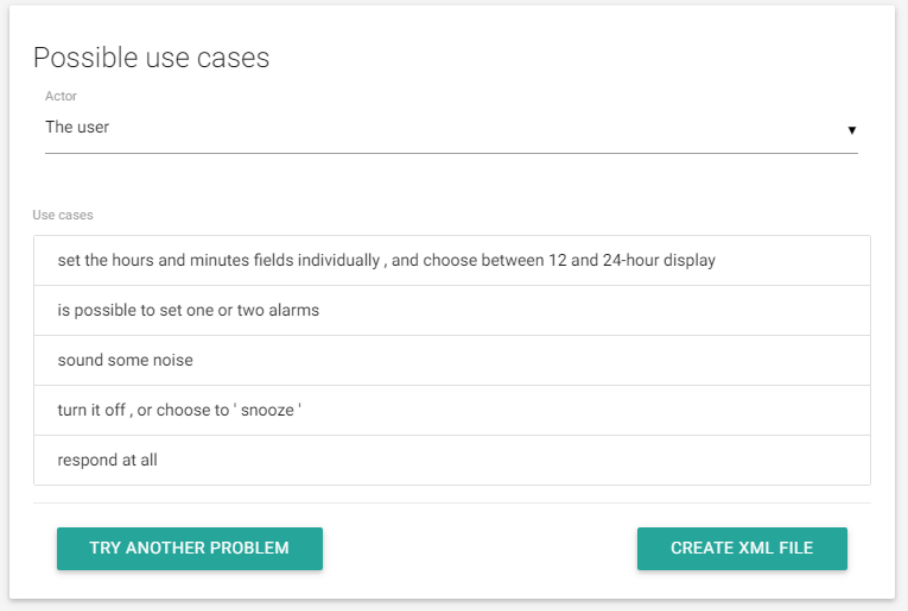


Figura 4.3: Atores encontrados pela ferramenta

1. *The clock*
   (a) *shows the time of day*
2. *The user*
   (a) *set the hours and minutes fields individually , and choose between 12 and 24-hour display*

(b) *is possible to set one or two alarms*

(c) *sound some noise*

(d) *turn it off , or choose to ' snooze '*

(e) *respond at all*

3. *The alarm*

(a) *turn off itself after 2 minutes*

(b) *fire again after some minutes of delay*

4. *This 'snoozing time'*

(a) *is pre - adjustable*

A enumeração acima representa a saída obtida após executar o problema 4.1 no software proposto por este trabalho. Ela está estruturada de forma que os itens sob uma numeração são os atores encontrados e os itens sob enumeração por letras são os possíveis casos de uso obtidos.

## 4.2 ANÁLISE DOS RESULTADOS

Analisando os resultados obtidos pelo software e enumerados com a enumeração da sub-seção anterior, é possível reparar que os maiores erros do software ocorrem com os possíveis casos de uso 2c e 2e. O possível caso de uso 2c na realidade pertence ao ator *The alarm* e o possível caso de uso 2e perdeu seu adverbio de negação tampouco possui um contexto.

O diagrama de casos de uso proposto como solução para este problema por seu autor [19] é o diagrama representado na Figura 4.4

O diagrama que pode ser construído com os casos de uso gerados pelo software proposto, após a exclusão dos casos de uso 2c e 2e é o mostrado na Figura 4.5

Como exibido na Figura 4.5, apenas cinco casos de uso são exatamente iguais entre as Figuras 4.4 e 4.5, são eles *Set time*, *Set display-mode Set alarm*, *Turn off* e *Snooze*. Há

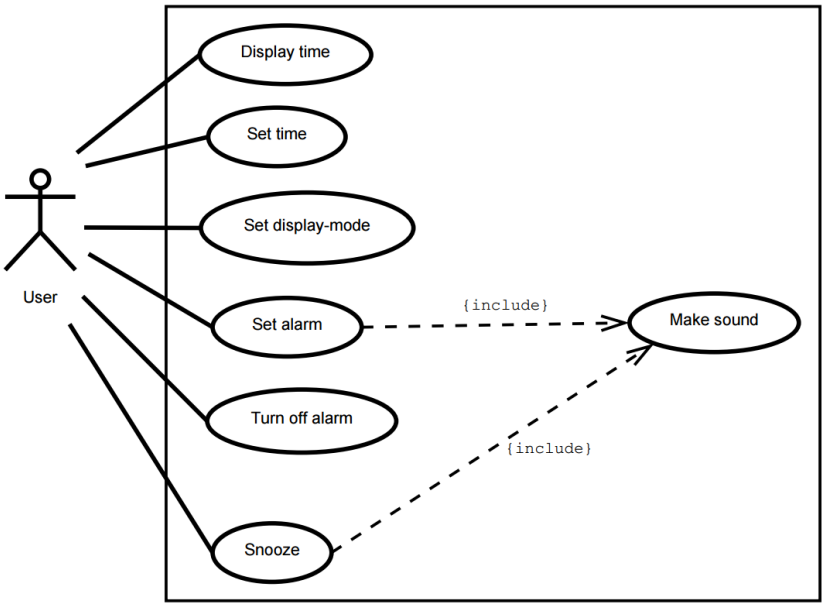


Figura 4.4: Diagrama de casos de uso que representa o problema. Fonte: `http://www.cs.uu.nl/docs/vakken/mso/1011/asgs/usecase.sol.pdf` - Wishnu Prasetya[19]

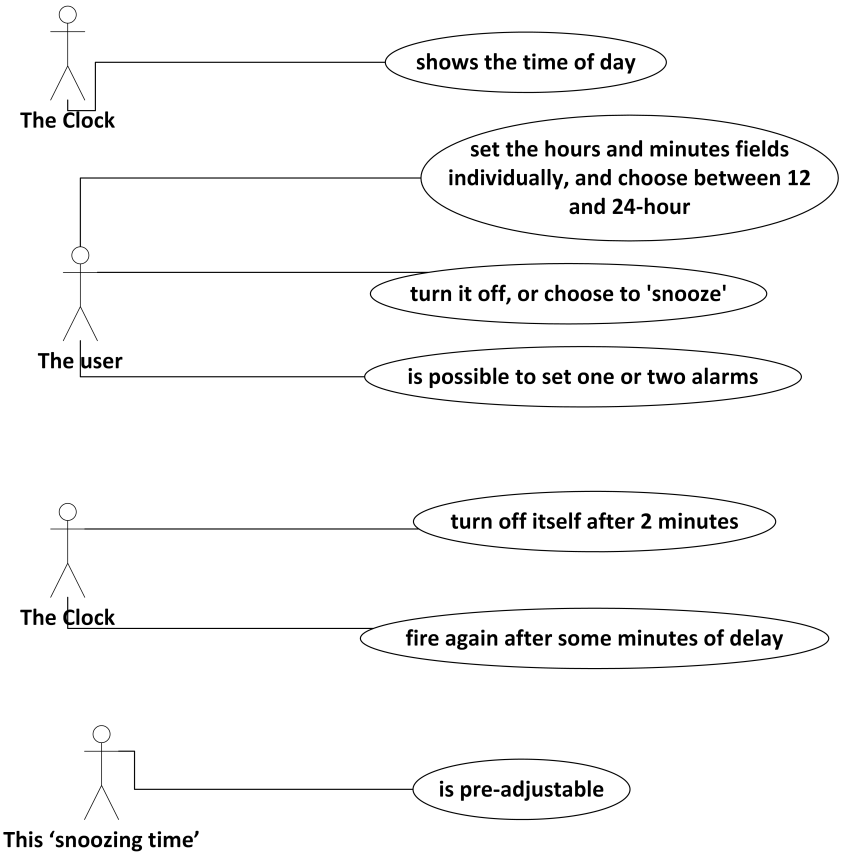


Figura 4.5: Diagrama de casos de uso que representa o problema criado com os dados coletados pelo software criado

apenas dois casos de uso que não estão exatamente iguais nas duas soluções, já que o caso de uso *Display time* é atribuído ao ator *The clock* ao invés de *The User*, mas não está completamente errado, dependendo do contexto do software, quem mostra o horário não é o usuário, mas sim o relógio, entretanto, para o contexto desta aplicação encontra-se incorreto. O outro caso de uso não representado é o *Make sound*, o qual foi atribuído erroneamente ao ator *The user* e portanto excluído desta seleção.

Um fato que deve ser considerado é que dos cinco casos de uso corretos, dois deles foram representados em conjunto com outros dois casos de uso, são eles *Set time* o qual foi representado com *Set display-mode* e *Turn off alarm* que foi representado em conjunto com *Snooze*. Embora estes conjuntos representem ações similares, deveriam ter sido representadas separadamente para facilitar na análise do problema e melhorar a qualidade do diagrama de casos de uso e do software que será produzido a partir dele.

Outro ponto que deve ser considerado é que o GPCU pode interpretar como atores palavras que não deveriam ser atores. Um exemplo disto é o ator *clock*, o qual é o sistema que está sendo descrito e não um ator. Uma forma de resolver este problema é a utilização de contexto. Para tanto seria preciso especificar um padrão de texto, entretanto esta abordagem poderia acabar saindo do escopo da proposta deste trabalho. Outra forma, esta dentro da proposta do trabalho, para contornar o problema seria permitir que o usuário mova os possíveis casos de uso de um ator para outro, desta forma, o usuário poderá conversar com o interessado pelo software para entender qual ator realmente é o responsável por qual caso de uso.

## 4.3 CONCLUSÃO

Neste capítulo o aplicativo foi mostrado em funcionamento com um problema real. Após a comparação do resultado do software com o resultado que um humano criou pode-se concluir que o software funciona bem, apenas precisa de alguns refinamentos pontuais em sentenças mais complexas. Estes refinamentos incluem políticas de decisão sobre dividir ou não um caso de uso em mais de um e a qual ator um caso de uso realmente pertence.

O próximo capítulo apresentará as conclusões que puderam ser obtidas a partir deste trabalho, as contribuições que este trabalho gerou e os trabalhos futuros.

# CAPÍTULO 5

# CONCLUSÃO

A ferramenta proposta neste trabalho teve como objetivo o processamento de linguagem natural para a elicitação automática de casos de uso. Para tanto ela recebe um texto que especifica um software, o processa e fornece os possíveis casos de uso. Para fazer isso foi utilizada a técnica de análise sintática baseada em árvores de dependência. Após a criação das árvores de dependência um algoritmo é executado para recuperar os possíveis atores e os possíveis casos de uso.

Os objetivos deste trabalho foram atingidos, já que a saída da aplicação são atores e casos de uso obtidos através do processamento de linguagem natural. Os resultados atingidos também são consistentes com os resultados de um especialista da área, o problema porém, é a falta de um contexto para o geração dos casos de uso, o que possibilita a geração de um caso de uso diferente do esperado.

A ferramenta criado auxilia na compreensão do problema do usuário e gera uma possível modelagem para o diagrama de casos de uso seguindo o padrão UML, atendendo, assim, as finalidades deste projeto. O resultado do processamento do texto, conforme avaliado no Capítulo 4, é satisfatório, mesmo não introduzindo nenhum algoritmo avançado de inteligência artificial.

Entretanto para que o usuário tenha uma boa resposta do sistema ele precisa interagir com a saída proposta para escolher quais casos de uso realmente são casos de uso e quais não são. Isso ocorre porque o processamento de linguagem natural traz consigo algumas dificuldades, como a dependência de contexto das palavras e a dependência do idioma. Desta forma os poucos incidentes relatados no capítulo 4 como a junção de dois casos de uso em um e a perda do advérbio de negação fazem parte das limitações que esta aplicação possui.

Para melhorar os resultados, em trabalhos futuros poderão ser utilizado algoritmos de

aprendizagem de máquina para aperfeiçoar os detalhes que apenas com processamento de linguagem natural não foram possíveis obter, como os problemas que houve no exemplo prático prático do Capítulo 4.

# REFERÊNCIAS

[1] S. Arnaud. Corenlp. http://stanfordnlp.github.io/CoreNLP/, 2010. Acessado no dia 19/09/2016.

[2] S. Bird, E. Klein, e E. Loper. *Natural Language Processing with Python*, volume 1. O'Reilly Media, 2009. Disponível em http://www.nltk.org/book/. Acessado em 02/11/2016.

[3] G. Booch, J. Rumbaugh, e I. Jacobson. *UML Guia do usuário*, volume 1. Editora Campus, 2000.

[4] P. Bourque e R. E. Fairley. *Guide to the Software Engineering Body of Knowledge*, volume 1. IEEE Computer Society, 2014. Versão 3.0, Disponível em https://www.computer.org/web/swebok/v3.

[5] J. D. Choi, J. Tetreault, e A. Stent. It depends: Dependency parser comparison using a web-based evaluation tool. 2015. http://aclweb.org/anthology/P/P15/P15-1038.pdf Acessado no dia 17/09/2016. Relatório técnico.

[6] A. Cockburn. *Writing Effective Use Cases*, volume 3. Addison-Wesley, 2001.

[7] D. K. Deeptimahanti e R. Sanyal. Semi-automatic generation of uml models from natural language requirements. 2011.

[8] D. Garrette, P. Ljunglöf, J. Nothman, e S. Bird. Nltk. http://www.nltk.org/, 2005. Acessado no dia 19/09/2016.

[9] R. Helm e D. Wildt. *Histórias de Usuário*, volume 1. Lucas Engel, 2014. Disponível em http://www.wildtech.com.br/historias-de-usuario/. Acessado em 11/11/2016.

[10] M. Honnibal e M. Johnson. spacy. http://spacy.io/, 2014. Acessado no dia 17/09/2016.

[11] M. Honnibal e M. Johnson. An improved non-monotonic transition system for dependency parsing. 2015. https://aclweb.org/anthology/D/D15/D15-1162.pdf Acessado no dia 17/09/2016. Relatório técnico.

[12] A. Ittoo, L. Maruster, H. Wortmann, e G. Bouma. Textractor : A framework for extracting relevant datasets. 2010.

[13] D. Jurafsky e J. H. Martin. *Speech and Language Processing: An introduction to natural language processing, computational linguistics, and speech recognition*, volume 2. Pearson Prentice Hall, 2008.

[14] C. Larman. *Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and the Unified Process*, volume 1. Prentice Hally, 1998.

[15] J. Lord. Electronjs. http://electron.atom.io/, 2013. Acessado no dia 19/09/2016.

[16] I. A. Melčuk. *Dependency syntax : theory and practice*, volume 1. State University of New York Press, 1987.

[17] A. R. Pimentel. Projeto de software usando a uml , year = 2015, volume = 1, pages = 13-20, note = Disponível em http://www.inf.ufpr.br/andrey/ci162/apostilaUml.pdf. Acessado em 28/11/2016.

[18] D. Popescu, S. Rugaber, N. Medvidovic, e D. M. Berry. Reducing ambiguities in requirements specifications via automatically created object-oriented models. 2008.

[19] W. Prasetya. Problemas casos de uso. http://www.cs.uu.nl/docs/vakken/mso/1011/asgs/usecase.sol.pdf, 2010. Exercício 05, acessado no dia 05/10/2016.

[20] G. D. Team. Google material design. https://material.google.com/, 2014. Acessado no dia 19/09/2016.

[21] A. Wang, A. Chang, A. Mark, e K. Louie. Materializecss. http://materializecss.com/, 2014. Acessado no dia 19/09/2016.