Universidade de São Paulo
Faculdade de Filosofia, Letras e Ciências Humanas
Departamento de Letras Clássicas e Vernáculas

Luiz Henrique Lima Veronesi

eDictor: da plataforma para a nuvem

São Paulo

2014

Luiz Henrique Lima Veronesi

eDictor: da plataforma para a nuvem

v.1

Dissertação apresentada ao
Departamento de Letras Clássicas e Vernáculas
da Faculdade de Filosofia, Letras e Ciências Humanas
da Universidade de São Paulo
para obtenção do título de mestre

Orientadora: Profa. Dra. Maria Clara Paixão de Sousa

São Paulo

2014

Nome: VERONESI, Luiz Henrique Lima

Título: eDictor: da plataforma para a nuvem

Dissertação apresentada ao
Departamento de Letras Clássicas e
Vernáculas da Faculdade de Filosofia,
Letras e Ciências Humanas da
Universidade de São Paulo
para obtenção do título de mestre

Aprovado em:

Banca Examinadora

Profa. Dra.:	Maria Clara Paixão de Sousa	Instituição:	Universidade de São Paulo
Julgamento:		Assinatura:	
Prof. Dra.:	Charlotte Marie Chambelland Galves	Instituição:	Universidade de Campinas
Julgamento:		Assinatura:	
Prof. Dr.:	Fabio Natanael Kepler	Instituição:	Universidade Federal do Pampa
Julgamento:		Assinatura:	

A point is that of which there is no part.
Euclid, Elements of Geometry.

No princípio era o **Verbo**, e o **Verbo** estava com Deus, e o **Verbo** era Deus.
O Evangelho de João, 1:1, Novo Testamento.

In the beginning was the **Word**, and the **Word** was with God,
and the **Word** was God.
The Gospel of John, 1:1, New Testment.

Nor does any divine or linguistic force prevent two characterizations from being inconsistent.
Ronald W. Langacker, Cognitive Grammar, a basic introduction.

RESUMO

VERONESI, L. H. L. **eDictor: da plataforma para a nuvem**. 2014. 122 f. Dissertação (Mestrado) - Faculdade de Filosofia, Letras e Ciências Humanas, Universidade de São Paulo, São Paulo, 2014.

Neste trabalho, apresentamos uma nova proposta para edição de textos que fazem parte de um corpus eletrônico. Partindo do histórico de desenvolvimento do corpus Tycho Brahe e da ferramenta eDictor, propõe-se a análise de todo o processo de trabalho de criação de um corpus para obter uma forma de organização da informação mais concisa e sem redundâncias, através do uso de um único repositório de informações contendo os dados textuais e morfossintáticos do texto. Esta forma foi atingida através da criação de uma estrutura de dados baseada em unidades mínimas chamadas *tokens* e blocos de unidades chamados *chunks*. A relação entre os *tokens* e os *chunks*, da forma como considerada neste trabalho, é capaz de guardar a informação de como o texto é estruturado em sua visualização (página, parágrafos, sentenças) e na sua estrutura sintática em árvores. A base de análise é composta por todos os arquivos pertencentes ao catálogo de textos do corpus Tycho Brahe. Através desta análise, foi possível chegar a elementos genéricos que se relacionam, desconstruindo o texto e criando uma relação de pontos de início e fim relativos às palavras (*tokens*) e não seguindo sua forma linear. A introdução do conceito de orientação a objetos possibilitou a criação de uma relação entre unidades ainda menores que o *token*, os *split tokens* que também são tokens, pois herdam as características do elemento mais significativo, o *token*. O intuito neste trabalho foi buscar uma forma com o menor número possível de atributos buscando diminuir a necessidade de se criar atributos específicos demais ou genéricos de menos. Na busca deste equilíbrio, foi verificada a necessidade de se criar um atributo específico para o *chunk* sintático, um atributo de nível que indica a distância de um nó da árvore para o nó raiz. Organizada a informação, o acesso a ela se torna mais simples e parte-se para definição da interface do usuário. A tecnologia web disponível permite que elementos sejam posicionados na tela reproduzindo a visualização que ocorre no livro e também permite que haja uma independência entre um e outro elemento. Esta independência é o que permite que a informação trafegue entre o computador do usuário e a central de processamento na nuvem sem que o usuário perceba. O processamento ocorre em *background*, utilizando tecnologias assíncronas. A semelhança entre as tecnologias *html* e *xml* introduziu uma necessidade de adaptação da informação para apresentação ao usuário. A solução apresentada neste trabalho é pensada de forma a atribuir aos *tokens* informações que indiquem que eles fazem parte de um *chunk*. Assim, não seriam as palavras que pertencem a uma sentença, mas cada palavra que possuiria um pedaço de

informação que a faz pertencente à sentença. Esta forma de se pensar muda a maneira como a informação é exibida.

Palavras-chave: linguística de corpus, corpus eletrônico, corpus anotado, edição filológica digital, linguística computacional, arquitetura web.

ABSTRACT

VERONESI, L. H. L. **eDictor: from platform to the cloud**. 2014. 122 f. Thesis (Masters) - Faculdade de Filosofia, Letras e Ciências Humanas, Universidade de São Paulo, São Paulo, 2014.

In this work, we present a new proposal for text edition organized under an electronic corpus. Starting from Tycho Brahe corpus development history and the eDictor tool, we propose to analyze the whole work process of corpus creation in order to obtain a more concise and less redudant way of organizing information by using a single source repository for textual and morphosyntactic data. This single source repository was achieved by the creation of a data structure based on minimal significative units called *tokens* and grouping units named *chunks*. The relationship between *tokens* and *chunks*, in the way considered on this work, allows storage of information about how the text is organized visually (pages, paragraphs, sentences) and on how they are organized syntactically as represented by syntactic trees. All files referred to the Tycho Brahe corpus catalog were used as base for analysis. That way, it was possible to achieve generic elements that relate to each other in a manner that the text is deconstructed by using relative pointers to each *token* in the text instead of following the usual linear form. The introduction of oriented-object conception made the creation of relationship among even smaller units possible, they are the *split tokens*, but *split tokens* are also *tokens*, as they inherit characteristics from the most significative element (the *token*). The aim here was being "attributeless" avoiding the necessity of too specific or too vague attributes. Looking for that balance, it was verified the necessity of creating a level attribute for syntactic data that indicates the distance of a tree node to its root node. After information is organized, access to it become simpler and then focus is turned to user-interface definition. Available web technology allows the use of elements that may be positioned on the screen reproducing the way the text is viewed within a book and it also allows each element to be indepedent of each other. This independence is what allows information to travel between user computer and central processing unit at the cloud without user perception. Processing occurs in background using asynchronous technology. Resemblance between *html* and *xml* introduced a necessity of adaption to present the information to the user. The adopted solution in this work realizes that *tokens* must contain the information about the *chunk* to which they belong. So this is not a point of view where words belong to sentences, but that each word have a piece of information that make them belong to the sentence. This subtile change of behavioring changes the way information is displayed.

Keywords: corpus linguistics, electronic corpus, annotated corpus, philological digital edition, computational linguistics, web architecture.

LISTAS DE FIGURAS

Figura 1 - processo do projeto Tycho Brahe 5
Figura 2 – exemplo de arquivo xml pré-eDictor 8
Figura 3 – trecho de arquivo *pos* 12
Figura 4 – estrutura do *xml* pré-eDictor 25
Figura 5 – diagrama de classes (marcações) do eDictor plataforma 26
Figura 6 – diagrama do eDictor plataforma em formato de cebola 27
Figura 7 - chunk / token 30
Figura 8 - trecho do xml do eDictor plataforma 31
Figura 9 – introdução da quebra na estrutura 33
Figura 10 – o *token* é dividido e o *split token* entra na estrutura 36
Figura 11 – introdução do *joined token*, o *token* juntado, na estrutura 42
Figura 12 - estrutura orientada a objetos 47
Figura 13 - Anotação sintática da oração 17.202 59
Figura 14 – árvore binária de n níveis com n categorias XP e n *tokens* 62
Figura 15 - árvore binária de n níveis com n categorias XP e n *tokens* 63
Figura 16 - árvore binária de n níveis com n categorias XP e n *tokens* 64
Figura 17 - html visualizado no navegador 72
Figura 18 – trecho do código fonte do html da figura anterior 72

LISTAS DE TABELAS

Tabela 1 – exemplos de *xml* para parágrafos e sentenças 18
Tabela 2 – exemplos de *xml* com marcas de quebra 19
Tabela 3 – exemplo de uma palavra em *xml* 38
Tabela 4 – exemplo de uma palavra em *xml* com uma marca de quebra 38
Tabela 5 – exemplo da estrutura em *xml* 52
Tabela 6 - chunks da árvore binária da Figura 14 62
Tabela 7 - chunks da árvore binária da Figura 15 63
Tabela 8 - chunks da árvore binária da Figura 16 64

SUMÁRIO

1. Introdução 1
2. O Corpus Tycho Brahe e o eDictor 4
   2.1. O Corpus Tycho Brahe 4
   2.2. Breve histórico do eDictor 13
3. Organização da informação 22
   3.1. Primeira parte: análise da estrutura do eDictor 22
   3.2. Segunda parte: análise da estrutura a partir das operações de edição . 34
   3.3. Terceira parte: orientando a objetos 44
   3.4. Quarta parte: atributos lógicos e físicos de *chunks* 52
   3.5. Quinta parte: a estrutura sintática e a tridimensionalidade dos *chunks* 59
4. eDictor: da plataforma para a nuvem 65
   4.1. Primeira parte: introdução 66
   4.2. Segunda parte: da estrutura para um formato de marcação 70
   4.3. Terceira parte: a interface de edição 74
   4.4. Quarta parte: a serialização e o abandono do *xml* 78
5. Conclusão 80
6. Referências bibliográficas 90
7. Apêndices 91
   7.1. APÊNDICE A: o processo operacional do Corpus Tycho Brahe 91
   7.2. APÊNDICE B: manual de utilização do eDictor 93
   7.3. APÊNDICE C: código-fonte do algoritmo de reversão dos chunks.... 109

1. Introdução

O objetivo deste trabalho é descrever a elaboração e desenvolvimento da versão web da ferramenta para edição textual conhecida como eDictor e apresentar uma estrutura capaz de armazenar em somente um arquivo as informações textuais, morfológicas e sintáticas. Esta ferramenta foi proposta em 2007 (Paixão de Sousa e Kepler, 2007) como solução para edição de textos no projeto do corpus Tycho Brahe (Galves & Faria, 2010; Paixão de Sousa & Trippel, 2006). Um dos destaques do Corpus Tycho Brahe em relação a outros é o de ser um corpus sintaticamente anotado. Assim, vários níveis de informação coexistem em um mesmo corpus: textual, morfológico e sintático.

O eDictor foi concebido como um editor de arquivos *xml*[1], removendo do usuário a necessidade de se realizar a edição direta (via editores de texto) neste tipo de arquivo (Paixão de Sousa & Trippel, 2006).

Dentro do escopo de desenvolvimento da ferramenta, levou-se em consideração todo seu histórico, aproveitando as concepções estabelecidas em função do sucesso obtido pelas versões previamente desenvolvidas. Um dos grandes passos do eDictor foi levar a edição do texto ao nível da palavra permitindo que a cada uma pudessem ser atribuídos dados independentemente do texto como um todo.

Havia, portanto, a necessidade da integração dos diferentes níveis de informação do texto em somente um único repositório comum como se pode ver em Paixão de Sousa & Trippel, 2004.

[1] W3C, http://www.w3.org/XML/

A fim de realizar a integração descrita acima, foi desenvolvida uma estrutura como forma de centralizar as informações eliminando a redundância de dados e criando uma consistência para permitir a análise mais profunda e eficiente das informações. Esta estrutura serviu de base para o desenvolvimento de todas as ferramentas que compõem a suíte de trabalho do projeto.

Através desta estrutura, foi possível contemplar uma estratégia de "desintegração" do texto, passando de uma organização linear sequencial para uma separação dos itens lexicais referenciando-os em blocos posicionais. A identificação do texto como texto só se dá novamente através da "rematerialização" do texto para compreensão como texto. Caso contrário, seria somente um amontoado de referências impossíveis para leitura humana convencional, isto é, a dependência da ferramenta se torna intrínseca para que o texto possa ser compreendido em função de sua lógica organizacional.

Uma questão relevante para a criação da estrutura era dada pela necessidade de se criar blocos intercalados, isto é, poder relacionar de forma não-hierárquica ou aninhada elementos de um texto (blocos dentro de blocos), por exemplo, uma sentença (também entendida como um bloco) pode ser iniciada em uma página (outro bloco) e ser terminada em outra página (outro bloco também). Esta era uma questão para a qual não havia solução no modelo anterior (Paixão de Sousa & Trippel, 2006) e que passou a ser possível a partir do momento em que foi adotada a relação entre *token* e *chunk* de maneira não hierárquica, ou seja, através de referências de posição.

Neste trabalho, tecnicamente, trato a palavra como *token* e o bloco como *chunk*.

Criada esta base para a organização informacional, passa-se então ao desenvolvimento de uma ferramenta web capaz de proporcionar todas as funcionalidades atuais, superá-las em eficiência e ainda possibilitar a integração de novos conceitos para aumentar a produtividade da elaboração dos textos dentro do projeto (e de todos os outros que possam vir a utilizá-la). Dentro destes conceitos, o mais fundamental, sem dúvida, é o relacionado à capacidade de ser *multi-corpora*, ou seja, estar preparado para trabalhar com diversos corpora, cada um com suas especificidades e necessidades, dentro de uma ferramenta versátil e moderna.

Esta dissertação possui a seguinte estrutura: um capítulo sobre o Corpus Tycho Brahe descrevendo seu processo de funcionamento levantado problemas tanto técnicos quanto operacionais e um breve histórico do eDictor colocando os pontos problemáticos que foram abordados anteriormente e atacados diretamente na solução deste trabalho. Um capítulo é dedicado à estrutura criada abordando noções básicas de linguagens de marcação e orientação a objetos, passando pelas análises realizadas até concluir a apresentação da estrutura. O último capítulo é dedicado à forma como foi realizada a passagem do eDictor de sua versão plataforma para a web, capítulo este que dá nome ao trabalho.

Para isto, primeiramente descrevo toda a problemática para organização da informação que vinha sendo adotada no projeto Tycho Brahe para armazenamento dos textos e como foi necessário realizar uma adequação do

processo desenvolvendo uma nova estrutura de dados para integração das informações textuais, morfológicas e sintáticas.

2. O Corpus Tycho Brahe e o eDictor

Neste capítulo aborda-se o corpus Tycho Brahe e o seu processo de trabalho e como a ferramenta eDictor se integra a ele.

O desenvolvimento deste projeto foi realizado a partir da análise de todos os textos que compõem o corpus através da análise (via programação) dos arquivos que guardam a informação dos textos.

2.1. O Corpus Tycho Brahe

A grosso modo, um corpus é uma coleção de documentos: livros, jornais, peças teatrais, etc. Enfim, um conjunto de dados organizados sob uma perspectiva.

Os textos preparados que compõem o corpus devem ser categorizados em função de algum valor de metadado. O corpus Tycho Brahe é composto por textos de autores do português nascidos entre o século XIV e XIX (Galves & Faria, 2010). Estas datas são metadados para os documentos do corpus, pois não é o corpus que define os valores de metadados que categorizam-no, mas sim os documentos que o compõem. São os documentos que possuem as datas e não o corpus.

Um corpus pode ser analisado somente no nível textual e recorrer a ferramentas computacionais de busca para encontrar os elementos que se deseja[2], por exemplo, pode ser de interesse da pesquisa saber o número de ocorrências da preposição DE nos textos do corpus.

---

[2] Algumas buscas de ferramentas de texto até oferecem recursos de contagem de totais e utilização de expressões regulares, por exemplo.

Os textos do corpus Tycho Brahe são anotados sintaticamente, isto significa que o nível de detalhe da análise avança além da barreira da busca textual e vai até o ponto da estrutura sintática das sentenças. Tanto para informações textuais quanto sintáticas (e morfológicas), o recurso de realização de buscas é fundamental na prática de análise de corpus.

Para que um texto possa chegar até esse nível de análise, ele deve passar por um processo que envolve diversas etapas:

1. Inclusão e edição dos textos no corpus;
2. Etiquetagem morfológica das palavras das sentenças através de um modelo computacional probabilístico;
3. *Parsing* sintático das sentenças etiquetadas através de um modelo computacional baseado em modelos probabilísticos e/ou regras conhecido como *parser*;

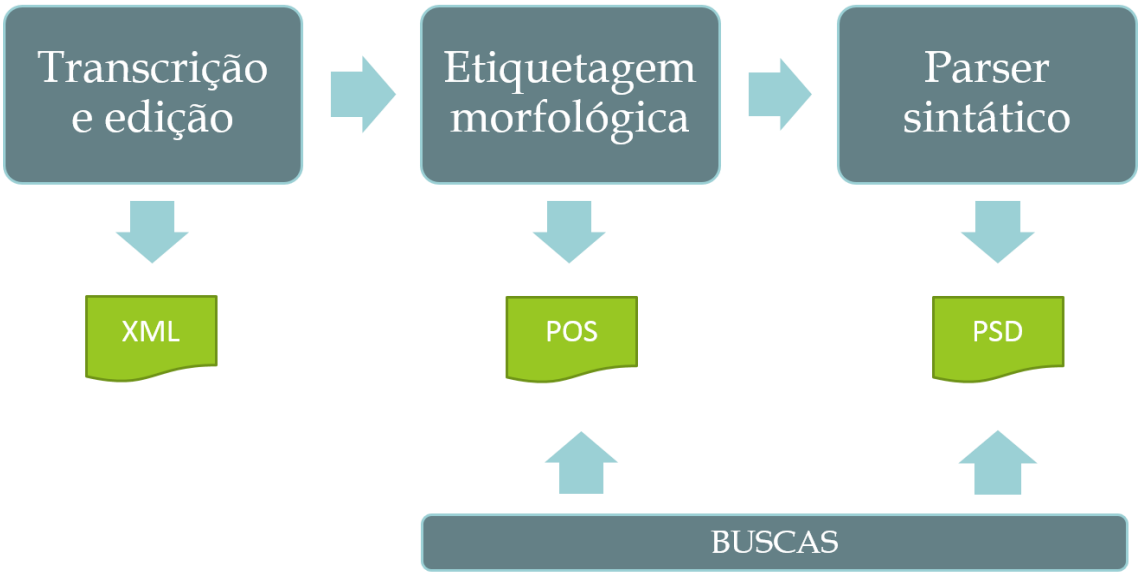


Figura 1 - processo do projeto Tycho Brahe

O eDictor compreende as etapas 1 e 2 descritas acima. A execução do passo 3 é realizada atualmente por um técnico responsável pela manutenção do sistema (vide apêndice A).

Na outra ponta do processo, cada um destes arquivos alimenta diferentes ferramentas de busca que possuem cada uma sua própria interface para analisar os arquivos gerados a partir do fluxo do projeto.

A primeira etapa composta pela transcrição e edição do texto gera como produto um arquivo estruturado de marcações no formato *xml*. Na fase de etiquetagem morfológica ocorre o processamento do arquivo produto da primeira fase por um modelo computacional probabilístico[3], sentença a sentença, gerando, como produto, um arquivo em formato de texto contendo sentenças com as palavras etiquetadas. A terceira fase compreendida pelo *parsing* sintático (realizado através de um outro modelo computacional probabilístico[4]) recebe como entrada as sentenças etiquetadas na segunda fase e entrega como produto um arquivo em formato de texto contendo as sentenças sintáticas representadas em árvores no formato de parênteses. Para fins de distinção entre os três produtos resultantes das ferramentas computacionais, adotam-se as terminologias *XML* (eXtended Markup Language), *POS* (Part-Of-Speech) e *PSD* (ParSeD), respectivamente.

Apesar do processo no projeto não estar totalmente automatizado, ele funciona adequadamente para seus propósitos, mesmo que onere o técnico responsável.

---

[3] Kepler, 2005.
[4] Bikel, 2003.

Por exemplo, caso o projeto obtivesse um aumento substancial na quantidade de textos preparados, o trabalho realizado pelo técnico iria ser o gargalo do fluxo de produção, além das questões de erro oriundos da execução repetitiva dos processos semiautomáticos dos *scripts*[5] e do alto tempo de processamento do *parser*.

Assim, fica evidente a necessidade de automatizá-lo, além de compreender que o processo como um todo se iniciaria e terminaria dentro do eDictor compondo, assim, o escopo da ferramenta. A análise do texto, buscas por padrões e quantificações estatísticas ficariam a cargo de outras ferramentas computacionais que trabalhariam em cima do conteúdo gerado pelo eDictor.

A adoção do *xml* para organização da informação do corpus (Paixão de Sousa, 2006) com a introdução do eDictor trouxe, sem dúvidas, benefícios aos usuários. O estabelecimento de um padrão traz resultados significativos para a compreensão do modo em que o corpus se organiza. A utilização de um padrão internacional facilita o trabalho do programador já que amplia-se a oferta de ferramentas que trabalham sobre o padrão.

Porém o formato *xml* não passa de um formato: uma regra adotada para padronização da comunicação. O dado para ser transformado para o formato *xml* passa por um processo chamado serialização. Serializar um objeto computacional consiste em manipular este objeto de tal forma que seja

---

[5] Estes processos incluem a realização do *upload* para o servidor e a execução de, pelo menos, 7 *scripts* incluindo conversores para preparação do texto para entrada das funções do parser e ainda o *splitter* para dividir os itens lexicais cujas anotações sejam compostas, por exemplo, P+D-F (preposição mais determinante feminino) para *da* que deve ser dividido em *de/P* + *a/D-F* e a cada item do par atribuindo sua etiqueta morfológica.

conservado seu estado atual de "mente"[6] possibilitando a recuperação das informações do objeto a qualquer momento no exato ponto em que foi serializado[7].

A imagem abaixo representa um trecho da estrutura *xml* adotada pelo corpus antes do uso do eDictor:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="../../xslt/catalog.xsl" type="text/xsl"?>
<document>
   <head id="a_001">
   </head>
   <body>
      <text id="a_001" t="unit" mapping="logic" num="1" title="Reflexões sobre a vaidade dos homens"
        title_ed="Reflexões sobre a vaidade dos homens">
        <sec t="pag"/>   <text_el t="pag_nr">01</text_el>
         <sec t="prologue" author="MA">
            <sec t="title" id="a_001_title_1"> REFLEXÕES SOBRE A VAIDADE DOS HOMENS </sec>
            <sec t="pag"/>   <text_el t="pag_nr">02</text_el>
            <sec t="pag"/>   <text_el t="pag_nr">03</text_el>
         </sec>
         <sec t="prologue" author="MA">
            <sec t="title" id="a_001_title_2">[DEDICATÓRIA]</sec>
            <sec t="title" id="a_001_title_3"> Senhor: </sec>
            <p id="a_001_p_1">
               <s id="a_001_s_1" t="">
                Ofereço a Vossa Majestade as Reflexões sobre a vaidade dos homens;
               </s>
               <s> isto é o mesmo que oferecer em um pequeno livro aquilo de que o mundo todo se compõe,
               e que só Vossa Majestade não tem: feliz indigência, e que só em Vossa Majestade se acha.</s>
               <s id="a_001_s_2" t="">Declamei contra a vaidade, e não pude resistir à vaidade inocente
               de pôr estes discursos aos Reais pés de Vossa Majestade; para que os mesmos pés, que
               heroicamente pisam as vaidades, se dignem proteger estas reflexões.</s>
               <s>Mas que muito, Senhor, que as vaidades estejam só aos pés de Vossa Majestade, se as
               virtudes o ocupam todo.</s><s> Alguma vez se havia de ver a vaidade sem lugar</s>
            </p>.
        </sec>
      </text>
   </body>
</document>
```

Figura 2 – exemplo de arquivo xml pré-eDictor

Cada marcação utilizada pode englobar outro nível de marcações, ou seja, permite o aninhamento das marcações. Analisando o exemplo acima, fica mais fácil de realizar a compreensão do que significa isto. (*document* → *body* → *text* → *sec*).

---

[6] Aqui me refiro aos mesmos termos utilizados por Turing, 1936.
[7] Irei retornar a este ponto no fim do capítulo 4.

Note que na terceira linha do exemplo, existe uma marcação denominada *document*, ela é usada no texto da seguinte forma:

<document>

Toda marcação em formato *xml* deve ser iniciada com < e fechada com >. A marcação de *document* indicada acima significa (para o modelo computacional programado para leitura de um arquivo em formato *xml*) que uma estrutura do tipo *document* se inicia neste ponto. **Toda** marcação deve ser fechada, pois é premissa deste tipo de formato que ela seja marcada como fechada, caso contrário todo o arquivo entrará em um processo de erro e será considerado como mal formado. Note que o fechamento da marcação ocorre na última linha do texto:

</document>

Outra característica fundamental do formato *xml* (e que dá o nome ao próprio formato *eXtended)* é que, apesar da estrutura rígida no formato de abre-fecha marcações, os valores que podem ser atribuídos a elas são virtualmente infinitos e estão relacionados à característica da regra adotada pelo modelo, por exemplo, o eDictor trabalha com documentos, logo a marcação mais alta na hierarquia de agrupamentos é aquela que representa o documento, porém se a regra de negócio adotada pelo modelo fosse relacionada a notas fiscais eletrônicas[8], a marcação inicial não precisaria ser obrigatoriamente *document*, mas poderia ser *nfe* ou ainda *nota*.

---

[8] Este exemplo é interessante pelo fato de o governo brasileiro ter adotado há alguns anos o formato *xml* para emissão e envio de notas fiscais de produtos. Quando se adquire um produto,

Os valores textuais devem estar fechados dentro das marcações e são compreendidos como texto puro (ou *string element* na implementação da estrutura), por exemplo:

```
<s id="a_001_s_1">Ofereço a Vossa Majestade as Reflexões sobre a vaidade dos homens;</s>
```

A marcação *s* corresponde a uma sentença identificada como *a_001_s_1* que se refere à sentença encapsulada *Ofereço a Vossa Majestade as Reflexões sobre a vaidade dos homens*. Esta identificação não é gerada aleatoriamente, ela faz parte das regras de marcação da ferramenta.

A sentença *Ofereço a Vossa Majestade as Reflexões sobre a vaidade dos homens* é transcrita para o eDictor (na fase de transcrição) pelo usuário. Ao término do processo, ou seja, assim que o usuário decide que a transcrição foi realizada totalmente, ele aciona uma ação (ou evento) disponibilizado pela ferramenta em sua interface que aciona regras programadas para preparação do texto transcrito para edição.

O texto é, então, quebrado em palavras através do reconhecimento dos espaços em branco. Este processo é conhecido como *tokenização*[9]. As palavras são agrupadas em parágrafos e sentenças e, por fim, aninhadas seguindo a regra "parágrafos possuem sentenças que possuem palavras" em uma relação 1 para *n*, sempre da menor unidade em direção à maior. Vamos inverter a lógica coloocando na passiva: palavras são possuídas por sentenças que são possuídas por parágrafos.

---

a validade da nota fiscal não é relativa ao impresso conhecido como DANFE (mero demonstrativo), mas sim por um arquivo *xml* que o consumidor pode solicitar ao fornecedor.

[9] Retorno a este ponto no capítulo 3.1.

Passando para a fase de edição do eDictor, é possível realizar edições no texto, atribuindo valores aos blocos e às palavras[10].

Finalizado o processo de edição, o usuário envia o texto para o etiquetamento morfológico. Este processo computacional consiste em recuperar as sentenças do arquivo *xml*, criar um arquivo em formato texto contendo em cada linha uma sentença e realizar a execução de outro módulo de programação. Este módulo é o etiquetador morfológico. Ele consiste em uma função que recebe uma lista de sentenças e, uma a uma, analisa probabilisticamente as possibilidades de ocorrências das palavras umas em relação às outras[11]. O resultado da função é um texto contendo as sentenças, uma em cada linha, com as palavras concatenadas, uma a uma, com sua provável etiqueta morfológica conforme a imagem abaixo.

[10] Conforme descrito no capítulo 2.2.
[11] Através de cadeias de Markov, Kepler 2005.

```
#!FORMAT=POS_0

<text>

<P_01>/CODE

<heading>/CODE REFLEXÕES/NPR-P SOBRE/NPR A/D-F VAIDADE/NPR DOS/P+D-P HOMENS/NPR-P
<_heading>/CODE

<P_02>/CODE

<P_03>/CODE <heading>/CODE Senhor/NPR :/. A-001,03.1/ID ./PONFP

<_heading>/CODE Ofereço/VB-P a/P Vossa/PRO$-F Majestade/NPR as/D-F-P Reflexões/NPR-P
sobre/P a/D-F vaidade/N dos/P+D-P homens/N-P ;/. A-001,03.2/ID ./PONFP

isto/DEM é/SR-P o/D mesmo/ADJ que/C oferecer/VB em/P um/D-UM pequeno/ADJ livro/N aquilo/DEM
de/P que/WPRO o/D mundo/N todo/Q se/SE compõe/VB-P ,/, e/CONJ que/WPRO só/FP Vossa/PRO$-F
Majestade/NPR não/NEG tem/TR-P :/. A-001,03.3/ID ./PONFP

feliz/ADJ-G indigência/N ,/, e/CONJ que/WPRO só/FP em/P Vossa/PRO$-F Majestade/NPR
se/SE acha/VB-P ./. A-001,03.4/ID ./PONFP

Declamei/VB-D contra/P a/D-F vaidade/N ,/, e/CONJ não/NEG pude/VB-D resistir/VB à/P+D-F
vaidade/N inocente/ADJ-G de/P pôr/VB estes/D-P discursos/N-P aos/P+D-P Reais/ADJ-G-P pés/N-P
de/P Vossa/PRO$-F Majestade/NPR ;/. A-001,03.5/ID ./PONFP

para/P que/C os/D-P mesmos/ADJ-P pés/N-P ,/, que/WPRO heroicamente/ADV pisam/VB-P as/D-F-P
vaidades/N-P ,/, se/SE dignem/VB-SP proteger/VB estas/D-F-P reflexões/N-P ./. A-001,03.6/ID
./PONFP

Mas/CONJ que/WD muito/Q ,/, Senhor/NPR ,/, que/C as/D-F-P vaidades/N-P estejam/VB-SP só/FP
aos/P+D-P pés/N-P de/P Vossa/PRO$-F Majestade/NPR ,/, se/CONJS as/D-F-P virtudes/N-P o/CL
ocupam/VB-P todo/Q ./. A-001,03.7/ID ./PONFP
```

Figura 3 – trecho de arquivo *pos*

O eDictor, via programação, recupera este arquivo, quebra-o em sentenças (seguindo um padrão pré-definido[12]) e realiza um processo de *tokenização* em espaços em branco. As palavras agora são composições do seu valor e sua etiqueta na forma VALOR/TAG (por exemplo, *Ofereço/VB-P*) e é realizada a quebra deste composto em mais um processo de *tokenização*, agora pelo caractere "/". Já que a sentença no arquivo *xml* deve, obrigatoriamente, possuir uma correspondência bijetora (uma a uma sequencialmente) com a sentença no arquivo *pos*, isto é, cada palavra da sentença no *xml* deve ser igual a cada palavra do arquivo *pos* em sequência, o eDictor realiza um pareamento, palavra

[12] Importante notar que este processo também é realizado pelo etiquetador, afinal o processo de leitura e escrita das sentenças em arquivo possui a mesma lógica, sempre colocando uma sentença por linha.

a palavra, atribuindo à palavra do arquivo *xml* a etiqueta correspondente do composto resultado do processo de etiquetagem.

Neste momento o eDictor, via programação, salva a estrutura em arquivo *xml*, terminando o fluxo automatizado do processo de análise de corpus do projeto Tycho Brahe. As outras etapas do processo que levam ao corpus sintaticamente anotado são processadas semiautomaticamente, ou seja, necessitam de um operador técnico (vide apêndice A).

Todos os documentos do corpus Tycho Brahe são, então, disponibilizados *online* com a possibilidade de serem trabalhados através de buscas morfológicas e sintáticas.

No futuro, o Projeto Tycho Brahe disponibilizará uma suíte de aplicativos possibilitando uma nova forma de experienciar as informações contidas nos documentos, proporcionando ao usuário formas de observar fenômenos linguísticos em escalas muito maiores do que as disponíveis até o momento.

2.2. Breve histórico do eDictor

Com a quantidade de informação disponível atualmente através dos meios de comunicação, a árdua tarefa de construção de um corpus, seja pela composição dos elementos que o caracterizam como, por exemplo, a escolha de um período histórico ou de uma certa localização geográfica, seja pelo minucioso processo de coleta dos dados *in natura*[13], fica ainda mais complexa quando não são utilizadas ferramentas computacionais para automatização de processos, seja

[13] Livro, *fac-símile*, imagem de um arquivo *pdf*. *In natura*, na Wikipedia, é brevemente descrita como "Conditions present in a non-laboratory environment".

no todo, seja em parte dele. As ferramentas computacionais não somente agilizam o processo, mas também colaboram para que os dados possam ser compreendidos de uma forma que sem eles não seria possível. A análise de dados em grandes quantidades pode revelar outras possibilidades que não podiam ser detectadas anteriormente (Unsworth, 2006).

São várias as ferramentas disponíveis para construção de um corpus e elas atuam em vários níveis deste processo. Uma dessas ferramentas é o eDictor (Paixão de Sousa et al., 2007): uma ferramenta computacional especializada para edição de textos em formato *xml* que permite que o usuário realize operações no nível da palavra. Muitas vezes, durante o processo de construção de um corpus, um texto precisa passar por uma fase de análise onde o usuário deve corrigir inconsistências (problemas na transcrição do texto), ajustá-lo ou ainda prepará-lo para uma forma esperada por um modelo computacional.

É sempre importante notar que computadores são desenvolvidos utilizando-se regras e a forma de entrada do dado deve ser a esperada, ou seja, deve haver uma preparação do conteúdo para que ele possa ser enviado para um modelo computacional a fim de que ele retorne um resultado esperado, isto é, em um formato que o usuário o compreenderá. O conteúdo do resultado varia de acordo com o conteúdo de entrada, porém a forma como ele se apresenta é uma só: a estrutura da informação é sempre conhecida.

No caso de textos impressos que não podem ser transpostos imediatamente para uma plataforma computacional[14], existe um processo anterior ao de preparação que consiste, basicamente, na transcrição (via digitação) do texto para o computador. Nesta etapa, o usuário digita o documento, página a página, informando as quebras de página[15] e inserindo caracteres especiais[16].

O eDictor, sob este ponto de vista, pode ser compreendido como uma ferramenta de processamento de textos que possui duas funções: uma inicial de transcrição do texto onde é disponibilizada ao usuário uma interface para que ele realize este processo de digitação e uma etapa posterior (com outra interface) para edição do conteúdo.

O eDictor se torna o centro do processamento de dados, pois é nele que cabe a estruturação da informação, sua concepção e as operações sobre a estrutura, modificando os dados à medida que é realizada a preparação do texto. Sendo assim, é fundamental que a informação fique estruturada de maneira não-

---

[14] Através de implementações de identificação automática de caracteres que "leem" uma imagem de uma página e retornam o texto contido nesta imagem. Esta tecnologia é conhecida por OCR, Optical Character recognition. http://en.wikipedia.org/wiki/Optical_character_recognition

[15] O termo "informar a quebra de páginas" pode soar um pouco estranho, mas é ideal para descrevê-lo, pois a quebra pode ser especificada através de uma marcação no texto ou pode-se utilizar um comando computacionalmente programado para realizar tal função. O eDictor trabalha com a primeira opção, o usuário deve informar uma tag específica (determinada a priori) que o sistema irá identificar e processar para realizar a quebra das páginas. Na versão web optei por uma ação do usuário que deve clicar em um botão na tela para que uma nova página em branco lhe seja exibida. Um processador de texto como o Microsoft Word trabalha na base de páginas e possui um atalho (ctrl+N) que cria a nova página e direciona o cursor para o seu início. Caso a digitação atinja o limite máximo determinado pelas propriedades de largura e altura pré-definidas através da escolha de um modelo *default* (A4, Letter, etc.), o sistema automaticamente realiza esta quebra de página através de funções programadas.

[16] Quaisquer caracteres que não façam parte do grupo de caracteres que compõe o teclado do computador do usuário.

ambígua, não-redundante e de forma simples para que durante todo processo a informação seja utilizada com o mesmo formato.

A centralização destas informações em uma fonte única já vinha sido discutida há anos (Paixão de Sousa & Trippel, 2004) e deu seus primeiros passos em direção a ela a partir da reestruturação do corpus Tycho Brahe em 2005. Nesta altura do processo, os dados estavam distribuídos em três camadas (ou, simplesmente, arquivos):

1. Estrutura do texto através de marcações (*xml*);
2. Estrutura morfológica por sentença em formatação específica (*pos*);
3. Estrutura sintática por sentença em formato de parênteses (*psd)*.

A segunda camada foi superficialmente eliminada através da interface do eDictor, ou seja, a ferramenta processa um arquivo do tipo (2), e retira dele as informações morfológicas após o processo de etiquetagem inserindo este dado na estrutura de marcação em (1). Através do disparo de um processo (*trigger*), o eDictor gera um arquivo de formato em (2), lê este arquivo e insere os dados em (1). O usuário não "vê" este processo, por isso pode-se dizer que o arquivo foi eliminado superficialmente. O arquivo do formato (3) é o produto final do processo completo de análise sintática do texto obtido a partir de operações semiautomáticas executadas por um técnico especializado (vide apêndice A).

No caso da edição de *tokens*, é possível atribuir valores relacionados aos processos de preparação do texto, por exemplo, em textos mais antigos do projeto que remetem ao uso de caracteres especiais (por exemplo o s longo

latino ſ), a palavra passa por um processo de grafia que o substitui por um *s* moderno. Esta edição pode ser de outros tipos e ainda possuir níveis de edição, por exemplo, a palavra "ſotis" sofre uma edição de grafia para "sotis" e um edição de padronização para "sutis"[17]. Os tipos de edição disponíveis são configuráveis via regras de programação do sistema e possuem uma interface própria nas configurações do sistema[18].

Utilizando-se a disposição sequencial das palavras no texto como forma de relacionamento dos elementos no texto, decorrem-se dois problemas: como representar as quebras de linha e como representar as quebras de página? A página é o mais alto elemento encadeador de palavras em um texto do ponto de vista de sua organização espacial original. As palavras se dispõe logicamente organizadas em blocos, que ultrapassam os limites das páginas, já que sentenças e parágrafos podem ser iniciados em uma página e terminados em outra. Este problema não pode ser resolvido através de uma estrutura que possui marcações aninhadas, pois uma marcação que seja aberta em um nível deve ser fechada dentro do mesmo nível, caso contrário a estrutura é considerada mal formada. Veja os exemplos abaixo:

1- Sentença (s) inicia e termina em um parágrafo (p) com marcação correta.	 <s>Ofereço a Vossa Majestade as Reflexões sobre a vaidade dos homens</s>
2- Marcação errada para	

[17] Exemplo retirado da edição de "O Velho da Horta" de Gil Vicente.
[18] Estas regras constituem o sistema de parâmetros.

uma sentença (s) iniciada em um parágrafo (p) e continuada após seu término.	<s>Ofereço a Vossa Majestade as Reflexões sobre a vaidade dos homens</s>
3- Uma opção correta para refletir a quebra de uma sentença em um parágrafo e continuidade em outro parágrafo.	 <s>Ofereço a Vossa Majestade</s> <s> as Reflexões sobre a vaidade dos homens</s>

Tabela 1 – exemplos de *xml* para parágrafos e sentenças

O problema do aninhamento existe porque a marca de fechamento em *xml* é significada pelo símbolo "/" em "</" ou "/>". É a barra que indica o fechamento. A identificação dos pares (, ou <s>,</s>) é realizada de acordo com a gramática do *xml*[19].

Além da questão do aninhamento, ainda havia o ponto de identificar qual seria o mais alto elemento encadeador no texto, ou seja, a partir de qual elemento o texto se derivaria.

Optou-se (Paixão de Sousa, 2006) pela utilização de uma entidade lógica como o parágrafo. Porém se dissermos que uma página é uma unidade espacial, poderíamos dizer que o parágrafo é lógico e não espacial? Quais seriam as definições para espacial e lógico? Ou físico e lógico?

[19] DTD, http://www.w3schools.com/dtd/.

Poderíamos entender o espacial como aquela unidade que traz a formatação para arranjo da informação. Por exemplo, a indentação inicial ou uma linha em branco separando um parágrafo do outro não seriam indicativos de um arranjo espacial. Mesmo utilizando-se a alternativa da organização da informação desta forma, o problema do aninhamento continua. A solução não poderia estar relacionada à compreensão de um bloco como espacial ou lógico simultaneamente, pois a adoção de uma unidade espacial ou lógica para encapsulamento não interfere na necessidade de se marcar a quebra, seja de linhas ou de páginas. Os exemplos abaixo mostram isso.

```
<w id="89">
      <o>compa<bk t="l" id="bk_6"/> nhia</o>
      <e t="seg">companhia</e>
      <e t="jun">companhia</e>
</w>
```

```
<w id="102">
      <o>respeitosamente<bk t="l" id="bk_8"/></o>
</w>
```

```
<w id="169">
      <o>.<bk t="p" id="bk_17"/></o>
</w>
```

Tabela 2 – exemplos de *xml* com marcas de quebra

No primeiro exemplo ocorre uma quebra de linha separando a palavra *companhia* em duas palavras (*compa* e *nhia*), a marcação *bk* é identificada como

uma marca de quebra que possui dois atributos, um *id* identificando a quebra unicamente no texto e um tipo (*t*) ao qual pode ser atribuído dois valores distintos: *l* significando uma quebra de linha (exemplos 1 e 2) e *p* correspondendo a uma quebra de página (exemplo 3).

É importante notar que neste modelo, a lógica que corresponde à compreensão das palavras do exemplo 1 (*compa* e *nhia*) para a formação da palavra *companhia* é feita através do encapsulamento de toda informação (duas palavras e a quebra) por uma marcação do tipo *o* que corresponde ao que seria o valor original da palavra[20].

É óbvia a necessidade da informação de quebra. A informação de quebras no texto pode ser obtida pelas limitações do espaço físico da interface, isto é, um livro possui um tamanho pré-determinado da folha de papel e uma ferramenta computacional depende da resolução do monitor (seja ele uma tela convencional, um *tablet* ou um aparelho celular). Porém, neste caso, a informação do local exato da quebra seria perdido ao menos que prévias configurações tenham sido estabelecidas para se utilizar uma determinada medida de largura e altura simulando o texto *in natura*. Isto não parece fazer muito sentido, já que um livro impresso pode possuir diversas formatações e exigiria constantes modificações para configuração dos valores associados a cada formatação, além de um esforço computacional para adequação da ferramenta.

---

[20] Isto decorre da funcionalidade de edição da ferramenta (descrito anteriormente), as marcações *e* correspondem aos valores de cada nível de edição. Elas são opcionais, já que uma palavra pode já se encontrar no formato necessário para processamento nos módulos computacionais.

Uma última possibilidade de eliminação da marca de quebra[21] poderia ser realizada caso a lógica de ordem sequencial fosse abandonada e as palavras passassem a ser "geolocalizadas"[22], isto é, estando o texto em um plano (a folha), cada palavra poderia ser mapeada em um sistema de coordenadas. Assim, seria possível eliminar a marca de quebra. A adoção deste sistema iria impactar a ferramenta e suas funcionalidades e exigiria esforço programacional.

Ainda que o eDictor não tenha solucionado todos os problemas para a solução do corpus Tycho Brahe e da edição de textos, seu surgimento marcou uma nova fase para os trabalhos em corpora. O avanço técnico-computacional proporcionado pela ferramenta aumentou a qualidade e eficiência dos trabalhos possibilitando uma melhor compreensão dos desafios que ainda deveriam ser superados.

---

[21] Cabe ressaltar que a análise sobre a marca é válida, pois tanto a inclusão de um novo objeto na estrutura quanto os atributos que cada um deles pode possuir são utilizados na modelagem da estrutura e aumentam a lógica no módulo computacional. Antes de qualquer programação deve ser realizada uma análise aprofundada sobre os processos de criação e modificação da estrutura pois, sendo o aspecto central do modelo, o impacto na programação é sempre relevantemente significativo.

[22] Este processo de geolocalização é exatamente o que fazem os arquivos *xml* gerados através de OCR.

3. Organização da informação

Este capítulo tem por objetivo explicar como a organização da informação dos dados do corpus Tycho Brahe manipulados pela ferramenta eDictor teve que ser refeita para que os problemas de redundância da informação fossem eliminados para possibilitar uma forma mais adequada de acesso à informação.

Esta centralização das informações possibilitou a criação de uma central de acesso aos dados, necessária para que a ação de manipulação da informação realizada pelo usuário se desse de forma única.

Para isto, foi necessária a criação de uma estrutura de dados capaz de armazenar em somente um repositório todas as informações que o corpus disponibiliza (textuais e morfossintáticas). Caso esta estrutura não tivesse sido adotada, o eDictor continuaria sendo um editor de arquivos *xml* o que inviabilizaria sua passagem para a nuvem[23].

3.1. Primeira parte: análise da estrutura do eDictor

De todo o fluxo descrito até agora decorrem dois problemas.

Primeiro, os dados do texto encontram-se fragmentados e são redundantes nos três arquivos: ao menos a palavra encontra-se nos três. O arquivo *pos* serve somente para transferência de informação, uma espécie de ponte entre os módulos computacionais.

O segundo problema é referente ao código de identificação da sentença. Durante os testes de importação dos dados, foi verificado que não existia uma

[23] Conforme discutido no fim do capítulo 4.

correspondência confiável entre as duas pontas do processo, isto é, não é trivial dizer que uma sentença no arquivo *psd* é a mesma que a do arquivo *xml*. Isto implica que não se pode estabelecer uma relação entre as informações textuais atribuídas aos blocos de texto (parágrafo, sentença) e as sentenças sintaticamente anotadas. Não que isto seja um problema para a análise do corpus, pois cada pesquisa é realizada dentro de um determinado nível de especificidade de suas características, ou seja, para um corpus que não possui como premissa a análise morfológica, bastam os módulos para transcrição e edição. Por outro lado, como no caso do projeto Tycho Brahe, as ferramentas que fazem a análise do corpus sintaticamente anotado não fazem uso da estrutura do texto transcrito.

A questão levantada no início é que, estando estes dados organizados de forma desfragmentada, diminuía-se o poder de cruzamento das informações e da análise. Logo, finalizar a automatização do processo era somente o início de um outro trabalho que consistia na reorganização desta informação e a premissa fundamental de eliminar esta redundância. A conclusão obtida foi a da necessidade da criação de uma estrutura de dados capaz de armazenar todos os dados de forma não-redundante.

A reprodução da estrutura de dados de forma aninhada, ou seja, incluindo camadas de elementos um dentro dos outros, foi a primeira solução adotada no eDictor. Nela, o texto em formato *xml* possuía uma representação muito semelhante ao do próprio texto *in natura*, isto é, os blocos de informação se davam de acordo com a própria estruturação do texto.

Para cada bloco era criada uma marcação: para um parágrafo existia uma marcação de parágrafo, para uma sentença uma marcação de sentença, e assim por diante em uma relação um para um. Existia ainda uma tipologia de dados e cada bloco poderia representar um tipo de acordo com a relação lógica que a ele foi atribuída, por exemplo, uma seção poderia ser do tipo *carta* e outra seção do tipo *descrição*. A atribuição destes tipos era relativamente livre (dada pelo usuário do eDictor em processo de configuração de parâmetros) gerando uma rica fauna na tipologia dos blocos estruturais. A relação de aninhamento também ficou complexa como pode ser visto pelo diagrama de classes abaixo.

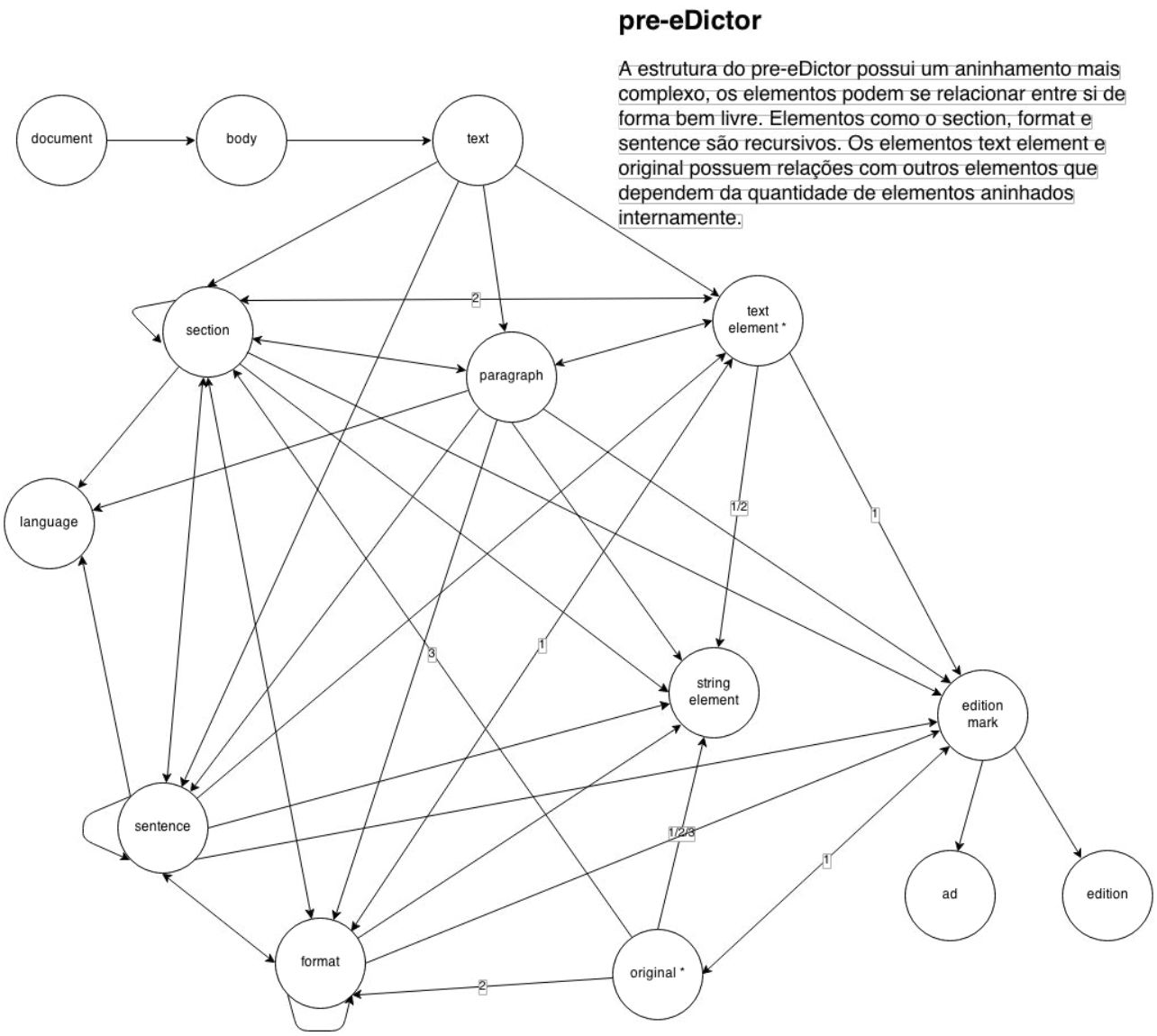


Figura 4 – estrutura do *xml* pré-eDictor

Como os elementos possuíam uma ideia genérica (por exemplo, como se define uma seção?), o aninhamento deles se tornou muito entrelaçado, isso sem contar a tipologia, ou seja, a complexidade do aninhamento da estrutura era ainda maior[24].

[24] Só foi possível compreender a complexidade das estruturas da fase pre-eDictor no projeto (fase em que os arquivos em formato *xml* eram escritos "à mão", sem a utilização de ferramentas especializadas para o formato) pelo exercício de importação de dados.

Após os trabalhos de Paixão de Sousa, et al. 2010 e o surgimento do eDictor em versão plataforma, a estrutura de aninhamento mudou. Ela passou a ser vista como uma espécie de aninhamento processual partindo-se de um maior nível na hierarquia até a menor informação possível de ser atribuída a uma palavra.

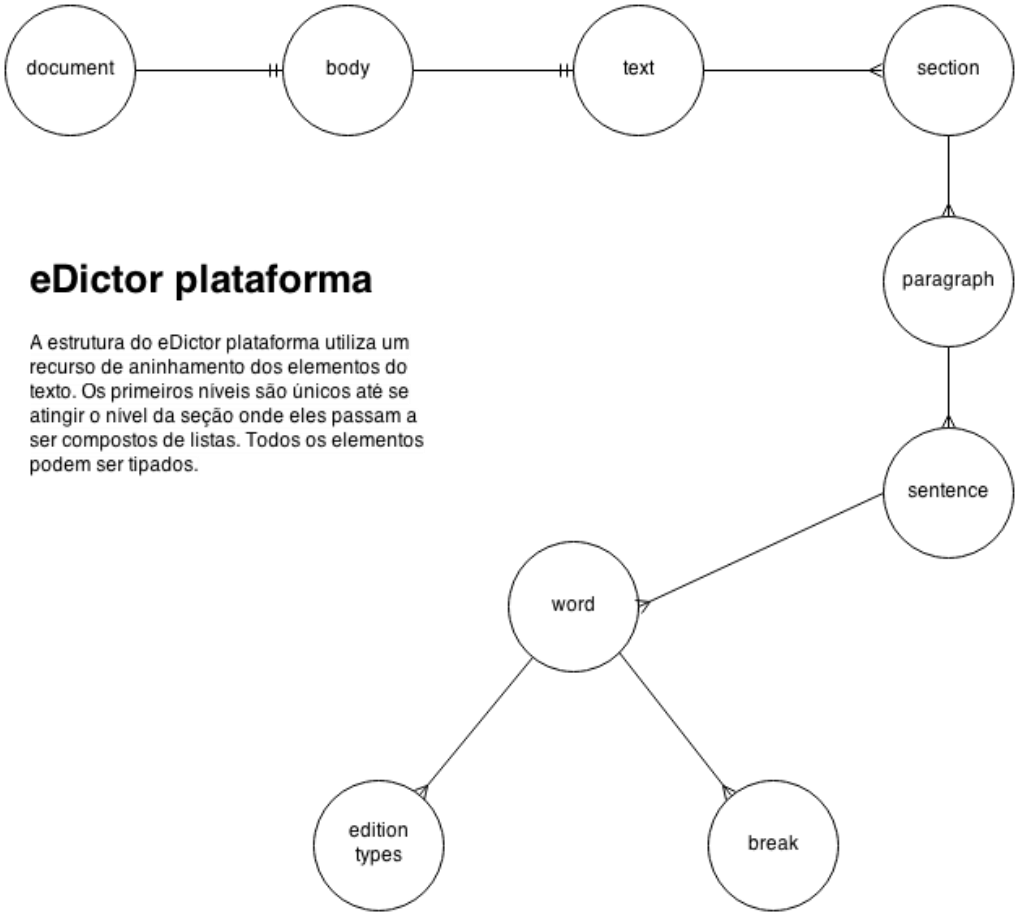


Figura 5 – diagrama de classes (marcações) do eDictor plataforma

Esta estrutura também poderia ser vista sob uma perspectiva de cascas, igual uma cebola:

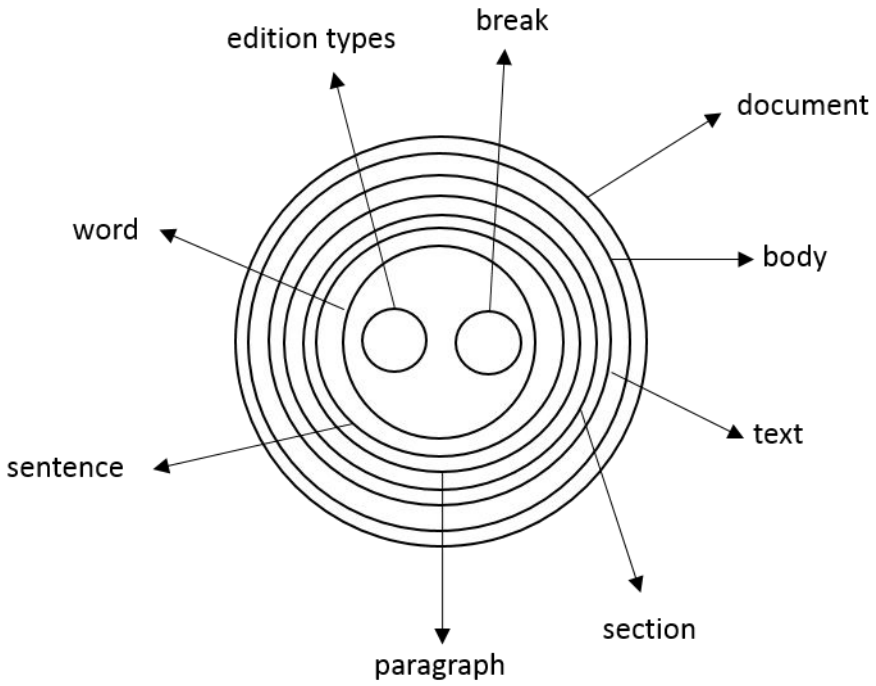


Figura 6 – diagrama do eDictor plataforma em formato de cebola

Esta é uma estrutura que organiza as estruturas lógicas do texto seguindo o modelo de relacionamento um para um entre marcações e blocos organizacionais: um parágrafo continua sendo identificado por uma marcação específica para ele e as tipologias continuam sendo adotadas.

Ao analisar as fases do eDictor pode-se dizer que a estruturação da informação caminhava em direção a uma desfragmentação do texto, ou seja, a busca por uma unidade mínima significativa já iniciada com o *string element* que se torna a palavra (e depois o *token*), podendo levar a um entendimento das entidades relacionadas entre si como relações de posicionamento entre mínimas unidades.

A estrutura já vinha se organizando em direção a esta unicidade, porém, com a adoção da estrutura que apresento neste trabalho, foi possível se desprender

totalmente da forma como os blocos organizacionais do texto se davam de um ponto de vista espacial.

Este passo em direção a uma generalização das entidades agrupadoras possui um sério reflexo na compreensão da informação. O texto se desprende da estrita visualização espacial e passa a depender totalmente de um software que possua as regras lógicas para sua compreensão.

O processo básico e fundamental para organização desta nova estrutura se chama *tokenização*. Ele consiste na detecção dos *tokens* para criação dos blocos lógicos de organização destes *tokens*. A palavra passa a ser compreendida no sistema como *token* a partir do momento em que ocorre este processo.

A tokenização é uma função que recebe um texto e retorna um conjunto de *tokens* ordenados separados pela identificação de um caractere específico. No caso de línguas ocidentais, o caractere *default* utilizado é o espaço em branco[25].

Programaticamente, a mesma função que quebra as palavras em *tokens*, pode ser utilizada para quebrar quaisquer conjuntos de palavras utilizando quaisquer regras. Por exemplo, se eu fornecer à função de tokenização a sequência de caracteres *abacaxi;pêra;maçã* e realizar a *tokenização* por ponto-e-vírgula, ele irá me retornar o conjunto de *tokens* com os nomes das frutas

[25] É importante saber que em sistemas computacionais um espaço em branco é identificado como um caractere da mesma forma que todos os caracteres "normais" (a, b, c, 1, ã, ç) e caracteres especiais de marcação (quebra de linha, quebra de página, fim de arquivo)

listadas. Em Java, a *tokenização* possui como valor *default* a quebra por espaço em branco[26].

Pela análise do fluxo do processo do Tycho Brahe e dos arquivos (produtos) de cada passo do fluxo, conclui-se que o elemento redundante nos três formatos é a palavra. Porém ela passa a ser compreendida como o resultado do processo de *tokenização* do texto e não mais como uma simples palavra. Agora ela é um composto do valor (a palavra em si) e de sua etiqueta morfológica, ainda que inicialmente vazia.

A organização do texto é feita em blocos: parágrafos, sentenças, seções, etc., que possuem relações livres com os *tokens*, ou seja, eles se relacionam de forma 1 para N, um bloco pode se relacionar com qualquer número de *tokens*, mas não no sentido de que o bloco possui *tokens*, mas sim que um bloco, como entidade, pode se relacionar com qualquer *token*, como entidade. Para a estrutura de dados, um bloco é um *chunk*.

No eDictor, os *chunks* são determinados pelos *tokens* inicial e final que o compõem. Estes dados são dois atributos do *chunk* que denominam estes valores. Assim, temos a estrutura de dados inicial representada na imagem ao lado.

De fato, esta relação já vinha sendo adotada pelo eDictor desde seu início, pois utilizando uma estrutura de marcações aninhadas, uma palavra está contida em um bloco, seja ele parágrafo, sentença, etc.

---

[26] A implementação se refere aos métodos *split* do objeto *String* e ao construtor da classe *StringTokenizer*, ambos naturais da implementação da linguagem Java atualmente. As referências à programação neste documento são referentes à API de versão 7.

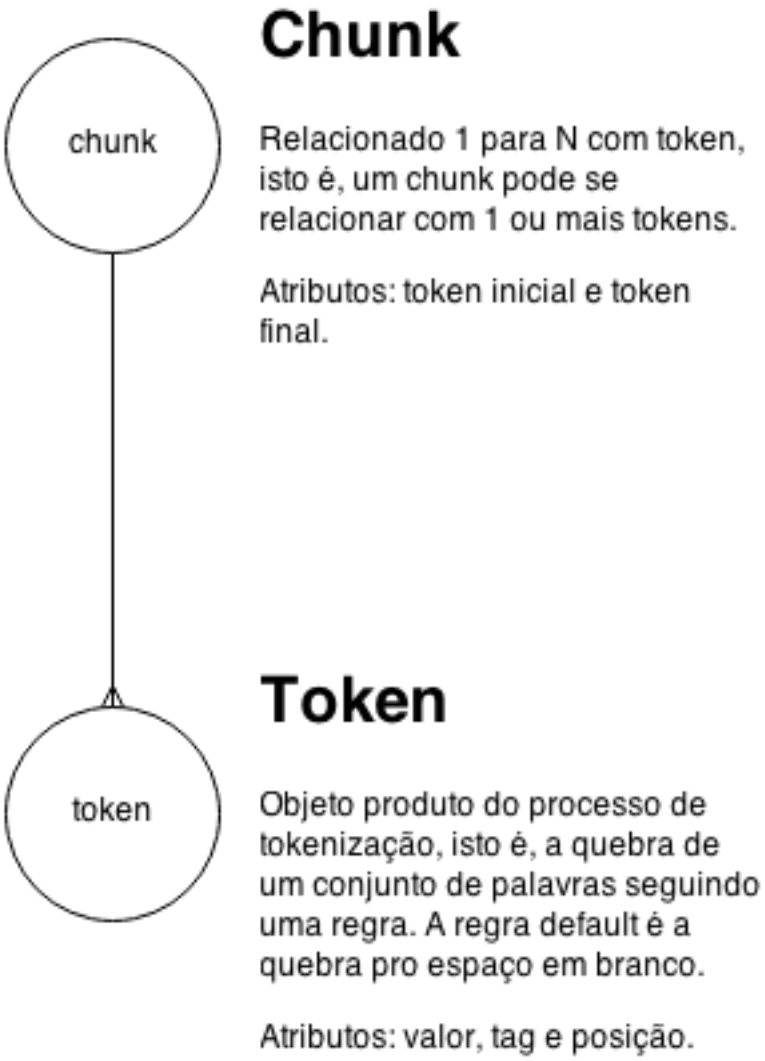


Figura 7 - chunk / token

A estrutura aqui proposta passa a trabalhar como uma generalização das formas específicas do texto.

```xml
<body>
  <text id="text_1" words="83164">
    <sc id="sc_1">
      <p id="p_1" t="numero-carta" f="b">
        <s id="s_1">
          <w id="1">
            <o>Carta</o>
          </w>
          <w id="2">
            <o>311</o>
          </w>
        </s>
      </p>
      <p id="p_2" t="comentario">
        <s id="s_2">
          <w id="3">
            <o>Arquivo</o>
          </w>
          <w id="4">
            <o>do</o>
          </w>
          <w id="5">
            <o>barão</o>
          </w>
          <w id="6">
            <o>de</o>
          </w>
          <w id="7">
            <o>Jeremoabo</o>
```

Figura 8 - trecho do xml do eDictor plataforma

Na figura 8, podemos observar que um texto está representado com parágrafos (marcação *p*) contendo sentenças (marcação *s*) que possui palavras (marcação w). O atributo de identificação da palavra (id) corresponde ao número sequencial da palavra no texto. Considerado o parágrafo e as sentenças como *chunks*, a estrutura da imagem 7 representaria de maneira adequada a estrutura aninhada disposta na imagem 8.

Novamente é importante reforçar que um arquivo *xml* é um formato obtido através do processo de serialização de uma estrutura. Ele representa a estrutura no seu atual "estado de mente", isto é, a estrutura contendo as palavras deste

texto. Este estado se dá em função dos diferentes valores atribuídos às palavras e à configuração dos encadeamentos dos blocos. Cada estrutura serializada é única em função da organização do texto e dos seus valores. Extraindo-se dela o essencial, chega-se à estrutura de relacionamento 1 para N entre *tokens* e *chunks*, isto é, um *chunk* está relacionado a um ou mais *tokens*. No caso do eDictor, esta relação se dá em função da ordem sequencial da ocorrência dos *tokens* no texto.

Levando em consideração a análise até o momento, o que fazer em relação à marca de quebra? Ela é necessária para um modelo que utiliza a ordem sequencial de *tokens* como ordem lógica da linearidade.

Em se tratando de textos de línguas ocidentais, esta ordem segue do canto esquerdo superior para a direita até encontrar uma marca de quebra de linha que caracteriza a lógica de continuar a leitura na próxima linha, a partir da esquerda, abaixo do primeiro *token* com uma distância de altura padronizada. A leitura continua seguindo por esta lógica até chegar ao canto direito inferior, onde encontra a marcação que define o fim da página. Esta lógica já não se aplica a textos em línguas orientais que possuem ordenações lineares diferentes horizontal e verticalmente.

Esta questão da orientação não é óbvia para uma máquina: os módulos computacionais são programados com regras específicas para apresentação do texto em sua interface. Independente da ordem de linearidade do sistema de escrita de uma língua, uma estrutura de dados que utiliza a lógica sequencial atribui aos *tokens* um valor numérico sequencial e o texto passa a ser compreendido como uma lista.

A quebra (*break*), portanto, deve possuir informações suficientes para que a interface possa ser disponibilizada da forma que se espera. Como a página é entendida como *chunk*, a entidade *break* deve orientar a quebra de linha somente e não mais a quebra de página. Afinal, ao fim de uma página há, sempre, uma quebra de linha.

Toda quebra de linha está relacionada a um único *token*. A imagem abaixo representa a estrutura ampliada com a adoção da marca de quebra.

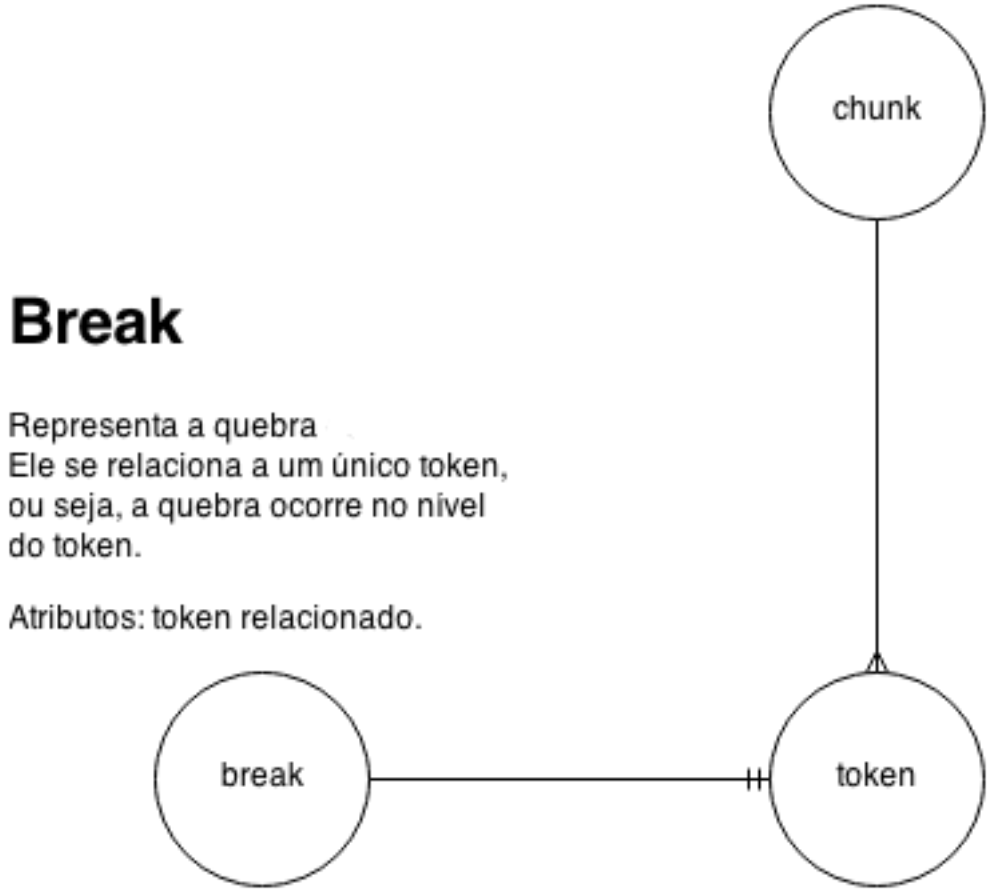


Figura 9 – introdução da quebra na estrutura

Caso a quebra ocorra no meio de uma palavra, como no exemplo da tabela 2, não se deve entender a palavra (*companhia*) como sendo uma palavra quebrada ao "meio", pois havendo a quebra de linha, podemos ser levados a

compreendê-la como uma unidade lógica que foi espacialmente dividida, mas, e se a víssemos como dois *tokens* espacialmente divididos, porém logicamente unidos? Não seria arbitrário dizer que a unidade lógica possui um significado atribuído em razão da compreensão de um ou outro elemento espacialmente disposto? O que quero dizer é que a compreensão de duas palavras escritas separadamente só se dá como unidade lógica pelo uso da regra da língua (pelo uso da palavra ou do símbolo indicador de quebra), mas para aquele que não usa a regra, os *tokens* não seriam nada mais que dois *tokens* isolados. O computador os entenderia desta segunda maneira.

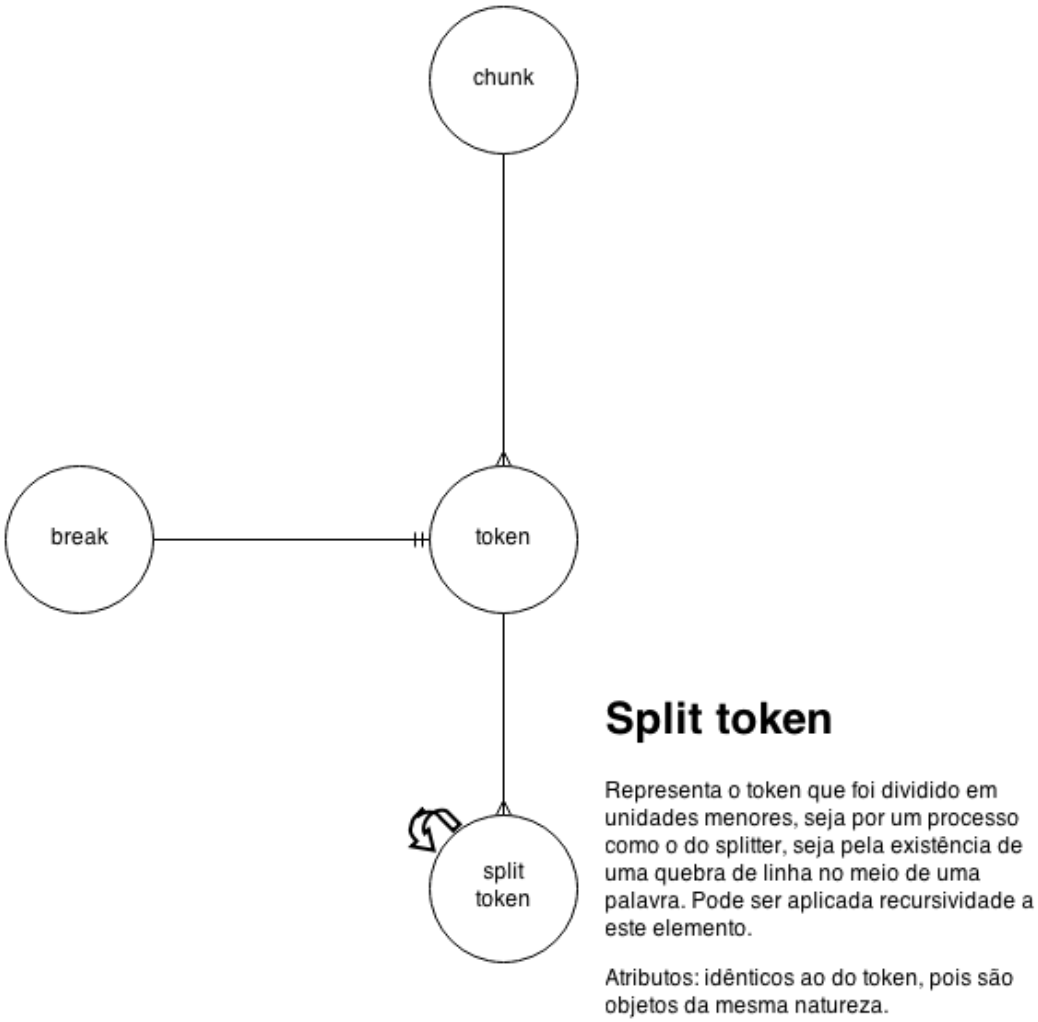
3.2. Segunda parte: análise da estrutura a partir das operações de edição

Os *tokens* do exemplo da tabela 2 (*compa* e *nhia*) possuem características que os fazem ser considerados *tokens* (são produtos do processo de *tokenização*), porém não possuem um valor de etiqueta morfológica associada a cada um deles, a etiqueta está associada ao *token companhia*, portanto para que ocorra a assimilação da informação morfológica é necessária uma programação que os relaciona de forma hierárquica. Assim, a estrutura deve possuir um objeto que compreenda esta relação de divisão do *token*.

Compreende-se que a quebra de um *token* em *tokens* menores ocorre a partir do momento em que um processo é disparado (via programação, ação do usuário, operação técnica, etc.) processando o valor do *token* gerando ao menos dois subprodutos: os *split tokens*. Deve-se entender que *split tokens* são *tokens* também, pois possuem uma relação de hierarquia com o *token*. A eles também podem ser atribuídas etiquetas, mantendo-se ainda a sua hierarquia de

constituição. Caso o *split token* não possua etiqueta morfológica, compreende-se que a etiqueta atribuída ao *token* representa o conjunto dos *split tokens*.

A estrutura possibilita a quebra recursiva de *split tokens* em unidades ainda menores, *ad infinitum*. O eDictor no estágio atual trabalha somente com um nível de recursão, isto é, um *token* pode ter de dois a *n split tokens* e *split tokens* não podem ser divididos[27].



[27] O aumento do nível de recursividade possui um impacto direto na interface de visualização das informações no eDictor. O escopo para a ferramenta foi o de trabalhar com um nível de recursividade somente, apesar de ser possível trabalhar com diversos níveis. Uma implementação recursiva sobre a divisão de *tokens* pode levar ao questionamento de quantidade de ciclos, ou ainda, quantos ciclos de recursão de divisão de *tokens* podem ser realizados, para que um *token* possa ainda ser considerado um *token*?

Figura 10 – o *token* é dividido e o *split token* entra na estrutura

O eDictor, como editor de textos, ou melhor, como editor de *tokens,* possui uma funcionalidade que permite ao usuário a inclusão de níveis de edição sobre um determinado *token*.

Esta preparação do texto ocorre pela necessidade de uma adequação para o melhor funcionamento dos módulos computacionais relacionado às regras concebidas para análise do corpus. Para o módulo computacional de etiquetagem, se um *token* de valor x está associado a uma etiqueta de valor y, esta associação será compreendida como uma regra para o processo. Se um *token* possuir o mesmo valor x, porém estiver associado a uma etiqueta de valor z, a probabilidade de que a etiqueta para o *token* de valor x seja y ou z será de 50%[28], caso um novo *token* de valor w seja inserido com a etiqueta de valor β, o modelo irá compreendê-lo como um novo *token* sem associação ao *token* anterior. Esta explicação é necessária para ficar claro que primeiro existe uma comparação entre os *tokens* no nível da *string*[29] do valor, e em relação à *string* do valor da etiqueta.

A edição de *tokens* no eDictor trabalha neste nível: na inclusão de valores relacionados à *string* do *token*. A ferramenta permite que diversos níveis de edição sejam aplicados a um único *token* e estes níveis são configurados como parâmetros do sistema através de uma interface de configurações. A escolha

---

[28] Não estou levando em consideração outros detalhes das regras probabilísticas como o próximo *token* da lista. Este é um exemplo relacionado à probabilidade de ocorrência.

[29] Mathematical term for a linear array of elements. E.g. given the set *A* = *{a, b, c}, a, baa, babac,...* are strings 'over' (that is, composed of elements of) *A*. Hence in linguistics in a sense equivalent to 'sequence'. Oxford Concise Dictionary of Linguistics.

destes níveis e como eles são utilizados se dá em função das orientações escolhidas para o corpus. Na versão de plataforma, dois níveis são determinados por *default* para funcionamento da ferramenta: a segmentação e a junção.

A segmentação é um processo semelhante ao realizado pelo eDictor, o usuário informa à ferramenta em quantos *split tokens* o *token* original deve ser dividido. Esta operação é necessária, pois algumas palavras passam por este processo para adequação da análise, por exemplo, um *token* como *d'água* passa a ser dividido em *de água*. O exemplo abaixo ilustra de forma completa este mesmo procedimento:

```
<w id="498">
      <o>d' agua</o>
      <e t="jun">d'agua</e>
      <e t="seg">de água</e>
      <m>de</m>
      <m>água</m>
</w>
```

Tabela 3 – exemplo de uma palavra em *xml*

Após o disparo da ação de edição (preenchimento do valor para segmentação e consequente confirmação), a ferramenta serializa a informação criando dois morfemas separados (identificados pela etiqueta *m*) para que possam ser enviados separadamente ao modelo computacional de etiquetagem. O mesmo

caso de segmentação é utilizado para identificação dos *tokens* que sofreram uma quebra por razão de uma marcação de quebra de linha:

```
<w id="520">
      <o>ca-<bk t="l" id="bk_65"/> pim</o>
      <e t="seg">capim</e>
      <e t="jun">ca-pim</e>
</w>
```

Tabela 4 – exemplo de uma palavra em *xml* com uma marca de quebra

O valor original do *token* é mantido e a ele são adicionados os valores nos níveis de edição.

O outro processo (presente nos exemplos acima) é o relacionado à junção que é o processo contrário ao processo de segmentação. Através dele pode-se juntar dois *tokens* modificando o valor de sua *string*. Os dois exemplos acima se dão em função da ocorrência de um espaço em branco entre os *tokens* divididos, seja pela formatação do texto (tabela 3), seja pela ocorrência da marca de quebra de linha (tabela 4), porém pode ocorrer junção por outra razão que caiba dentro da especificidade do corpus.

Quaisquer modificações de valores do *token* são realizadas em função do valor original, isto significa dizer que, uma vez realizado o processo de segmentação ou junção, qualquer valor editado será atribuído ao mesmo *token* que fora

anteriormente segmentado ou juntado. Este processo de edição ocorre em níveis.

Assim sendo, os processos de segmentação e junção são os primeiros (caso necessários) e os outros níveis de edição sempre são considerados posteriores a eles. Melhor dizendo, caso um *token* possua edições relativas à segmentação ou junção e mais alguma outra edição qualquer, as edições de segmentação e junção são consideradas primeiras em uma escala de prioridade determinada como regra da ferramenta.

Sendo assim, entende-se que propriedades de segmentação e junção são inerentes ao *token* no sentido de que transformam o seu valor de maneira que sempre são colocados a priori em relação a edições subsequentes. Em função disto, os dois passam a ser compreendidos como parte da estrutura do texto neste trabalho.

O processo de edição do tipo segmentação passa a ser compreendido como o *split token* já definido anteriormente. Já o processo de junção se torna o *joined token* atrelado ao *token* como uma espécie de atributo, porém com características especiais que são descritas a seguir.

Elevar estes dois níveis de edição para a estrutura possibilita que os valores atribuídos aos outros níveis de edição[30] passem a ser estendidos para estes outros dois objetos, ou seja, informações relevantes para análise computacional dos dados do corpus passam a ser possíveis para unidades menores que o *token*

[30] Alguns exemplos: grafia (correção de erros ortográficos), modernização (adequação do *token* a uma versão modernizada).

e também para agrupamento de *tokens*, não no sentido de *chunks*, pois este agrupamento (o *joined token*) não corresponde a um bloco que se inicia em uma posição e termina em outra à frente, mas sim a um novo valor atribuído a um conjunto de *tokens* que passa a ser compreendido como uma nova espécie de *token*.

Sendo a estrutura determinada em função da ordem linear de ocorrência dos tokens, o processo de junção deve se dar respeitando tanto esta linearidade como a sua ocorrência, isto é, ao se juntar dois ou mais *tokens* não se pode simplesmente criar uma nova instância do objeto *token*, atribuir a ele o novo valor de *string* e eliminar os *tokens* que deram origem a este novo. Este processo causaria uma perda que não faz sentido no processo de criação de corpus (na forma como foi concebido para o projeto). Um *token* pode ser ligado a outros e o primeiro *token* (aquele por onde se inicia o processo de junção), recebe os valores de edição que o pesquisador informar.

Cada *joined token* possui uma marcação que o liga ao próximo *token* da cadeia, indo até o último *token*. Sendo que todo *token* pode ser juntado, entende-se a necessidade de um atributo que indique esta ligação. Este atributo consiste em um índice para o relacionamento de um *token* à posição de outro *token*. O último *token* da cadeia se relaciona ao primeiro criando uma circularidade. *Tokens* que não possuem a característica de junção possuem o valor nulo como *default* para este atributo.

Em resumo, esta nova estruturação permite que *tokens* sejam juntados indefinidamente e que os valores de edição estejam logicamente atrelados ao

joined token e não ao composto original. Esta mesma lógica de atribuição de valores é válida para os *split tokens*, pois cada um deles pode possuir diferentes valores de edição (e também diferentes etiquetas morfológicas). A imagem abaixo (Figura 11) esquematiza este novo objeto da estrutura. Note que, apesar de identificado como entidade, o *joined token* corresponde a um valor de índice atribuído ao token e a regra de composição do *token* original compete às regras lógicas computacionais programadas que entendem que o atributo correspondente ao índice indica que (1) o *token* é um *joined token* e (2) qual o próximo *token* da cadeia.

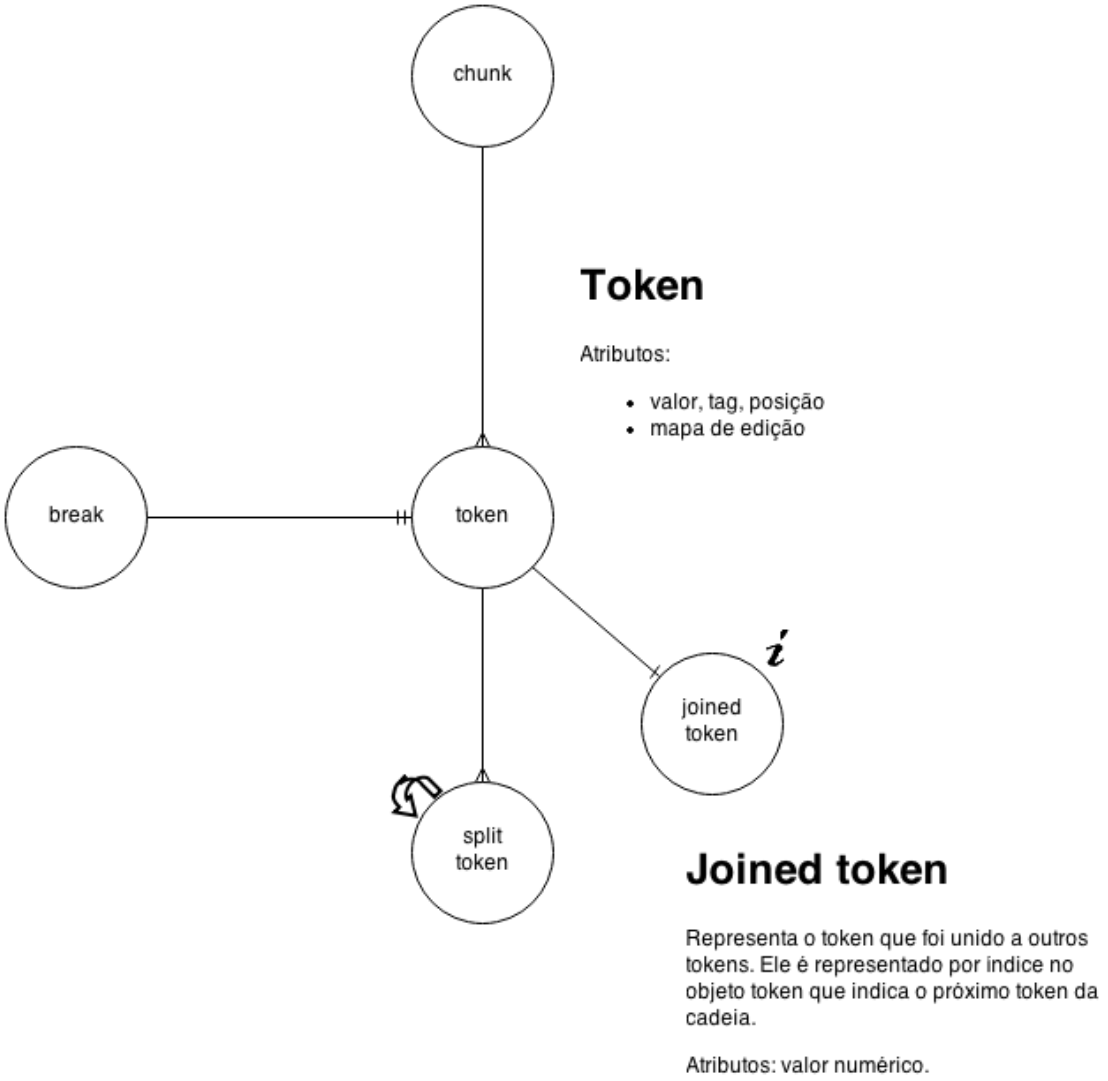


Figura 11 – introdução do *joined token*, o *token* juntado, na estrutura

Voltando à funcionalidade de edição dos *tokens*, existe a possibilidade de configuração individual dos níveis de edição, conforme explicado anteriormente. Isso leva ao questionamento de como atribuir estes valores ao *token* na estrutura, pois é evidente que esta informação diz respeito ao *token* em si, independentemente de quais orientações foram definidas para construção do corpus.

Para um *token*, os valores dos níveis de edição são mapeados em uma relação um para um, ou seja, caso o *token* passe por um processo de *grafia* ou de

modernização (por exemplo) não cabe, nesta representação, que um *token* possua dois valores distintos de *grafia* ou de *modernização*. Através desta análise, conclui-se que uma estrutura de dados adequada para este tipo de organização é uma estrutura conhecida como mapa.

Um mapa[31] consiste em uma estrutura do tipo par chave-valor, isto é, para uma chave qualquer dada, há um único valor determinado. A estrutura não possui valores pré-determinados, eles são preenchidos em função da lógica de programação desenvolvida de acordo com as regras da interface.

A utilização da estrutura de dados de mapa flexibiliza o tipo de informação que pode ser atribuído a um *token*, isto é, ao invés de se possuir no objeto *token* vários atributos designando os níveis de edição (um atributo para *grafia*, um atributo para *modernização* e, assim sucessivamente) tem-se somente um atributo mapa cujas chaves são os valores em *string* das referências dos níveis de edição e os valores relacionados a cada chave correspondem à informação fornecida pelo usuário da ferramenta.

A não utilização de uma estrutura como esta tornaria menos flexível a configuração dos tipos de edição como parâmetros da ferramenta. Eles seriam fixos na estrutura e a criação de novos tipos de edição implicaria,

---

[31] Interface Map da linguagem de programação Java com várias implementações disponíveis, sendo a utilizada neste sistema a HashMap que consiste em uma estrutura híbrida de mapa com tabela hash, esta última consiste em uma estrutura de acesso rápido aos valores do mapa através de suas chaves.

necessariamente, em um ajuste na programação e, consequentemente, em todo processo de distribuição da ferramenta[32].

3.3. Terceira parte: orientando a objetos

Recapitulando, o texto fica basicamente constituído por unidades mínimas significativas chamadas *tokens* que correspondem às palavras (ou caracteres) do texto após passarem pelo processo de *tokenização* que consiste na quebra das palavras do texto seguindo alguma regra, no caso de línguas ocidentais, esta quebra é realizada pela ocorrência do espaço em branco. O resultado deste processo é uma lista ordenada de *tokens*.

Os *tokens* podem ser agrupados em blocos, os *chunks*, que estão relacionados aos *tokens* de acordo com as posições inicial e final que caracterizam o bloco. O elemento *break* representa a quebra de linha no texto e está sempre associado a um único *token*. Além disso, *tokens* podem ser unidos em cadeia formando um *joined token* utilizando um índice que referencia a posição do *token* ao qual está na ligado na cadeia (o próximo elemento na cadeia) e ainda podem ser divididos, os *split tokens* que são unidades ainda menores, porém da mesma natureza que a do *token*.

Até agora, a estrutura veio sendo definida a partir da análise das funcionalidades do eDictor e da estruturação adotada ao longo das pesquisas

[32] No caso do eDictor como ferramenta de plataforma, ou seja, que deve ser instalada em um sistema operacional, se não fosse adotada uma estrutura de mapa para este atributo, toda vez que um atributo novo fosse criado, seria necessária a re-execução do processo de instalação. No caso de aplicações web, seria necessário subir a aplicação para o servidor.

realizadas no processo de organização da informação do projeto Tycho Brahe. A referência ao termo **objeto** foi utilizada mais de uma vez.

O conceito de objeto é fundamental dentro do paradigma de programação adotado para o projeto Tycho Brahe. Este conceito caracteriza a chamada orientação a objetos e parte do princípio de que todas as entidades programadas do sistema se dão em função do objeto, isto é, um ponto inicial a partir do qual todos os outros objetos são derivados. Assim sendo, qualquer elemento da estrutura é compreendido como objeto naturalmente, pois é inerente a uma linguagem de programação orientada a objetos que qualquer entidade modelada seja um objeto por *default*.

Uma das características mais fundamentais da orientação a objetos é a de hierarquia. Ela consiste na afirmação de que objetos herdam atributos de objetos mais gerais, por exemplo, em um sistema para gerenciamento de estacionamento, os diversos modelos de carros (em suas especificidades e semelhanças) podem ser compreendidos como instâncias de um mesmo objeto: o carro. Este, por sua vez, pode ser compreendido como um veículo e assim, sucessivamente. Através do conceito de hierarquia, ao afirmarmos que um carro é um veículo, entende-se que ele possui as mesmas características de um veículo, por exemplo, um carro tanto quanto uma bicicleta são veículos e ambos servem para locomoção. Se entendemos que um veículo serve para locomoção, logo o carro também serve. Na orientação a objetos, este conceito pode avançar indefinidamente, mas seu ponto final é o objeto caracterizando-se como entidade última, ou seja, dizer que um objeto possui as características de um

objeto não somente é redundante como irrelevante, principalmente quando se leva em conta a objetividade da programação.

Ao realizar a modelagem de uma estrutura de dados como a proposta neste trabalho, entende-se que todos os elementos são objetos dentro da programação e que eles podem herdar as características de outros objetos desde que a relação hierárquica seja criada.

As diagramações propostas para a estrutura são somente propostas, afinal existem diversos tipos de modelos[33] para designar a relação entre os objetos de programação. Os conectores entre os elementos mostram como os objetos se dispõe organizacionalmente e, na sua implementação, utilizam-se diversas estruturas de dados[34]. Além disso, é importante saber que um atributo de um objeto também é um objeto.

Sendo o conceito de orientação a objeto inerente ao sistema da linguagem de programação neste paradigma não é necessário indicar que um *token* herda as características de um objeto, isto é natural da linguagem de programação. O importante é saber que todo objeto criado pode herdar as características de outro objeto criado e que esse processo de herança pode ser indefinidamente realizado até se chegar ao objeto último. Enfim, todo objeto em uma linguagem de programação orientada a objetos possui as seguintes características:

1. Possui herança
2. Pode ser comparado.

---

[33] Área dominada pela engenharia de software e um dos mais conhecidos é o UML – Unified Modelling Language.
[34] Listas, mapas, tabelas hash, vetores, arrays, etc

A estrutura adotada até aqui também faz parte da programação, logo ela também é um objeto, afinal a lista de *breaks, tokens* e *chunks* devem pertencer a um objeto comum, a um objeto que representa toda a estrutura do texto. Este objeto foi definido com o nome de *struct*.

Dadas estas informações, a estrutura pode ser representada de acordo com a imagem abaixo.

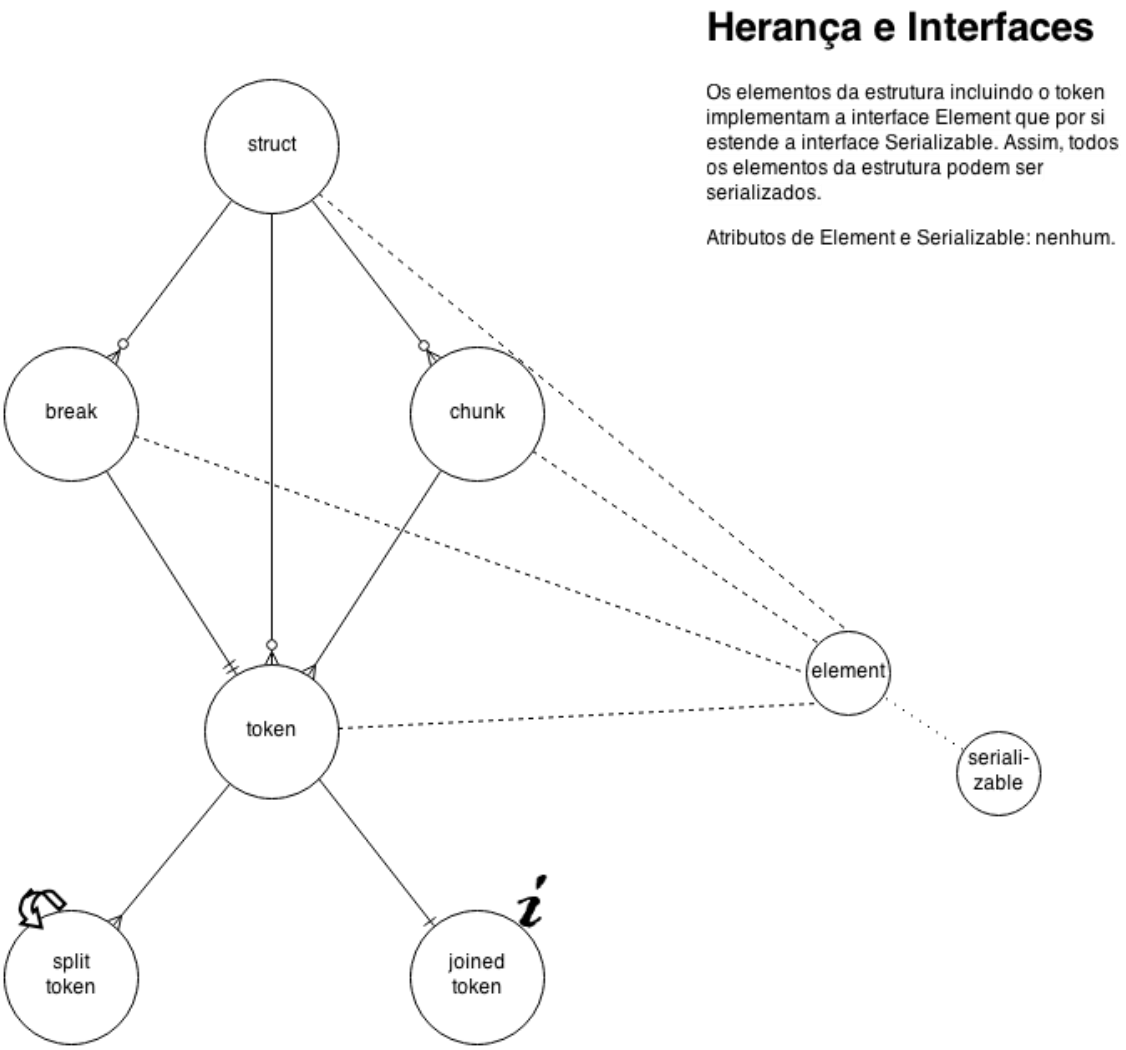


Figura 12 - estrutura orientada a objetos

A concepção da orientação a objeto ajuda na clarificação do conceito de natureza empregado no texto. Ao dizer que os *split tokens* e *tokens* são da mesma

natureza, já subentende-se que ambos são objetos, graças ao emprego da linguagem de programação, mas, além disso, sendo o *token* um objeto e tendo todo objeto a característica de possuir herança, o *split token* herda (ou estende) o objeto *token*. Assim, todos os atributos de um *token* também são característicos de um *split token*. Em Java, isto é expresso da seguinte maneira:

```
public class SplitToken extends Token
```

Estas palavras representam para a linguagem de programação a declaração de um objeto (ou classe, *class*) do tipo público[35] chamado *SplitToken* que estende outro objeto, o *Token*. Isto permite que qualquer instância do objeto *SplitToken* possua os mesmos atributos do *Token*.

Esta é, tecnicamente, a definição para o conceito de natureza que utilizo neste trabalho.

Conforme dito anteriormente, o conceito de hierarquia na orientação a objetos pode avançar indefinidamente[36] e a decisão de como se organiza esta topologia na modelagem dos objetos do sistema cabe ao responsável pela análise e implementação da lógica dos módulos computacionais e da modelagem da estrutura da informação como um todo.

Ao utilizar a estrutura de dados proposta, desconfigura-se a visão de tipologias aninhadas, o aninhamento se realiza no nível de referência de posição dos

---

[35] Java trabalha com quatro tipos de escopo: público, privado, protegido (protected) e ausente. Um objeto e atributos públicos podem ser acessados por quaisquer outros objetos, ao contrário do privado que possui o escopo mais restrito, pois só é visto dentro do próprio objeto (classe). Os escopos protegidos e o ausente são intermediários a estes dois.

[36] Não irei levantar questões técnicas relevantes a processamento ou capacidade de memória, teoricamente é possível haver a quantidade de níveis hierárquicos que seja necessária.

elementos. Passa a ser possível que elementos de mesma natureza se intercalem, ou seja, passa a ser dispensado um objeto que diferencie os parágrafos das seções e até das páginas. O objeto *chunk* cria este mapeamento referencial dos *tokens* e o texto é desconstruído e reconstruído através de lógicas programadas. Cabe à lógica de programação reestabelecer a estrutura visual do texto, independentemente de suas variações.

Além disso, esta estrutura permite realizar lógicas anteriormente impossíveis com estruturas aninhadas. Ao compreender uma página e uma sentença como *chunks* pode-se passar a entender que uma sentença se inicia em uma página e termina em outra, diferentemente do que era realizado anteriormente onde se utilizava a técnica de inclusão de uma marcação de quebra tipologicamente marcada no exato local em que ela ocorria no texto.

Neste ponto, compreende-se que existe uma clara distinção entre o *token* e os outros objetos definidos para a estrutura (*break, chunk* e *struct*), o *token* é o elemento que fornece a relação para identidade destes outros objetos, ele fornece a referência para o posicionamento (ou até mesmo existência) dos objetos estruturadores.

As linhas pontilhadas na figura 12 represnetam a implementação de uma interface. A *interface* é uma técnica da linguagem de programação que permite a criação de uma implementação genérica para os objetos (classes). Os objetos implementam uma *interface*, ou seja, seguem regras pré-determinadas que podem estar contidas na definição da *interface*. Uma *interface* diz que quem implementá-la deverá criar o corpo de regras lógicas para uma regra que ela

especifica, mas que não implementa, ou seja, a *interface* diz que algo precisa ser feito, mas não sabe exatamente **como** deve ser feito. Visto por outro lado, a *interface* é uma generalização de assinaturas, ou seja, ela consegue ser genérica o suficiente para abstrair toda a lógica da programação e, a partir daí, direcionar a implementação de suas regras lógicas de certa forma[37].

Tendo em mente isto, compreende-se que todos os elementos da estrutura precisam implementar uma *interface* única que sirva de direcionamento para a implementação de sua lógica, esta *interface* possui o nome de *element* (ou *edition element* na implementação[38]).

Em Java, escreve-se isto da seguinte maneira:

```
public class Token implements EditionElement
```

Em Java, a definição da *interface* do elemento (*element*) é:

```
public interface EditionElement extends Serializable
```

A definição desta *interface* não possui assinaturas (métodos) nem valores constantes, ela somente, por si só, estende uma outra *interface*, que consiste na atribuição da capacidade de ser serializada, ou seja, convertida para algum formato serializado, como o *xml*[39].

---

[37] Uma outra explicação (mais técnica): http://en.wikipedia.org/wiki/Interface_(Java)

[38] Dar nomes a elementos abstratos da programação é uma tarefa um pouco confusa, afinal, qual o melhor nome para todos os objetos da estrutura apresentada até o momento? Um nome como *element* é utilizado em muitos dos módulos computacionais que o sistema do projeto Tycho Brahe utiliza. Na implementação, por decisão minha, eu preferi utilizar o nome *EditionElement* para não confundir na organização automática dos *imports* de uma classe. É uma questão meramente técnica.

[39] A interface Serializable em Java informa sobre a necessidade de um índice para serialização no objeto que a implementa. Este índice é um valor utilizado para a serialização das informações, seja em *xml*, seja em *bytes* ou através de qualquer outra técnica existente. Esse índice é uma restrição técnica da linguagem de programação que o usa como uma espécie de

O texto passa a ser compreendido por listas que se relacionam logicamente em função da posição linear que os *tokens* ocupam após o processo de *tokenização*. A estrutura serializada no formato *xml* apresentaria uma formatação como a exibida abaixo[40]:

chave de comunicação entre um processo de envio e recebimento de dados. Gerar um *serialVersionUID* para cada classe que implementa a *interface* é fortemente recomendado.

[40] Este é somente um trecho de um texto e a amostra foi retirada aleatoriamente.

```
<struct tokenCount="66394" lastUid="178">
  <tokens>
    <t pos="1" value="REFLEXÕES"/>
    <t pos="2" value="SOBRE"/>
    <t pos="3" value="A"/>
  </tokens>
  <chunks>
    <ck   data-physical="page"   data-number="01"   uid="1"   ti="1"
tf="5"/>
    <ck data-logical="title" ti="1" tf="6"/>
    <ck data-physical="page" data-number="02" uid="2"/>
  </chunks>
  <breaks>
    <br tr="138"/>
    <br tr="138"/>
  </breaks>
</struct>
```

Tabela 5 – exemplo da estrutura em *xml*

O arquivo *xml* não seria mais lido como um texto em um livro. A estrutura proposta configura um desmantelamento do texto em muitos pequenos pedaços permitindo o aninhamento de qualquer grupo de *tokens*, porém atrelando a capacidade de leitura ao reconhecimento do padrão por uma ferramenta computacional.

3.4. Quarta parte: atributos lógicos e físicos de *chunks*

Adotar o *chunk* como elemento único para a relação entre *tokens*, não exclui a existência das entidades do texto na forma como se organizam, ou seja, utilizar

o objeto *chunk* não implica que deixe de existir no texto blocos de informação como parágrafo, sentença, seção, etc. Não somente eles continuam existindo, como é necessário realizar uma distinção entre eles na edição do texto. Para isto duas soluções poderiam ser adotadas: a criação dos objetos específicos para cada um dos blocos (estendendo o objeto *chunk* de acordo com a orientação a objetos) ou a criação de atributos que sirvam para a identificação de um *chunk* como sendo uma representação de um tipo específico.

A primeira opção levaria o desenvolvimento da ferramenta de volta ao seu estágio original, pois seria realizado novamente o mapeamento um-para-um da organização da informação no texto e do objeto na modelagem computacional, opção esta que foi abandonada durante a evolução da ferramenta. A segunda opção leva ao questionamento de quais, quantos e como seriam estes atributos.

A análise dos diversos tipos de dados encontrados durante o processo de importação dos documentos do projeto Tycho Brahe para o modelo da nova estrutura foi determinante na concepção destes atributos.

Mesmo que a relação entre os blocos do texto não fosse claramente determinada, de certo modo havia um padrão mais comum de hierarquia de aninhamentos: as seções continham parágrafos que continham sentenças. As seções eram frequentemente tipadas (*character, letter, carta, desc, sc_desc*) e seus valores eram determinados em razão de uma prévia verificação de quais tipos de seções o texto tratava ou poderiam ser inseridos seguindo a intuição do próprio usuário durante o processo de edição.

Na ferramenta, não há restrições quanto ao tipo de valor que pode ser associado a um elemento marcador, sendo previamente estabelecido ou não, o valor dado ao tipo era livre no sentido que era constituído por uma *string*. Em um texto cujo gênero pode ser entendido como teatral, *sc_desc* representaria a descrição de cena, porém não há impedimentos de que outro valor com mesmo sentido de verdade fosse atribuído como *scene_description*, ou ainda, *sd*. Tudo dependeria de um consenso entre os usuários.

A sistematização do texto desta forma leva a uma compreensão singular: elementos que possuem algum tipo de marcação constante no texto possuem uma maior propensão a serem considerados de maneira constante. Afinal, um parágrafo é sempre um parágrafo desde que entre um bloco de texto e outro haja alguma regra pré-estabelecida (mesmo que estética) que o determine: uma linha em branco separando os blocos ou uma tabulação especial para o primeiro *token* do parágrafo.

O reconhecimento de uma sentença já é um pouco mais complexo. O padrão não parece ser tão simples quanto a identificação de parágrafos já que mais de um símbolo pode representar a finalização de uma sentença (. ? !). E alguns questionamentos podem ser levantados. Por exemplo, outros símbolos (, ; ... :) podem representar uma marca de finalização de sentença? A marca de três pontos é constituída somente de um *token* ou ela é a composição de três *tokens* de ponto final? Em línguas como o espanhol, o processo seria diferente graças às marcas de início de sentença questionadora ou exclamativa (¿ ou ¡)?

Mesmo que estas perguntas ainda não estejam respondidas, o fato é que existe uma marcação dada pelo valor de um *token* que indica o estabelecimento de um bloco de sentença. Já para seção, não existe elemento marcador, ela pode ser criada e tipada sem a necessidade de seguir um padrão de reconhecimento já que ela é criada de acordo com a ação do usuário.

Tendo elaborada esta distinção, foram criados dois atributos para o objeto *chunk* a fim de conseguir distinguir de forma clara os dados relativos à edição. Estes atributos foram separados como físicos e lógicos, ambos formados por listas de *strings*. Um *chunk* pode não possuir nenhum valor físico ou lógico, pode possuir os dois atributos valorados ou somente um deles, não existe regra que impeça a combinação dos valores. As questões são relativas ao uso da estrutura e não àquilo que é capaz de se realizar com ela.

O que se determinou como físico para a ferramenta são os blocos que podem ser automaticamente detectados através de marcações no texto como no caso dos parágrafos e sentenças. Aqui incluindo também a página que passa a ser vista como *chunk* dentro da estrutura e que é usada pela ferramenta como elemento máximo na hierarquia de aninhamento[41]. Os atributos lógicos são valorados da mesma forma que os tipos das seções, pois não há a necessidade de se conceber a entidade seção, ela é uma redundância quando compreendida como *chunk*, basta atribuir ao *chunk* o valor da tipologia, ou seja, um *chunk* passa a possuir um atributo lógico *scene_description*, por exemplo.

---

[41] Atualmente estou analisando a possibilidade da determinação do elemento máximo da hierarquia ser estabelecido via parâmetros.

De um ponto de vista lógico, nada impede que um parágrafo seja constituído de somente uma sentença. Neste caso, pergunta-se: seriam criados dois *chunks* contendo as mesmas posições inicial e final de *tokens*? Esta situação deve ser previamente determinada em regra na ferramenta, pois ela poderia entrar em uma situação de erro.

Caso tal regra exista e o usuário tente quebrá-la, o sistema iria entrar em um fluxo de erro e, caso o erro não fosse tratado, o usuário seria alertado sobre o erro podendo ocorrer um encerramento abrupto da ferramenta, a exibição de uma caixa de alerta ou a aparição de uma mensagem de *stakcktrace*[42], etc. Se a regra de restrição não existir, será criado um novo *chunk* repetido em relação às posições determinadas, mesmo que pudesse possuir valores distintos para os atributos físicos ou lógicos.

A regra programada para o eDictor não permite a criação de dois *chunks* com valores iguais de posição inicial e final dos *tokens*, porém esta é uma regra da ferramenta, nada impede que a estrutura possua tais *chunks*. Os valores de atributos físicos são constituídos de *strings* que determinam se um chunk é um parágrafo e uma sentença ao mesmo tempo, pois parece também que nada impede que isto ocorra no texto.

Assim, para a estrutura, o *chunk* não é somente visto como um elemento estruturador, mas ele também é visto sob a ótica de um acumulador de significação. Os atributos de lista são virtualmente infinitos, a eles é possível

---

[42] Mensagem de erro que informa o caminho que a ação tomou em relação às linhas de código percorridas.

atribuir tantos valores quanto caibam na memória da máquina. E sob a compreensão da diferença entre os atributos físicos e lógicos, entende-se que a lista de atributos lógicos possa ser muito superior à de físicos em razão das próprias condições materiais restritas do texto em um livro como ele é, porém será que é possível que existam tantas possibilidades físicas quanto lógicas? De qualquer maneira, os atributos lógicos são considerados livres e tratados sob a ótica da lógica de um processo muito utilizado na web conhecido como *tagging*[43]. Este processo consiste na atribuição de quaisquer palavras (*tags*) que caracterizem semanticamente uma informação. Ele é basicamente o mesmo conceito das *keywords* em um *abstract*, porém o modelo já é concebido para alimentar um processo de organização sistemática e eficiente na busca de informações.

Por outro lado, os atributos físicos, na implementação do sistema, são constantes de código, isto é, a *string* que determina internamente para a ferramenta o valor que ela, via regra de programação, entende como um valor de parágrafo, é fixada no código.

Estas constantes são valores atribuídos a uma variável considerada estática e que depende inteiramente do programador que a determinou com um valor específico. Caso este valor tenha que ser alterado, é necessário que o programador modifique o código-fonte da ferramenta e realize todo o processo natural de *deploy*[44] da aplicação. O caminho mais natural para uma constante é

---

[43] Não confundir com a etiquetagem morfológica, são processos diferentes.
[44] Consiste no processo de compilação do código-fonte, empacotamento dos códigos e posterior "subida" da aplicação para o servidor.

que ela se torne um parâmetro do sistema, possuindo uma flexibilidade muito maior já que o valor poderia ser alterado *on-the-fly*, ou seja, no momento da execução. A ferramenta pode comportar um próprio programa de parâmetros (como já estabelecido no eDictor em plataforma nos seus princípios de configuração) e este sistema (integrado à ferramenta) pode possuir uma interface própria possibilitando a alteração destes valores sem a necessidade de se realizar um novo *deploy* da aplicação no servidor[45].

A especificação dos valores de atributos físicos em constantes foi determinada de forma arbitrária para a ferramenta eDictor, pois a ela parecem convencionadas certas formas como a sentença e o parágrafo e até mesmo a forma de página. A página é o elemento determinante a partir do qual o usuário inicia seu processo de edição, enquanto a sentença é crucial para a realização dos processos de etiquetagem e *parsing*. Apesar da ferramenta utilizar o parágrafo como forma física, até o momento ele não parece ser essencial para o fluxo do processo.

---

[45] Em trabalho futuro irei entrar em maiores detalhes do sistema de parâmetros do projeto Tycho Brahe. Mas, somente como observação experimental, até que uma variável se torne constante, ela parece passar por estágios no desenvolvimento. Primeiro ela seria uma variável comum em meio ao corpo do código com um valor fixo atribuído. Estas variáveis são conhecidas como *hard-coded*, ou seja, um valor definido arbitrariamente atribuído a uma variável "jogada" em meio ao mar de código do sistema. Depois ela passa a um status de constante, onde ela é destacada do corpo do código e permanece em posição (visual) destacada ou é concentrada em um único repositório de constantes. O próximo passo é associar a constante a um parâmetro flexibilizando a atribuição de valores e permitindo que diversos usuários do sistema possam atribuir valores específicos para uma mesma variável do sistema de acordo com as regras às quais os parâmetros estão relacionados. De qualquer forma, é muito mais elegante que um sistema seja baseado em um sistema de parâmetros do que co-exista com uma infinidade de variáveis *hard-coded*.

3.5. Quinta parte: a estrutura sintática e a tridimensionalidade dos *chunks*

O último atributo necessário para o *chunk* e para integração de todas as informações utilizadas no projeto Tycho Brahe utilizando-se o eDictor está relacionado à integração da estrutura sintática das árvores à estrutura proposta.

A anotação sintática corresponde à "identificação e codificação da estrutura sintagmática"[46] das sentenças do texto. A anotação ocorre via processos semiautomatizados (conforme descrito no apêndice A) através da utilização de componentes automatizados. Abaixo segue um exemplo de anotação sintática:

```
( (IP-MAT (NP-SBJ-1 *pro*)
          (VB-P parecem-)
          (NP-SE (CL -se))
          (PP (P n@)
              (NP (D-F @a) (N feição)))
          (PP (P com)
              (NP (N-P pepinos)))
          (, ,))  (ID G_008,17.202))
```

Figura 13 - Anotação sintática da oração 17.202[47]

Esta visualização da árvore sintática é conhecida como *symbolic expression*[48] e corresponde a uma estrutura de dados conhecida como árvore[49].

Toda árvore, binária ou não, é descrita como possuindo um nó raiz a partir do qual se derivam os nós filhos, recursivamente, até atingirem os nós folhas. Nós folhas são nós que não possuem filhos.

---

[46] Paixão de Sousa, 2014.
[47] Paixão de Sousa, 2014.
Veja-se www.tycho.iel.unicamp.br/~tycho/corpus/texts/psd/txt/g_008_psd.txt para a anotação completa deste texto.
[48] http://en.wikipedia.org/wiki/S-expression
[49] http://en.wikipedia.org/wiki/Tree_%28data_structure%29

Na estrutura sintática adotada pelo projeto Tycho Brahe, os nós folhas são os *tokens*. Os nós intermediários entre a raiz e os *tokens* são as categorias sintagmáticas às quais os *tokens* estão atrelados.

Cada nó se encontra em um nível em relação ao nó raiz da árvore. Este valor é fundamental para identificação e reconstituição dos *chunks* sintáticos. Mas primeiramente irei descrever a relação entre a árvore sintática e os *chunks* sintáticos.

Para decompor a árvore sintática em *chunks* sintáticos é necessário pensar que a partir de qualquer nó não terminal da árvore (todos exceto os nós folha) atinge-se, sempre, dois *tokens* nas extremidades percorrendo todas as conexões de seus nós filhos. Basta seguir uma orientação *top-down* na vertical e *left-right* na horizontal. Todo *chunk* sintático é binário nesse sentido, pois não é a quantidade de nós filhos que determina sua binariedade, mas a regra estrita de que ele pode ser determinado por somente dois *tokens*. O *chunk* mantém sua condição de relação binária com *tokens* tanto no discurso quanto na sintaxe. O que diferencia um do outro é tridimensionalidade do nível em que o *chunk* sintático se encontra em relação ao nó raiz da árvore sintática.

Assim, foi necessária a inclusão de um atributo para o *chunk*, um atributo que corresponde ao nível em que o *chunk* está em relação ao nó raiz. Na implementação do projeto Tycho Brahe, os valores dos níveis são arbitrariamente definidos em função da unidade do número 1. O nó raiz se encontra no nível zero e para cada nível da árvore acrescenta-se o valor numérico 1.

Para os *chunks* não sintáticos, o valor do nível é o valor *default* nulo, ou seja, não existe uso para ele neste nível.

Chamar o *chunk* sintático de tridimensional deriva exatamente da inclusão deste novo atributo na estrutura. Sem ele seria impossível remontar a estrutura sintática para apresentação de árvore (vide apêndice C). Dois *chunks* podem possuir os mesmos *tokens* na sua posição inicial e final, conforme descrito anteriormente, porém a ocorrência deste tipo de situação é muito mais frequente nas estruturas sintáticas. Uma categoria sintagmática pode possuir os mesmos *tokens* de seu nó filho caso ele só possua um filho. Diferenciar os dois *chunks* que compõem estas categorias só é possível através da informação do nível. Segue abaixo três exemplos de como uma árvore pode ser decomposta em *chunks*.

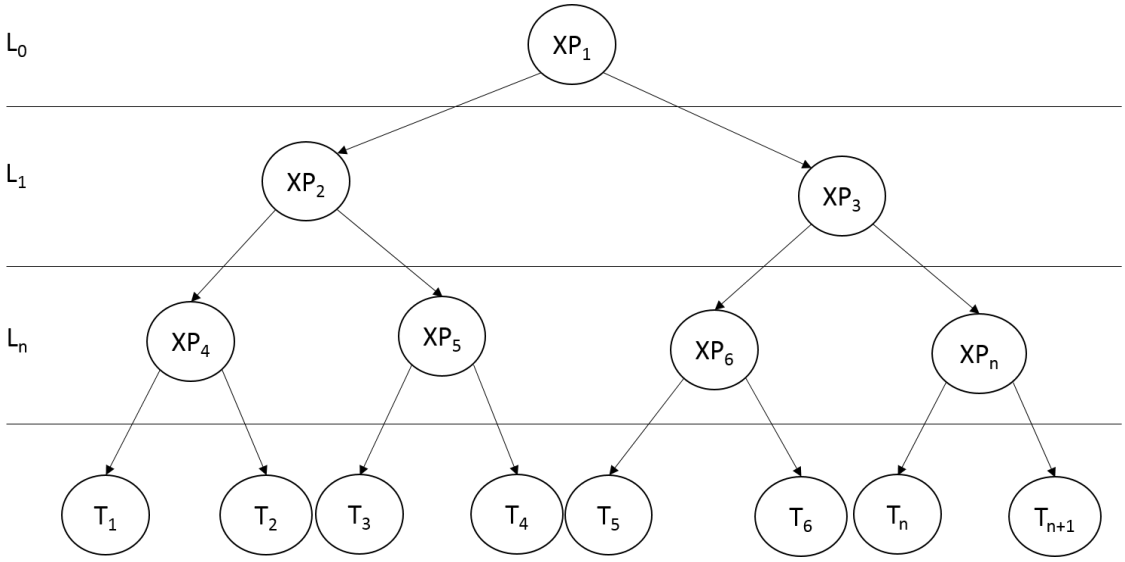


Figura 14 – árvore binária de n níveis com n categorias XP e n *tokens*

chunks	*nível*	*token inicial*	*token final*
chunk XP_1	L_0	T_1	T_{n+1}
chunk XP_2	L_1	T_1	T_4
chunk XP_3	L_1	T_5	T_{n+1}
chunk XP_4	L_n	T_1	T_2
chunk XP_5	L_n	T_3	T_4
chunk XP_6	L_n	T_5	T_6
chunk XP_n	L_n	T_n	T_{n+1}

Tabela 6 - chunks da árvore binária da Figura 14

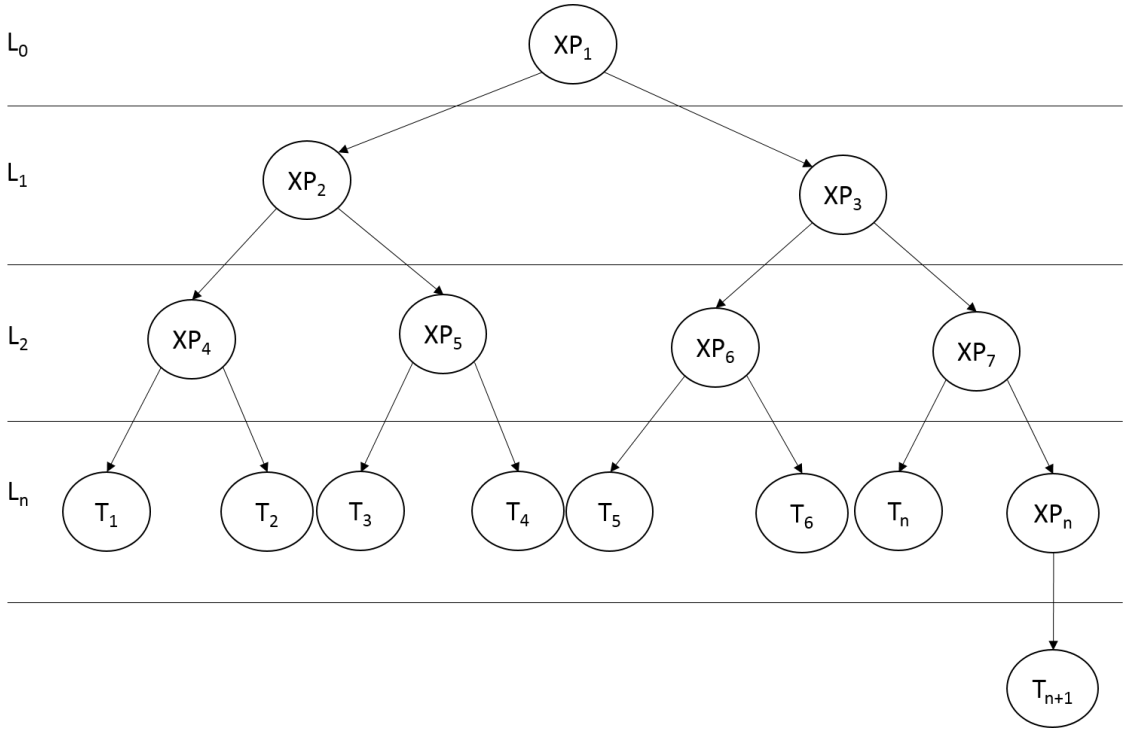


Figura 15 - árvore binária de n níveis com n categorias XP e n *tokens*

chunks	*nível*	*token inicial*	*token final*
chunk XP_1	L_0	T_1	T_{n+1}
chunk XP_2	L_1	T_1	T_4
chunk XP_3	L_1	T_5	T_{n+1}
chunk XP_4	L_2	T_1	T_2
chunk XP_5	L_2	T_3	T_4
chunk XP_6	L_2	T_5	T_6
chunk XP_7	L_2	T_n	T_{n+1}
chunk XP_n	L_n	T_n	T_{n+1}

Tabela 7 - chunks da árvore binária da Figura 15

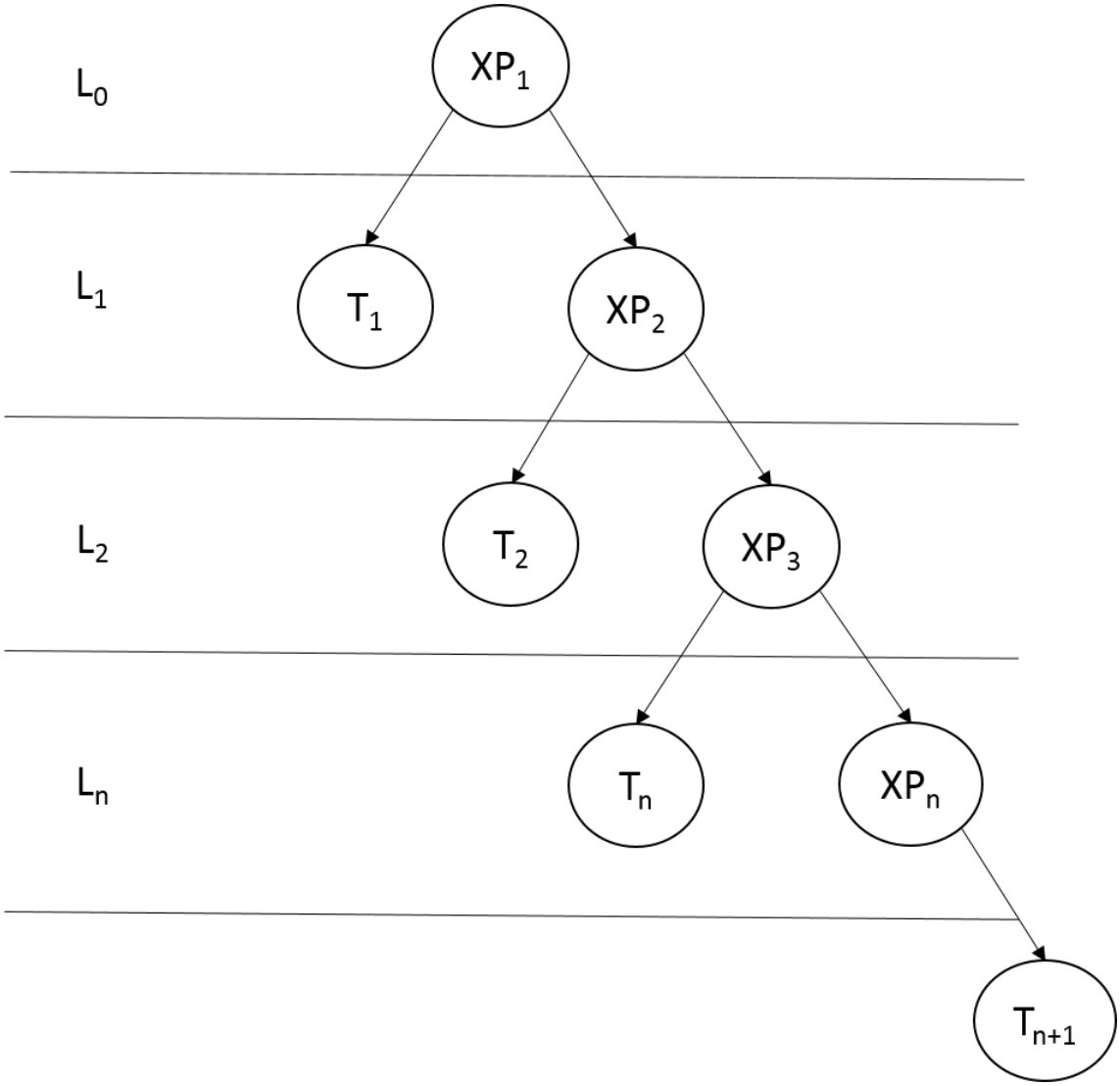


Figura 16 - árvore binária de n níveis com n categorias XP e n *tokens*

chunks	*nível*	*token inicial*	*token final*
chunk XP_1	L_0	T_1	T_{n+1}
chunk XP_2	L_1	T_2	T_{n+1}
chunk XP_3	L_2	T_n	T_{n+1}
chunk XP_n	L_n	T_{n+1}	T_{n+1}

Tabela 8 - chunks da árvore binária da Figura 16

4. eDictor: da plataforma para a nuvem

O objetivo deste capítulo é descrever o funcionamento tecnológico do eDictor em relação às técnicas de desenvolvimento de softwares de arquitetura web mostrando como foi possível a transposição desta ferramenta de sua versão plataforma (i.e. dependente de um sistema operacional), para sua versão na nuvem, ou seja, como foi possível incorporar à ferramenta os conceitos e tecnologias utilizadas em outro tipo de desenvolvimento de software. Este é o capítulo mais técnico no sentido computacional e ele é necessário para que se compreenda como se dá o funcionamento de aplicações de arquitetura web.

Esta apresentação se dá a partir da análise das diferenças entre ferramentas de plataforma e web introduzindo a utilização da estrutura descrita no capítulo anterior até chegar na descrição das tecnologias de interface. Por fim, é realizada uma crítica em relação à tecnologia *xml* e é explicado o conceito de serialização.

Este trabalho foi realizado durante os anos de 2012 e 2014 enquanto bolsista pela Fapesp[50] juntamente com a equipe de trabalho do corpus Tycho Brahe. O código fonte do projeto encontra-se em processo de abertura, pois ainda está restrito à equipe de desenvolvimento. Futuramente ele será aberto a toda comunidade científica.

Recomenda-se a leitura prévia do apêndice B, pois ele serve como manual de usuário do eDictor descrevendo as telas e funcionalidades do sistema tornando o leitor mais familiarizado com suas particulariedades.

---

[50] Processo Fapesp 2012/19756-5

4.1. Primeira parte: introdução

A ideia central deste capítulo é explicar o funcionamento da ferramenta eDictor na sua nova versão web descrevendo os passos de sua passagem da plataforma para a nuvem.

Quando uma ferramenta é desenvolvida em plataforma significa dizer que ela foi feita em função de um sistema operacional. O usuário realiza o *download* do instalador da ferramenta para determinado sistema operacional e executa o processo de instalação. Uma vez instalada a ferramenta, o usuário pode passar a utilizá-la em seu computador sem a necessidade de um servidor e de conexão em rede para realizar qualquer operação.

Ao realizar a transição das funcionalidades de uma ferramenta de plataforma para a web, este processo de *download* e instalação passa a não ser mais necessário. O usuário acessa os seus documentos em uma base comum distribuída na "nuvem", ou seja, ele precisa somente de um navegador *web* e de acesso à rede da Internet.

Quando se diz que algo está na "nuvem" significa dizer que o usuário não sabe determinar exatamente onde suas informações estão fisicamente armazenadas, tampouco se elas estão ou não fragmentadas em diversos menores pedaços espalhados por muitos outros computadores. Ele simplesmente usa estas informações.

Uma das maiores vantagens de uma aplicação ser *web* é a de poder controlar a versão de uma aplicação de modo muito mais simples, pois existe somente aquela única implementação computacional do sistema.

O funcionamento básico de acesso *web* é este: o usuário requisita um serviço acessando uma *url*[51], um servidor base compreende as palavras da *url* e direciona a requisição para uma máquina disponível na rede sob um determinado endereço. Este endereço é o IP (*Internet Protocol*), uma série de números que indicam a posição da máquina servidor na rede *web*. Uma vez que o servidor recebe a requisição, ele processa os dados enviados através da *url* e direciona para os serviços configurados. Este direcionamento leva a requisição do usuário para compreensão (tradução) das palavras de acordo com a lógica interna programada do sistema. O sistema processa as informações e devolve um conjunto de dados que percorre de volta todo o caminho até ser decodificado no navegador do usuário em sua máquina.

Este conjunto de dados analisados pelo navegador do usuário consiste naquilo que se chama de operações no cliente, enquanto todo o resto são operações no servidor.

Assim, o sistema é sempre único, ou seja, a versão disponibilizada *online* é a mesma para todo usuário que acessá-la.

---

[51] Todo caminho digitado no navegador. Os mais comuns começam com "http://www".

Este foi um ponto fundamental na decisão de transposição da ferramenta eDictor para uma plataforma *web*. Assim, começaram a ser realizadas análises de viabilidade tecnológica para execução da implementação.

Uma das questões levantadas foi sobre como fazer para apresentar a estrutura de dados adotada com a possibilidade de *chunks* intercalados através de uma linguagem de marcação como a da *web*, ou seja, como exibir os dados em uma página *html*[52] padrão dos navegadores atuais.

A linguagem de marcação *html* possui as mesmas restrições que o *xml* em relação ao aninhamento das informações. São linguagens de marcação da mesma forma. E ainda existem outras questões técnicas: posicionamento dos elementos na área de visualização que diferem navegador a navegador (tamanhos de margem, *padding*[53]) e a utilização de lógica em *javascript*[54] e de estilo em *css*[55]. Estas três tecnologias (*html, css, javascript*) operam em conjunto para que as funcionalidades possam ser reproduzidas na nova plataforma. A tecnologia atual de desenvolvimento *web* permite hoje que algo compreendido como estático (*html*) possua o dinamismo necessário para execução de operações complexas para manipulação lógica de elementos em sua interface.

Após a definição das metodologias técnicas para execução das operações, inicia-se uma etapa de aperfeiçoamento da interface homem-máquina, isto é, como apresentar, na área visual (a tela do computador), todas as informações de tal maneira que seja o mais intuitivo possível para o usuário. Não é somente uma

---

[52] W3C, http://www.w3.org/html/
[53] Margem interna do elemento *html*.
[54] Manipulação dos elementos estáticos de uma página para que possam ser dinamizados.
[55] *Cascade Style Sheet*. São atributos referentes à estética da apresentação dos elementos *html*.

questão de desenvolver as funcionalidades e garantir o funcionamento do processo, mas também o de realizar a integração entre as necessidades do usuário e o conhecimento técnico do programador. Neste momento, surge a figura do arquiteto de sistemas ao qual cabe a compreensão do sistema e a análise de como ele deve ser trabalhado para transpor para a máquina (dentro de suas limitações) esta forma de uso, transformando lógicas programadas em uma ferramenta de trabalho eficiente e dinâmica.

Um aspecto técnico que deve ser ressaltado antes de entrar em maiores detalhes do funcionamento da ferramenta está relacionado à utilização da tecnologia *AJAX*[56]. A sigla significa *Asynchronous Javascript And XML* (*Javascript* e *XML* assíncrono) e está associada a uma tecnologia popularizada a partir de 2007 que permite que requisições assíncronas sejam realizadas entre o cliente (navegador) e o servidor (central) possibilitando que diversos elementos da página sejam manipulados independentemente de outros. Um exemplo de utilização desta tecnologia são os *chats* online de redes sociais que funcionam independentemente de outras áreas do sistema. É possível realizar a sessão de bate-papo trocando mensagens sem a necessidade de uma total requisição da página *web*. Esta tecnologia permitiu a modernização da Internet para a forma como a conhecemos hoje.

O eDictor em sua versão *web* possui basicamente as mesmas funcionalidades de sua versão plataforma, mas permite trabalhar com uma maior flexibilidade já

[56] http://en.wikipedia.org/wiki/Ajax_(programming)

que os elementos são mais livres, já que são definidos pelos elementos da estrutura apresentada neste trabalho.

A fase de transcrição permanece basicamente a mesma, porém foi incluída uma funcionalidade que permite a inclusão de caracteres que não podem ser digitados através de um teclado comum e o aperfeiçoamento da manipulação de imagens.

A maior parte do desenvolvimento acabou sendo na edição onde buscou-se diminuir a utilização do mouse trazendo as funcionalidades para atalhos do teclado e diminuindo a distância entre os elementos na tela para que haja menos deslocamento do cursor do mouse. As operações são realizadas *token* a *token* com diversas funcionalidades. Não é mais necessário salvar o trabalho realizado, pois a concentração das ações no *token* retira qualquer operação da edição do texto.

4.2. Segunda parte: da estrutura para um formato de marcação

Esta seção irá tratar do caminho de volta da estrutura definida para um formato de marcações. Parece estranho imaginar este caminho, mas não. Ele é necessário em função da forma como a interface *web* funciona. Ela utiliza a linguagem *html*.

Como descrito anteriormente, o arquivo em formato *xml* é a serialização de uma estrutura de dados em um formato de marcação estendida, ou seja, é possível determinar quaisquer valores para as etiquetas desde que elas sejam abertas e fechadas corretamente, logo podemos dizer que a regra de seu formato estrutural é única. Sistemas com interfaces desenvolvidas para navegadores

web[57] adotam o formato *html* como padrão para exibição da informação. Este formato também é composto de marcações, porém mais rígidas, não podendo ser atribuído qualquer nome ao elemento[58].

Para realização do eDictor web, obviamente que todas as informações contidas na estrutura de dados adotada devem ser apresentadas através do formato *html*, por isso a preocupação com a questão do retorno da estrutura para um formato de marcações.

O trajeto que levou à fragmentação do texto em blocos lógicos relacionados a uma unidade mínima significativa é o mesmo que levou a pesquisa a atribuir a esta unidade (*token*) as informações necessárias para converter a estrutura para o formato de marcações html da interface.

Não seria possível encapsular espacialmente os blocos lógicos da estrutura para apresentação em formato sequencial, de certa forma o *token* deveria possuir em si a informação que indica que ele corresponde a determinados blocos. A solução, portanto, foi atribuir ao *token* indicações de que ele está encaixado em blocos, como atributos para visualização.

Em *html*, para cada marcação é possível atribuir valores a atributos que seguem padrões convencionados de nomenclatura[59], além disso, é possível manipular

---

[57] Os mais comuns atualmente são o Chrome da Google, Firefox da Mozilla Foundation, Internet Explorer da Microsoft e o Safari da Apple, porém existem muitos outros: http://en.wikipedia.org/wiki/List_of_web_browsers

[58] Apesar da versão 5 do formato *html* ser mais flexível em relação a isso. É importante notar que um navegador *web* é um software capaz de identificar um formato de marcações, no caso o HTML5 foi adotado por um consórcio mundial, o W3C e as empresas costumam adotar acordos internacionais de uso do formato, porém não significa que um navegador é *full-compliance* em relação a seu uso podendo gerar incompatibilidades no funcionamento dos sistemas entre um navegador e outro.

de maneira muito eficiente os marcadores através de valores atribuídos a classes de estilo[60]. As imagens abaixo exemplificam de forma mais clara seu funcionamento.

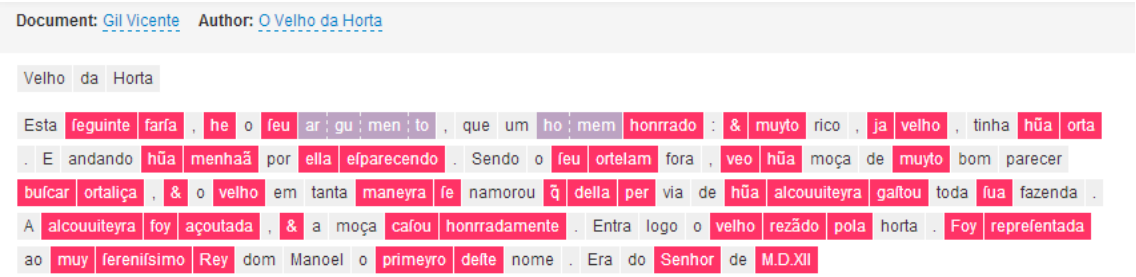


Figura 17 - html visualizado no navegador

```
<div id="divTokenEdition" class="display-page-inner ui-selectable">
 <label class="token ui-selectee ui-widget-content" data-pk="1">Velho</label>
 <label class="token ui-selectee ui-widget-content" data-pk="2">da</label>
 <label class="token ui-selectee ui-widget-content" data-pk="3">Horta</label>
 <div class=" clearfix line-break" data-pk="3"></div>
 <div class=" clearfix line-break" data-pk="3"></div>
 <label class="token ui-selectee ui-widget-content" data-pk="4">Esta</label>
 <label class="token edited ui-selectee ui-widget-content" data-pk="5" data-writing="seguinte">ſeguinte</label>
 <label class="token edited ui-selectee ui-widget-content" data-pk="6" data-writing="farsa">farſa</label>
 <label class="token ui-selectee ui-widget-content" data-pk="7">,</label>
 <label class="token edited ui-selectee ui-widget-content" data-pk="8" data-modernization="é">he</label>
 <label class="token ui-selectee ui-widget-content" data-pk="9">o</label>
 <label class="token edited ui-selectee ui-widget-content" data-pk="10" data-writing="seu">ſeu</label>
 <label class="token edited hide ui-selectee ui-widget-content" data-pk="11" data-expansion="2" data-modernizati
 data-writing="1" data-standardization="6" data-punctuation="4" data-tag="HV">argumento</label>
 <label class="token split ui-selectee ui-widget-content" data-pk="11" data-original="ar" data-ref="11" data-spl
 <label class="token split ui-selectee ui-widget-content" data-pk="11" data-original="gu" data-ref="11" data-spl
 <label class="token split ui-selectee ui-widget-content" data-pk="11" data-original="men" data-ref="11" data-sp
 <label class="token split ui-selectee ui-widget-content" data-pk="11" data-original="to" data-ref="11" data-spl
 <label class="token ui-selectee ui-widget-content" data-pk="12">,</label>
 <label class="token ui-selectee ui-widget-content" data-pk="13">que</label>
 <label class="token ui-selectee ui-widget-content" data-pk="14">um</label>
```

Figura 18 – trecho do código fonte do html da figura anterior

O módulo computacional do eDictor responsável pela conversão dos dados verifica quais são os *chunks* aos quais o *token* está associado e atribui ao *token* os valores que o identificam como parte de um determinado *chunk*. Não se cria um

---

[59] A partir do HTML5 passou-se a adotar a marcação data-*, por exemplo, um atributo pode ser chamado de data-pk e possuir uma chave que identifica o dado para utilização na programação.
[60] Em *html*, existe um atributo *class* que define nomes para marcações de estilo em *css*. Através do *framework Jquery*, é possível realizar manipulações com os objetos através dos valores dados nos seus atributos.

encapsulamento invisível das informações para que determinada ação (um clique para visualização de todas as sentenças ou seções) faça-o visível ou, ainda, não são criadas estruturas temporárias para visualização (também seria possível identificar os elementos e adicionar um encapsulamento *on demand*). A ferramenta disponibiliza ações para visualização que, quando acionadas, alteram configurações de estilo (opacidade, cor do fundo ou do texto) que destacam o elemento, dando o efeito de encapsulamento.

Note que isto é muito diferente de criar a estrutura de encapsulamento e demonstra que, a partir dos *tokens*, é possível realizar a identificação de um bloco, ao menos no nível de visualização da informação.

Finalizando, a *tokenização* que dá início a todo processo de análise do corpus através da identificação do agente separador (espaço em branco nas línguas ocidentais) e quebra em *tokens* transformando-os em uma lista, pode ser compreendido como um processo de uma única via. Não parece haver um processo de *destokenização*. Uma vez que o objeto se transforma em um *token*, o processo de conversão que faz possível a visualização continua utilizando o *token* como unidade, o dado apresentado ao usuário que utiliza a interface é uma instância do objeto *token*, mesmo que representado sob um formato de marcação. Isto se dá pelo fato de que não é necessário realizar a reintrodução do elemento separador, a separação visual entre os elementos na visualização se dá em função de atributos de estilo que o configura (como margens).

Apesar da transformação da estrutura de volta em texto não parecer relevante, seria necessária uma lógica reversa àquela da *tokenização*, isto é, a inserção de

um espaço em branco após cada valor do *token* e criação de uma regra de exceção para as marcações de pontuação (que não costumam ser separadas). Tudo isto deveria ser programado para obter o resultado esperado e, no caso de línguas que não utilizam o espaço em branco, a regra seria um pouco mais complexa. Porém algo é intrigante neste processo, após o último elemento da lista, ou seja, o último *token* do texto, a regra de programação iria inserir um elemento separador extra. Para evitar isto, deveria ser realizada uma lógica de verificação do último elemento para que não seja inserido um espaço em branco extra (em línguas ocidentais) após o último *token*, caracterizando o famoso *halting problem* de Alan Turing[61]: o problema da parada. Não existe este último espaço em branco quando o texto é *tokenizado*, ou seja, são processos diferentes que envolvem uma regra que não é conhecida a priori[62].

4.3. Terceira parte: a interface de edição

Neste capítulo são descritos os processos e tomadas de decisão em relação a como as funcionalidades da ferramenta eDictor foram integradas ao processo de conversão entre a estrutura e o formato de marcação. Também são endereçadas algumas informações sobre a relação cliente-servidor que é estabelecida na comunicação da interface com os módulos lógico-computacionais.

Nesta altura, é importante compreender um pouco do funcionamento de uma arquitetura *web*. Um navegador *web* funciona como um cliente, ou seja, ele

---

[61] Turing, 1936.
[62] Somente para programadores: esse é um *if* que incomoda na programação.

realiza requisições no servidor de acordo com sua necessidade. Este *software* envia e recebe informações através de uma rede lógica (e física) a um servidor localizado em algum lugar. Como não é necessário (ou, às vezes, possível) determinar a localização deste servidor, costuma-se dizer que ele está na nuvem. Mas esta é uma força de expressão, o dado trafega sob fios ou ondas (rádio, satélite), passando por várias máquinas (*backbones*, roteadores, servidores, etc.) até atingir um destino, uma máquina que irá servir o cliente dando a ele aquilo que ele pede. De maneira muito grosseira, é assim que funciona um sistema cliente-servidor web.

O navegador web, sendo um *software*, realiza processamentos (graças aos módulos computacionais programados que o compõem). Estes processamentos são realizados no cliente, ou seja, localmente, de acordo com a capacidade de processamento da máquina do usuário. Sempre existe um processamento local, a própria transformação dos dados recebidos do servidor em informação compreensível para o usuário é realizada em um processo de conversão do navegador. Além disso, muitas outras operações são possíveis envolvendo manipulações dinâmicas das informações (via *javascript*) e de estilos (via *css*).

A manipulação de dados ocorre no cliente, mas eles devem ser enviados para o servidor em algum momento, para que sejam atualizados na estrutura e, consequentemente, no banco de dados. A comunicação entre o cliente e o servidor pode ser realizada de duas formas diferentes. A primeira forma constitui a criação de uma *url* de acesso, isto é, um *link* em *http* com todos os

valores dos dados concatenados[63]. Este método é conhecido como *get* e é um meio comum de acesso a páginas web, por exemplo, quando um usuário acessa qualquer página na internet digitando um caminho na barra de endereços do navegador, está sendo realizada uma chamada do tipo *get* ao servidor.

A outra forma de realizar o processo é através de uma chamada *post*. Um *post* consiste no envio de um formulário (um grupo de informações) ao servidor. Ao contrário das chamadas do tipo *get*, uma chamada do tipo *post* não realiza a concatenação dos valores de todos os atributos em uma *string* que constituiria a chamada *http* da *web*. Chamadas do tipo *get* costumam enviar menos dados, chamadas do tipo *post* não possuem um limite pré-definido e somente elas podem ser utilizadas para se fazer *upload* de arquivos.

Tanto o *get* quanto o *post* podem ser realizados de forma síncrona ou assíncrona. Chamadas síncronas são aquelas que submetem a página *html* toda para o servidor, a grosso modo, são aquelas que, quando realizada uma ação (clique em um botão, *link*, etc.), toda a página é alterada. Caso uma página seja submetida a ela mesma (e caso a internet possua uma conexão rápida), o efeito é de que houve um "piscar" na tela. As chamadas assíncronas funcionam de forma diferente, elas são realizadas em *background*, ou seja, não ocorre este efeito de "piscar". É possível submeter um dado ao servidor (e obter retorno) e realizar a modificação de somente um elemento na área de visualização[64].

---

[63] Por exemplo:
http://www.tycho.iel.unicamp.br/tbf/mestrado.html
[64] Reitero que isto só foi possível com o avanço da tecnologia AJAX que revolucionou a Internet aproximadamente a partir de 2007.

O eDictor, em sua versão *web*, utiliza todas estas formas de comunicação entre cliente e servidor de forma mista, porém a comunicação da área que envolve a edição do texto, ou seja, o processo de conversão entre a estrutura e o formato de marcações, é realizada essencialmente através de chamadas *post* assíncronas.

As ações do usuário no eDictor web são realizadas no nível do *token*. Após a página ser carregada exibindo todo seu conteúdo, as chamadas ao servidor ocorrem somente quando uma ação é realizada no *token*. O usuário clica no *token* e seus dados são exibidos. O usuário aplica uma operação ao *token* (edição, divisão ou união) e os dados são salvos no servidor.

Uma importante característica da ferramenta é que ela possibilita a edição simultânea de páginas de um documento, isto é, dois usuários podem trabalhar simultaneamente no mesmo documento. Isto somente é possível através de uma regra implementada que altera os números das posições dos *tokens* para a edição, ou seja, enquanto se está realizando a edição de uma página, no cliente trabalha-se como se esta página fosse única, não se utiliza como referência a posição do *token* na estrutura, mas sim uma posição na interface. Assim, quando é realizada uma ação no servidor, a primeira regra do algoritmo é a de reestabelecer a ordem das posições dos elementos somando-se a cada posição da interface (iniciada em 1) o valor do *token* inicial da página (*chunk*) a que eles correspondem, ou seja, se a página editada se iniciar no *token* de posição 231, basta somar 231 a cada posição de cada elemento para reestabelecer sua posição.

Por fim, a estrutura é serializada em *bytes* e armazenada na base de dados relacionada ao documento em questão.

4.4. Quarta parte: a serialização e o abandono do *xml*

Tendo em vista os atributos dos *chunks*, a possibilidade de atribuição múltipla de valores a eles, o relacionamento realizado entre as várias possibilidades físico-lógicas no texto entre seus *tokens*, as relações de aninhamento que podem ser criadas entre os *chunks* e, obviamente, a quantidade de informação que um texto pode possuir, o formato serializado *xml* passou a ser um problema no sentido de performance do sistema.

A interação constante entre a cliente e o servidor nas ações do usuário faziam com que a ferramenta respondesse de forma lenta devido às transformações necessárias entre o formato *xml* e a estrutura adotada. Este processo transformacional é conhecido como *marshalling*[65] e existem *frameworks* específicos para sua realização[66].

Este é um processo considerado lento, ler e escrever um arquivo no formato *xml* exige uma lógica transformacional que já estava sendo considerada crítica mesmo durante o processo de desenvolvimento. Um arquivo no formato *xml* de um texto estava chegando ao tamanho de 8 *megabytes* onerando o processamento da máquina desnecessariamente.

---

[65] http://en.wikipedia.org/wiki/Marshalling_(computer_science)
[66] A implementação utiliza o *Xstream*, porém existem outros.

O processo de serialização da estrutura não precisa, necessariamente, retornar como produto um arquivo em formato *xml*, a serialização pode ocorrer transformando a informação de outra forma, no caso, a forma de *bytes*.

O custo computacional foi assustador, o mesmo documento, neste formato, chegou ao módico tamanho de 235 *kilobytes*, ou seja, quase 35 vezes. A velocidade de acesso à informação (e, consequentemente, das funções aplicadas) aumentou consideravelmente, além da própria linguagem de programação já oferecer recursos nativos para os processos de conversão.

Ao atribuir o termo "abandono do *xml*" não significa não utilizá-lo de forma alguma, mas somente deixar de usá-lo como fonte de recursos. O formato *xml* continua sendo adotado como maneira de transferir a informação, afinal a ferramenta ainda comporta um processo de exportação dos dados para diversos formatos, incluindo o *xml*.

A adoção pela comunidade internacional de um formato estrutural único para transmissão da informação foi um grande passo para integração global dos processos de comunicação, porém devemos compreender os limites de utilização de um ou outro formato sem que ultrapassemos a barreira de sua funcionalidade.

Se pensarmos no formato *xml* como uma espécie de protocolo de comunicação, faz sentido pensarmos que ele é adotado para comunicação entre diferentes interfaces e sistemas, mas que não precisa, necessariamente, ser adotado como

fonte de recursos internos de uma ferramenta. Existem outras possibilidades de formatos, porém o processo de serialização continua.

5. Conclusão

Neste trabalho, apresentamos uma nova proposta para edição de textos que fazem parte de um corpus eletrônico.

Partindo do histórico de desenvolvimento do corpus Tycho Brahe e da ferramenta eDictor, foi possível analisar todo o processo de trabalho de forma que fosse possível chegar a uma forma mais concisa e sem redundâncias de organização da informação através do uso de um único repositório de informações contendo os dados textuais e morfossintáticos do texto.

Para realizar tal tarefa foi criada uma estrutura de dados baseada em unidades mínimas chamadas de *tokens* e blocos de unidades chamados *chunks*. A relação entre os *tokens* e os *chunks,* da forma como considerada neste trabalho, é capaz de guardar a informação de como se estrutura o texto em sua visualização (página, parágrafos, sentenças) e em relação à estrutura sintática no formato de árvores.

A decisão de criação desta estrutura foi realizada após análise dos arquivos *xml* e *psd* que compõem o catálogo de documentos do corpus Tycho Brahe. Através desta análise, foi possível chegar a elementos genéricos que se relacionam desconstruindo o texto criando uma relação de pontos de início e fim relativos às palavras (*tokens*) e não seguindo sua forma linear.

Com a introdução do conceito de orientação a objetos foi possível a criação de uma relação entre unidades ainda menores que o *token*: os *split tokens*. Eles também são *tokens*, pois herdam as características do elemento mais significativo. A relação de posição também foi estendida para o conceito de *joined token*, pois há necessidade de marcar logicamente a junção de *tokens* que espacialmente aparecem separados, seja pelo fato das palavras terem sido escritas separadamente, seja pelo necessidade de criar esta relação lógica. Todos estes

Os elementos que compõem a estrutura possuem atributos. O intuito neste trabalho foi utilizar o mínimo possível de atributos, sem se criar atributos específicos demais ou genéricos de menos. Na busca deste equilíbrio, foi verificada a necessidade de se criar um atributo específico para a unidade sintática, um atributo de nível que indica a distância do nó para o nó raiz. Chamamos este atributo de terceira dimensão do *chunk*. Para a estrutura textual, este valor possui um nulo *default*.

Organizada a informação, o acesso a ela se torna muito mais simples e parte-se para definição da interface do usuário. A tecnologia web disponível permite que elementos sejam posicionados na tela reproduzindo a formatação que ocorre no livro e que também permite que haja uma independência entre um e outro elemento. Esta independência é o que permite que a informação trafegue entre o computador do usuário e a central de processamento na nuvem sem que o usuário perceba. O processamento ocorre em *background*, ele ocorre utilizando tecnologias assíncronas.

A semelhança entre as tecnologias *html* e *xml* introduziu uma necessidade de adaptação da informação para apresentação ao usuário. A solução apresentada neste trabalho é pensada de forma a atribuir aos *tokens* informações que indiquem que eles fazem parte de um *chunk*. Assim, não são as palavras que pertencem a uma sentença, mas cada palavra possui um pedaço de informação que a faz pertencer à sentença. Esta forma de pensar muda a forma como a informação é exibida.

Os testes realizados durante o desenvolvimento do sistema também mostraram a inviabilidade de se operar na nuvem com arquivos *xml*. Eles são muito grandes e transformar um arquivo *xml* para sua visualização normal como texto na interface do usuário torna o processamento da máquina muito oneroso não sendo possível atingir o mínimo necessário de tempo de resposta.

Por fim, adotamos a utilização de estruturas serializadas em *bytes* abandonando o formato *xml* como fonte de recursos, ainda que ele continuasse sendo utilizado como forma de *backup*.

Do ponto de vista lógico computacional, não existe diferença entre o texto no formato de livro e o texto na estrutura lógica apresentada neste trabalho. O que ocorre é um processo de conversão em ambos os casos: no sentido do texto para estrutura e da estrutura para texto. Uma grande questão que se levanta neste momento é: qual seria a perda relativa a esse processo de conversão? Ou seria este um processo completamente isomórfico que não abriria espaço a perdas significativas da informação? O objetivo computacional é o de diminuir ao máximo estas perdas, pois, mesmo que não seja relevante para o processo de

construção de um corpus (em relação aos requisitos iniciais para sua construção), a modelagem deve prever seus resultados, ou seja, o nível ótimo que pode ser atingido.

A transcrição de um texto levanta vários problemas, exemplos deles são as notas às margens, as informações de rodapé e de cabeçalho e até mesmo imagens representando letras iniciais de sentenças. Aqui cabem regras para determinar como estas informações devem ser posicionadas em relação à lógica sequencial da informação contida no texto. Uma nota localizada à margem direita de um texto receberia um número sequencial em relação ao texto que a descreve ou seria ela um metadado do texto? Ou um *chunk*?

Ainda assim a conversão da estrutura para o formato linear do texto persiste e pode ser entendida como um processo de desfragmentação.

Ao se trabalhar com processos computacionais, os modelos são desenvolvidos em cima de uma base empírica, neste caso, todos os textos que compõem o projeto Tycho Brahe. A manipulação dos elementos que compõem o modelo é sempre restrita a uma tecnologia, seja ela a adotada, seja ela a existente (ou a possível). A técnica empregada restringe as possibilidades e exige que determinadas regras sejam seguidas para que ela possa ser utilizada. Independentemente disto, à análise da situação cabem outras perspectivas, outros âmbitos de compreensão do objeto analisado para que se possa chegar a um modelo computacional, isto é, cabe analisar os fatores que identificam o objeto a ser modelado dentro daquilo que é possível realizar.

Uma das formas mais tradicionais de compreensão da informação é pensar que um objeto possui atributos e os atributos definem o objeto cabendo àquele que o utiliza manipular seus atributos, valorando-os conforme a instância em que este objeto se apresenta. Exemplificando, o objeto livro possui seus atributos que o identificam como livro[67], cria-se um objeto *livro* no sistema que é identificado por meio de seus atributos (sejam quais eles forem). No sistema computacional, ocorre um processo de instanciação do objeto *livro*, ele se torna uma entidade cujos atributos passam a ser valorados com as informações que o usuário do sistema atribui a ele. Em uma visão computacional web, isto é realizado através de uma interface composta de campos de entrada que são preenchidos pelo usuário, uma espécie de cadastro de informações[68].

Em um cadastro, umas das operações possíveis é a de edição dos dados onde se realiza uma transposição dos atributos de um objeto para uma interface compreensível àquele que a utiliza, ou seja, passa-se a uma ordenação das informações de tal forma que ela seja compreensível a um usuário.

As interfaces são responsáveis pela realização deste processo: tornar informações compreensíveis. Estas interfaces se dão em diversos níveis de acordo com o grau de conhecimento do usuário em relação ao objeto especificado, afinal, para que a informação possa ser definida em uma interface amigável a um usuário do eDictor, outro alguém (o programador) utilizou

---

[67] Sem levantar maiores questões filosóficas, irei tratar isto da maneira mais simples possível dentro da perspectiva programacional da orientação a objetos.

[68] Cadastros são a forma mais comum de *input* de dados sobre um objeto pelo usuário do sistema.

alguma outra interface possibilitando a manipulação da informação para que ela seja disponibilizada para outros.

Isto significa dizer que a escolha de uma ou outra interface depende do nível de conhecimento do usuário que a utiliza, ou seja, seria perfeitamente possível que qualquer um pudesse manipular a informação do texto apresentada na estrutura adotada, desde que ele possuísse a compreensão da estrutura. De certa maneira, esta análise mostra que os dois formatos (um texto *in natura* e a estrutura de dados) podem ser compreendidos como instâncias do mesmo objeto, pois tanto faz ler uma obra no livro ou no *tablet*: a obra permanece.

Caso eu pegue uma edição das obras completas de William Shakespeare traduzidas em japonês e entregue esta versão impressa a uma pessoa qualquer que não sabe ler japonês, esta pessoa poderia, no máximo, me dizer algo como: eu não sei ler japonês. Ou ainda me perguntar em que língua aquele texto está escrito.

Caso eu pegue uma edição eletrônica das obras completas de William Shakespeare que utiliza a estrutura descrita neste projeto e a imprima no formato de *bytes* e entregue para qualquer pessoa ler, no máximo, eu obteria uma resposta próxima a: "O que significa isto? Para mim são, no máximo, códigos de computador. Eu precisaria de uma máquina para entendê-los."

Porém não seriam ambas edições das obras completas de William Shakespeare?

Um falante de japonês e uma pessoa com as ferramentas adequadas às mãos não poderiam se deleitar imaginando seus próprios sonhos de uma noite de

verão? Seria diferente se o caso fosse relacionado às amarguras no sertão de João Cabral de Melo Neto?

Ao meu ver, a obra, em si, não é afetada pelas reações sinestésicas na relação de uso de um objeto sólido como o livro ou pelas frias e distantes relações estruturais impostas pelos exercícios computacionais. A obra se mantém. A arte parece ser imanente.

Porém como todo sistema, este também possui suas fragilidades. E elas estão expostas esperando o mínimo abalar de uma peça da estrutura para que todo o significado se derrube.

Imaginemos a seguinte situação hipotética surrealista.

A estrutura apresentada neste trabalho é adotada mundialmente como solução técnica para textos virtuais e um grande esforço político é realizado para que passemos todos os nossos textos para este formato, pois parece que estamos nos aproximando de uma hecatombe mundial e temos que salvar nossos maiores bens: nossos livros, nossa história.

Precisaremos de uma massa de trabalhadores para realizar tal tarefa dantesca: programadores e digitadores em ação. Se seguirmos a lógica de organização hierárquica iniciada no começo do século XX (linhas de produção fordistas), teríamos uma organização do trabalho em forma de pirâmide, seja com a base inchada pelos programadores e digitadores ou por um miolo inflado por pesquisadores e intelectuais, ou ainda por um topo (sempre pequeno) de administradores, gerentes, patrões, o que for.

Se fatiarmos qualquer um deste níveis, poderemos entendê-lo como horizontal, ou seja, não haveria relação hierárquica entre os indivíduos que compõem uma ou outra fatia.

Em qual destas fatias a estrutura se encontraria? Se pensarmos que a estrutura é um objeto da programação, ela se encontraria na parte mais inferior da pirâmide.

O problema é que a estrutura é muito frágil. Qualquer alteração nela traz impacto na compreensão do todo. Se um atributo for modificado, toda informação é perdida. Agora, imagine o impacto na organização da informação, se um programador desavisado em meio àquelas centenas de cabeças posicionadas entre divisórias, uma após a outra, indefinidamente até onde a vista alcance, simplesmente alterasse a estrutura.

Já estaríamos nós, agora, correndo atrás dos *backups* e tentando o salvar o máximo de informação que fosse possível.

Esta situação é mais hipotética do que surreal. Ela seria surreal se a política entendesse a tecnologia de forma mais séria como na historinha acima, mas não tão hipotética porque é assim que se reproduz a organização do trabalho de tecnologia da informação na primeira década do século XXI.

Esta conclusão serve como uma espécie de aviso sobre os pontos vulneráveis deste projeto. A adoção de uma estrutura serializada como atributo de uma entidade em um banco de dados relacional foi tomada somente a partir das

análises das informações do projeto Tycho Brahe. Existe um risco muito sério na adoção de uma estrutura deste tipo.

Tecnicamente adoto um procedimento de *backup* constante dos documentos do projeto Tycho Brahe em *xml*, mas ele é realizado somente como ação preventiva.

Outra questão importante é a arbitrariedade da unidade numérica de valor 1 como base para cálculo do nível do *chunk* sintático.

O algoritmo criado para reformulação da estrutura visual de árvore a partir dos *chunks* foi o algoritmo mais complexo desenvolvido neste trabalho e ele será ainda frutos de outras pesquisas futuras (vide apêndice C). Porém a arbitrariedade para este valor numérico é muito maior do que a nomenclatura dada para qualquer elemento da estrutura ou para a própria estrutura em si. Sâo apenas nomes e nada mais. A adoção da nomenclatura em inglês acontece muito mais pela proximidade que existe entre o inglês e o Java, nada mais.

Determinar um valor numérico arbitrário é ultrapassar a barreira da determinação de um valor *default* qualquer para um objeto. Em todo este trabalho, cito os valores *default* como nulos. O estado de nulo na programação orientada a objetos não é nem verdadeiro, nem falso, ele é um nulo, um terceiro estado. Determinar algo como nulo é completamente diferente de determinar algo como 1. O algoritmo supra citado dá pistas na argumentação contra esta arbitrariedade.

Por fim, os avanços tecnológicos obtidos durante o desenvolvimento do eDictor e a integração com o projeto Tycho Brahe atingiram níveis satisfatórios no meu

ponto de vista. A arquitetura adotada e as tecnologias disponíveis foram determinantes para que as funcionalidades fossem desenvolvidas de forma que o trabalho do usuário possa ser cada vez mais eficiente sem que a curva de aprendizado já adquirida pelo uso da versão plataforma do eDictor seja perdida.

Agora, se um dia o mundo hipotético e surreal vir a acontecer, um simples programador sem intenção de nenhum mal, somente pela ignorância da existência de uma estrutura, viesse a encontrar esta estrutura e a modificasse na ingenuidade humana de querer o bem, centenas de milhões de obras seriam perdidas, a perda da história seria irreparável como em mitos babilônicos que ecoam há eras. Restaria a nós escavar e restaurar.

Em uma era de desorganização da informação, tanto faz se *bytes* ou papiros.

6. Referências bibliográficas

Bikel, D. *Intricacies of Collins' Parsing Mode*. 2003.

Galves, C. e Faria, P. *Tycho Brahe Parsed Corpus of Historical Portuguese*. URL: http://www.tycho.iel.unicamp.br/~tycho/corpus/en/index.html. 2010

Kepler, F. *Um etiquetador morfo-sintático baseado em cadeias de Markov de tamanho variável*. Dissertação de mestrado. 2005.

Paixão de Sousa, M. C. e Trippel, T. *Building a historical corpus for Classical Portuguese: some technological aspects*. 2006.

Paixão de Sousa, M. C. *Memórias do Texto*. In: Revista Texto Digital, ano 2, no. 1. 2006

Paixão de Sousa, M. C. *Digital Text : Conceptual and Methodological Frontiers*. 2007.

Paixão de Sousa, M. C. *O Corpus Tycho Brahe: contribuições para as humanidades digitais no Brasil*. 2014. http://revistas.usp.br/flp/article/view/88404

Paixão de Sousa, M. C. et al. *An integrated tool for annotating historical corpora*. 2010.

Paixão de Sousa, M. C. et al. *E-Dictor: novas perspectivas na codificação e edição de corpora de textos históricos*. 2010.

Trippel, T. e Paixão de Sousa, M. C. *Single source processing of historic corpora*. 2004.

Turing, A. M. *On computable numbers, with an application to the Entscheidungsproblem*. 1936.

Unsworth, John. *Digital Humanities Beyond Representation*. 2006.

Unsworth, John. *Knowledge Representation in Humanities Computing*. 2014.

7. Apêndices

7.1. APÊNDICE A: o processo operacional do Corpus Tycho Brahe

Finalizado o processo de edição e etiquetagem, o usuário possui dois arquivos: um arquivo *xml* contendo as informações textuais e morfológicas e um arquivo *pos* com as sentenças etiquetadas morfologicamente. Para completar o fluxo para análise sintática, o usuário envia este segundo arquivo por e-mail para o operador técnico que executa algumas operações para preparar o arquivo para seu processamento no *parser* sintático.

Estas operações consistem na execução de regras de identificação de padrões nos caracteres do arquivo e processos de inclusão, eliminação e substituição destes e outros caracteres. Dentro destas operações, a mais importante é o *splitter*[69] que consiste na identificação de etiquetas compostas, isto é, palavras identificadas pelo etiquetador como sendo composições de outras etiquetas, por exemplo, a palavra *da* é identificada pela etiqueta *P+D-F* (preposição mais determinante feminino), portanto cabe ao *splitter* quebrar a palavra em "subpalavras" atribuindo a cada uma delas a etiqueta correspondente[70]. Para o exemplo acima, a palavra de par *da/P+D-F* é quebrada em duas: *de/P* e *a/D-F*. A regra que determina esta quebra é previamente determinada utilizando-se expressões regulares, ou seja, o *splitter* executa uma série de expressões regulares para quebrar um composto em partículas menores de mesma

[69] Kepler, 2005.
[70] Observar que este processo de quebra também é uma *tokenização*, porém pelo caractere +.

natureza, pois ambos continuam sendo compreendidos pela ferramenta como palavras[71].

Todos os outros processos consistem em *scripts* programados para preparação dos dados do arquivo para o formato esperado pelo *parser* e correção de marcas de identificação.

O *parser* também consiste em uma função que recebe um arquivo em formato texto e retorna outro arquivo em formato texto contendo as estruturas das árvores sintaticamente anotadas. Este arquivo é conhecido pela sigla *psd* e é o produto final de todo processo para preparação dos textos sintaticamente anotados.

Após o término do processamento computacional, os textos são revisados para que possam constituir uma base de busca de informações para pesquisadores. Todas as outras ferramentas[72] operam utilizando como fonte o arquivo *psd*. Os números de identificação das sentenças nestes arquivos são aqueles utilizados como referência para as sentenças, sendo únicos por sentença.

O operador técnico pega este arquivo e envia por email para o usuário que solicitou o processamento e o fluxo semiautomatizado do projeto Tycho Brahe é encerrado.

---

[71] Este é o processo atual do projeto Tycho Brahe, porém não seria possível que o *parser* pudesse trabalhar com etiquetas compostas?
[72] Ferramentas de busca e edição em árvores sintáticas.

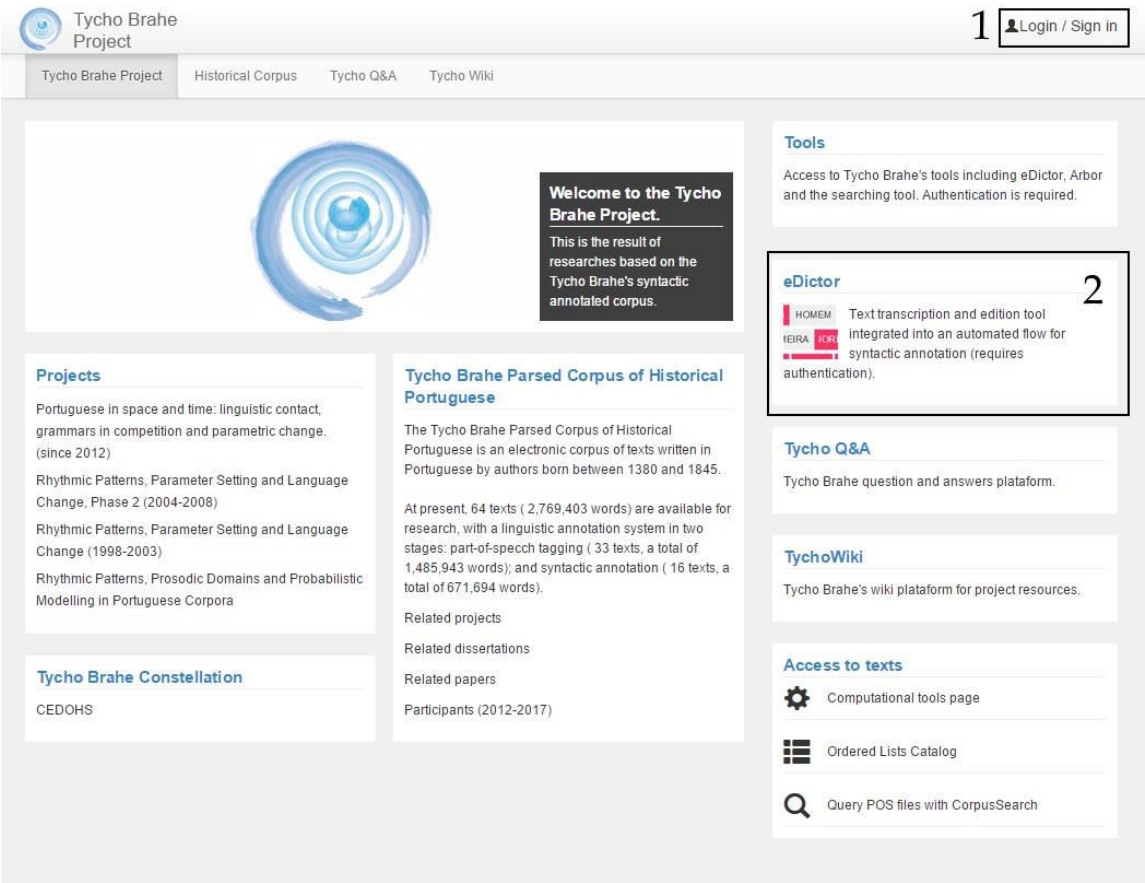
7.2. APÊNDICE B: manual de utilização do eDictor

Este apêndice descreve as funcionalidades de cada tela do eDictor. Os quadrados numerados correspondem às funcionalidades descritas em tabela subsequente à imagem. A tabela contém três colunas, a primeira contém a indicação do número correspondente na imagem seguido pela descrição daquela funcionalidade e, por fim, o evento correspondente à ação. Um evento, por exemplo, seria o direcionamento do usuário para outra tela quando ele realiza a ação de clique no elemento da tela.

7.2.1. Portal Tycho Brahe

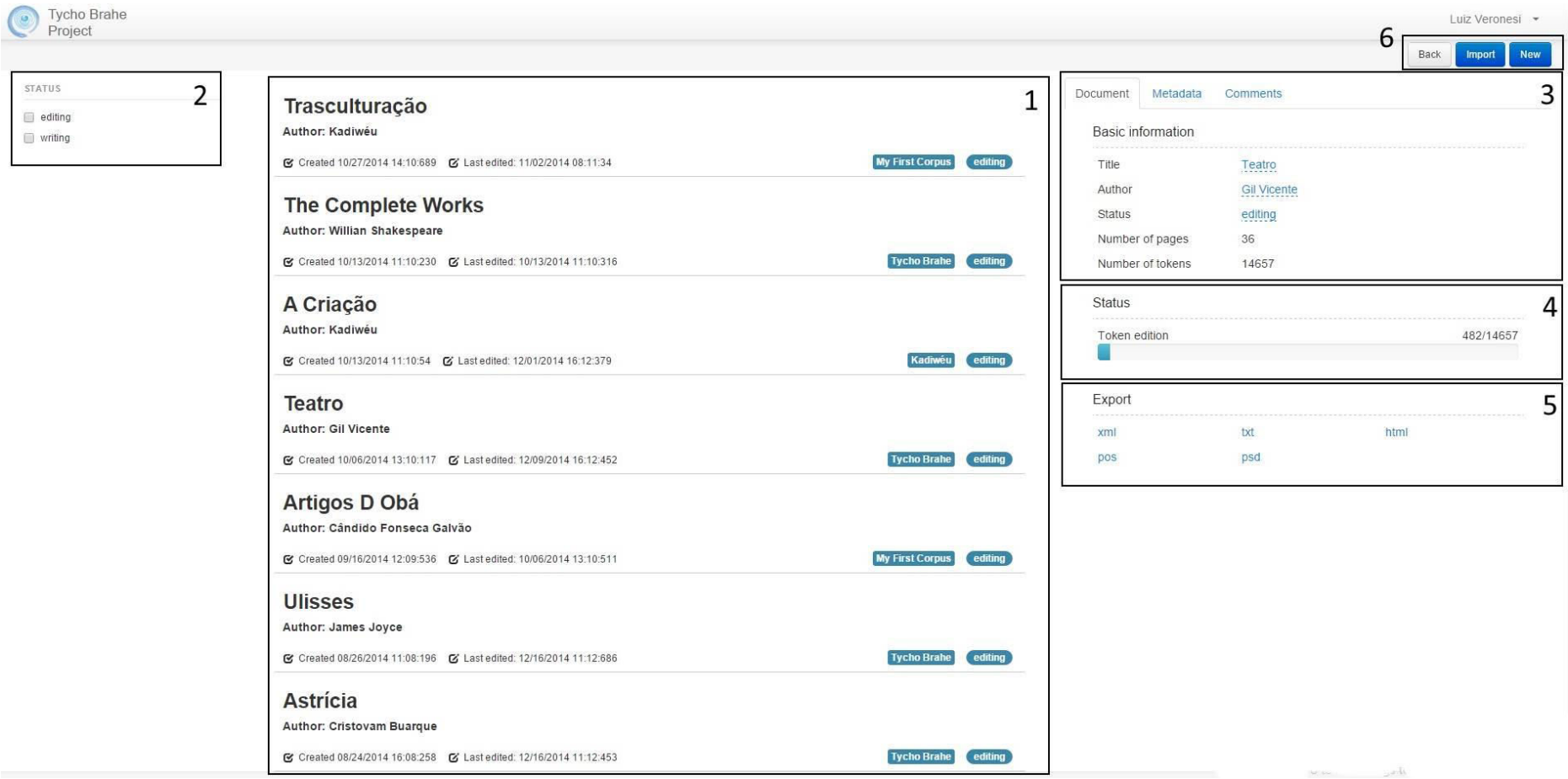
Tela principal do corpus Tycho Brahe que dá acesso ao eDictor.

	descrição	*evento*
1	Área para login ou cadastro no portal do corpus Tycho Brahe	Exibe a área de login ou cadastro para o usuário no clique.
2	Acesso ao eDictor	Caso o usuário esteja logado no portal, ele é direcionado para a tela em 7.2.2



7.2.2. Tela inicial - lista de documentos

Esta tela contém a lista de documentos aos quais o usuário tem acesso. Esta é também a tela inicial do sistema após o usuário ter realizado seu login no portal do Corpus Tycho Brahe.

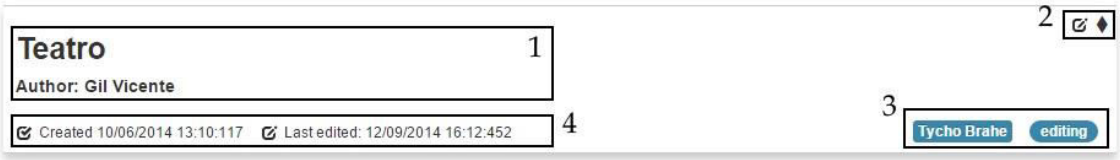

Tycho Brahe Project
Luiz Veronesi
6
Back
Import
New
STATUS
2
editing
writing
1
Trasculturação
Author: Kadiwéu
Created 10/27/2014 14:10:689
Last edited: 11/02/2014 08:11:34
My First Corpus
editing
The Complete Works
Author: Willian Shakespeare
Created 10/13/2014 11:10:230
Last edited: 10/13/2014 11:10:316
Tycho Brahe
editing
A Criação
Author: Kadiwéu
Created 10/13/2014 11:10:54
Last edited: 12/01/2014 16:12:379
Kadiwéu
editing
Teatro
Author: Gil Vicente
Created 10/06/2014 13:10:117
Last edited: 12/09/2014 16:12:452
Tycho Brahe
editing
Artigos D Obá
Author: Cândido Fonseca Galvão
Created 09/16/2014 12:09:536
Last edited: 10/06/2014 13:10:511
My First Corpus
editing
Ulisses
Author: James Joyce
Created 08/26/2014 11:08:196
Last edited: 12/16/2014 11:12:686
Tycho Brahe
editing
Astrícia
Author: Cristovam Buarque
Created 08/24/2014 16:08:258
Last edited: 12/16/2014 11:12:453
Tycho Brahe
editing
Document
Metadata
Comments
3
Basic information
Title
Teatro
Author
Gil Vicente
Status
editing
Number of pages
36
Number of tokens
14657
Status
4
Token edition
482/14657
Export
5
xml
txt
html
pos
psd

	descrição	*evento*
1	Lista de documentos aos quais o usuário possui acesso.	Ao clicar no documento são exibidas as informações descritas em 3, 4 e 5. Ao passar o mouse sobre o documento são exibidas as funcionalidades detalhadas no item 2 de 7.2.3.
2	Lista de status disponíveis para filtragem de documentos.	Ao selecionar um ou mais status, a lista de documentos é recarregada exibindo somente os documentos válidos quanto ao critério de filtragem.
3	Informação básica do documento e abas para metadados e lista de comentários.	Ao clicar na aba de metadados são exibidas as informações descritas em 7.2.9. Ao clicar na aba de metadados são exibidas as informações descritas em 7.2.11. Os campos sublinhados são editáveis, i.e., o dado de informação do documento pode ser alterado simplesmente clicando-o.
4	Área que informa estatísticas do documento como, por exemplo, a quantidade de *tokens* editados.	
5	Lista os formatos para os quais o documento pode ser exportado.	Ao clicar no formato, o usuário realiza o download do documento. Formatos disponíveis: *xml, txt, html, pos, psd*.
6	Área onde são listadas as funções disponíveis de ações para o usuário.	Clicando-se no botão de importar, o sistema exibe área para upload do arquivo que se pretende importar (formatos disponíveis: *psd, pos, xml, xml* do eDictor plataforma). Clicando-se no botão de novo, um novo documento é inserido à lista do usuário.

7.2.3. Detalhe do documento

Este item descreve as funcionalidades disponíveis para operação sobre o documento e os detalhes principais.

	descrição	*evento*
1	Título do documento e nome do autor.	
2	Funcionalidades disponíveis para o documento. Neste caso, transcrição (primeiro ícone) e edição (segundo ícone).	Ao clicar em quaisquer uma das funcionalidades, o usuário será direcionado para o módulo de edição correspondente.



7.2.4. Transcrição do documento

Na transcrição do documento, o usuário digita ou cola o texto de cada página em que ele está trabalhando. É possível realizar o upload das imagens de cada página e associá-las exatamente à página na qual o usuário está trabalhando.

A ferramenta de visualização de imagens permite que o usuário possa visualizar a imagem e digite o texto sem a necessidade de utilizar programas auxiliares para visualização das imagens. Esta ferramenta possui uma funcionalidade de zoom útil para visualização de elementos que estejam mais difíceis de se ler.

Obs.: é importante salvar constantemente o trabalho para atualização no banco de dados central.

	descrição	*evento*
1	Área de digitação.	
2	Abas de funcionalidade: upload da imagem da página e mapa de caracteres.	Realiza o upload de uma imagem ao arrastá-la para a área de upload (*drag*) ou clicando na área. O mapa de caracteres está descrito em 7.2.12
3	Dados básicos do documento.	Ao clicar em quaisquer dos campos editáveis, é possível realizar modificação da informação.
4	Botões de funcionalidades: salvar e voltar.	Ao clicar no botão, a ação correspondente é executada.
5	Botões de paginação. Eles são utilizados em todo o sistema para navegação pelas páginas dos documentos.	Ao clicar no botão, a ação correspondente é executada.

Tycho Brahe Project

Luiz Veronesi

Document: Ulisses **Author:** James Joyce **Status:** editing 3

Back Save 4

Stately, plump Buck Mulligan came from the stairhead, bearing a bowl of lather on which a mirror and a razor lay crossed. A yellow dressinggown, ungirdled, was sustained gently behind him on the mild morning air. He held the bowl aloft and intoned: 1

—Introibo ad altare Dei.

Halted, he peered down the dark winding stairs and called out coarsely:

—Come up, Kinch! Come up, you fearful jesuit!

Solemnly he came forward and mounted the round gunrest. He faced about and blessed gravely thrice the tower, the surrounding land and the awaking mountains. Then, catching sight of Stephen Dedalus, he bent towards him and made rapid crosses in the air, gurgling in his throat and shaking his head. Stephen Dedalus, displeased and sleepy, leaned his arms on the top of the staircase and looked coldly at the shaking gurgling face that blessed him, equine in its length, and at the light untonsured hair, grained and hued like pale oak.

Buck Mulligan peeped an instant under the mirror and then covered the bowl smartly.

—Back to barracks! he said sternly.

He added in a preacher's tone:

—For this, O dearly beloved, is the genuine Christine: body and soul and blood and ouns. Slow music, please. Shut your eyes, gents. One moment. A little trouble about those white corpuscles. Silence, all.

He peered sideways up and gave a long slow whistle of call, then paused awhile in rapt attention, his even white teeth glistening here and there with gold points. Chrysostomos. Two strong shrill whistles answered through the calm.

—Thanks, old chap, he cried briskly. That will do nicely. Switch off the current, will you?

He skipped off the gunrest and looked gravely at his watcher, gathering about his legs the loose folds of his gown. The plump shadowed face and sullen oval jowl recalled a prelate, patron of arts in the middle ages. A pleasant smile broke quietly over his lips.

—The mockery of it! he said gaily. Your absurd name, an ancient Greek!

He pointed his finger in friendly jest and went over to the parapet, laughing to himself. Stephen Dedalus stepped up, followed him wearily halfway and sat down on the edge of the gunrest, watching him still as he propped his mirror on the parapet, dipped the brush in the bowl and lathered cheeks and neck.

Buck Mulligan's gay voice went on.

—My name is absurd too: Malachi Mulligan, two dactyls. But it has a Hellenic ring, hasn't it? Tripping and sunny like the buck himself. We must go to Athens. Will you come if I can get the aunt to fork out twenty quid?

He laid the brush aside and, laughing with delight, cried:

—Will he come? The jejune jesuit!

Ceasing, he began to shave with care.

—Tell me, Mulligan, Stephen said quietly.

—Yes, my love?

—How long is Haines going to stay in this tower?

Buck Mulligan showed a shaven cheek over his right shoulder.

Image Charmap 2

Drop files *to upload*

(or click)

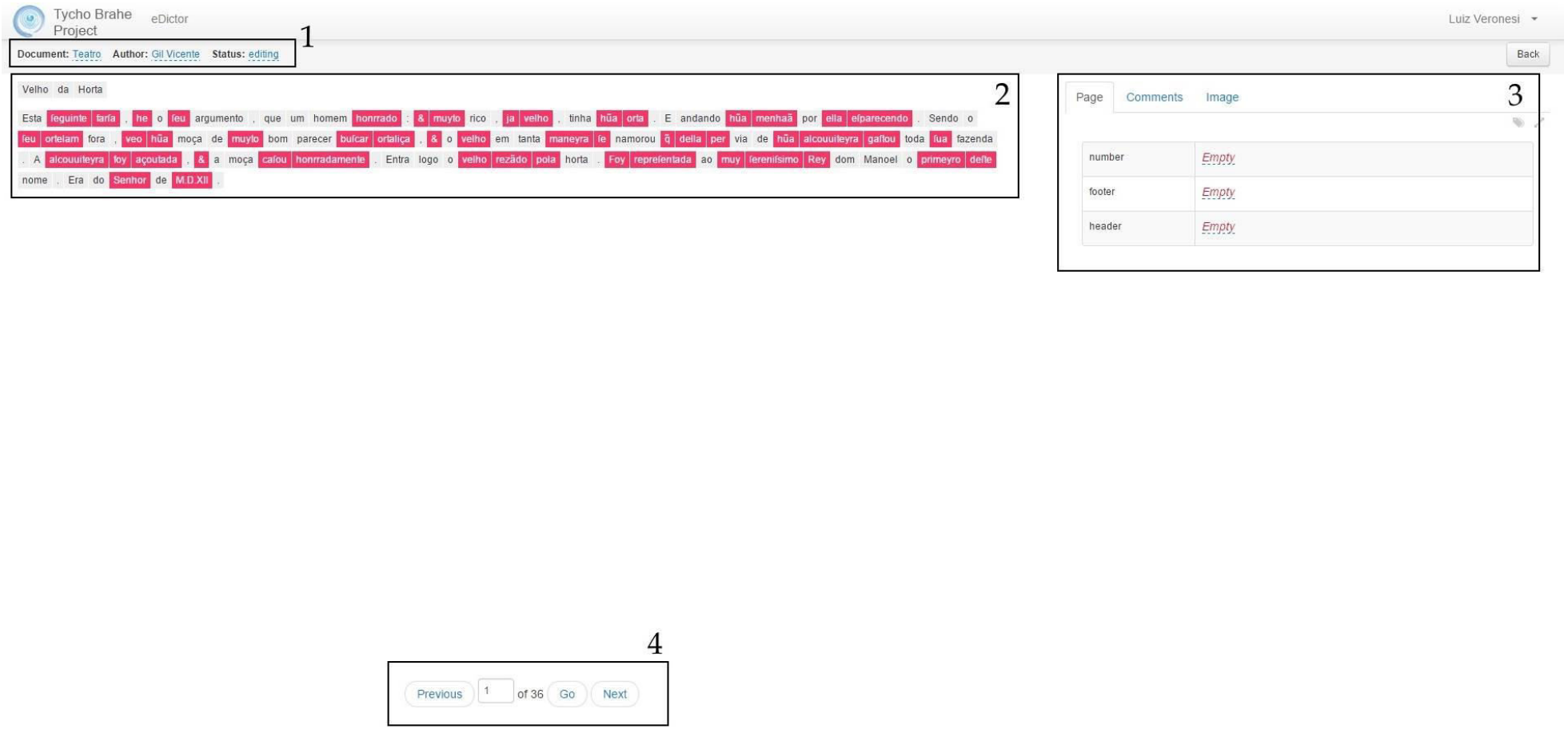
Previous 1 of 1 Go Next 5

7.2.5. Edição do documento

Após a realização do processo de transcrição do documento, o usuário pode passar a trabalhar com a edição do documento.

Neste momento, o texto do documento passa pelo processo de *tokenização* e é transformado na estrutura proposta neste trabalho.

	descrição	*evento*
1	Dados básicos do documento.	Ao clicar em quaisquer dos campos editáveis, é possível realizar modificação da informação.
2	Área de edição.	Ao clicar no *token* é exibida a área de edição descrita em 7.2.6.
3	Abas de funcionalidade: informações básicas da página, comentários e imagem.	Ao clicar nos elementos editáveis, eles podem ser modificados. A aba de imagem está descrita em 7.2.4. A aba de comentários está descrita em 7.2.11.
5	Botões de paginação.	Ao clicar no botão, a ação correspondente é executada.


Tycho Brahe Project
eDictor
Luiz Veronesi
1
Document: Teatro Author: Gil Vicente Status: editing
Back
2
Velho da Horta
Esta ſeguinte ſaría , he o ſeu argumento , que um homem honrrado : & muylo rico , ja velho , tinha hũa orta . E andando hũa menhaã por ella eſparecendo . Sendo o ſeu ortelam fora , veo hũa moça de muylo bom parecer buſcar ortaliça , & o velho em tanta maneyra ſe namorou q̃ della per via de hũa alcouuiteyra gaſtou toda ſua fazenda . A alcouuiteyra foy açoutada , & a moça caſou honrradamente . Entra logo o velho rezãdo pola horta . Foy repreſentada ao muy ſereniſsimo Rey dom Manoel o primeyro deſte nome . Era do Senhor de M.D.XII .
3
Page
Comments
Image
number
Empty
footer
Empty
header
Empty
4
Previous
1
of 36
Go
Next

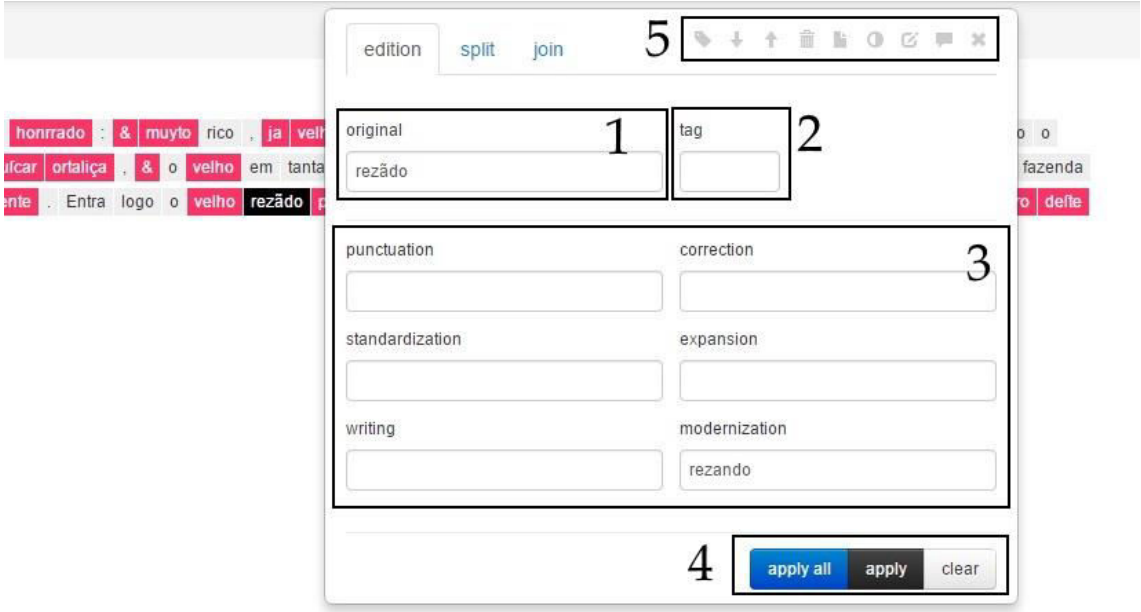
7.2.6. Edição - detalhe *token*

Tela de edição de um token contendo as funcionalidades de operação e os valores que podem ser atribuídos de acordo com os níveis de edição pré-configurados parta o corpus.

	descrição	*evento*
1	Valor original da palavra.	
2	Valor da tag morfológica atribuída pelo etiquetador.	
3	Valores dos níveis de edição.	
4	Funcionalidades de edição do *token*.	

As funcionalidades criadas para edição do *token* na página em que está são as seguintes (descrevendo da esquerda para a direita):

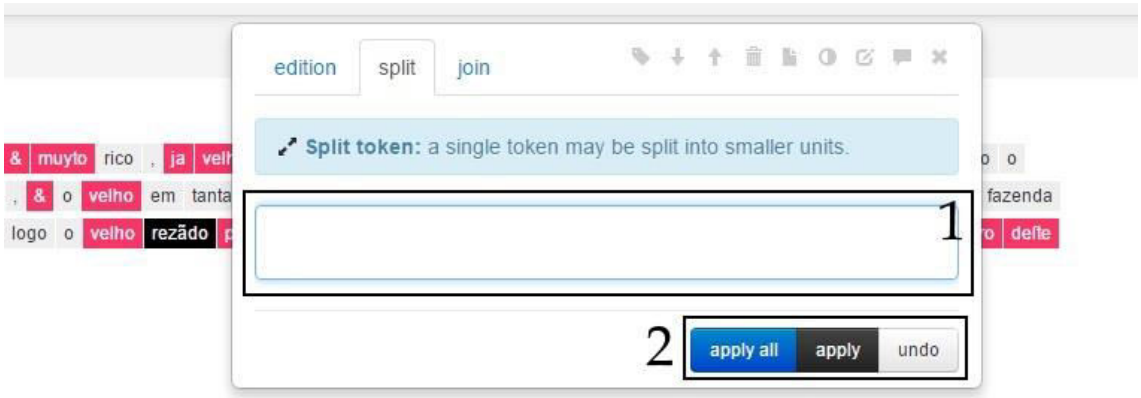
- Indicador de utilização pelo etiquetador: caso marcado (valor *default*) o token é usado no processo de etiquetagem, caso contrário ele é ignorado.
- Insere quebra de linha: "joga" o *token* selecionado para a linha de baixo.
- Remove quebra de linha: "joga" o *token* para cima removendo a quebra de linha, caso haja quebra de linha antes deste *token*.
- Remove o *token*: remove o token caso a palavra tenha sido digitada por engano, por exemplo.
- Insere nova página: insere uma página em branco deste *token* em diante, ou seja, o sistema cria uma nova página contendo os *tokens* localizados entre o *token* selecionado e o último *token* da página.
- Junta duas páginas: é realizado um *merge* entre a página atual e a próxima página do documento suprimindo uma página do documento.
- Insere texto: permite a inclusão de um bloco de texto após o *token* selecionado.
- Insere comentário: inclui um comentário relacionado a um *token* ou a um conjunto de tokens.
- Fechar janela: fecha a tela de edição de token.



7.2.7. Edição - detalhe *split*

Esta tela representa a segunda aba de funcionalidade da edição do *token*. O usuário deve informar as palavras (separadas por espaço em branco) que correspondem à divisão do *token* em unidades ainda menores.

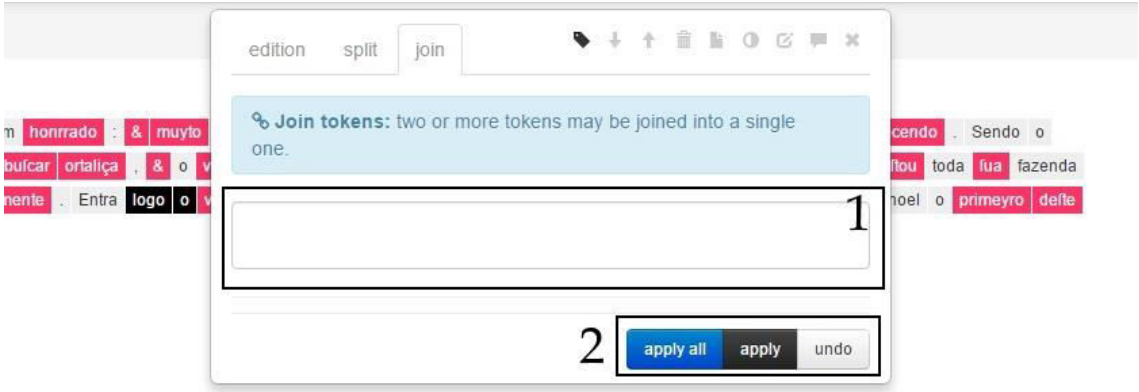
	descrição	*evento*
1	Campo de entrada para as palavras.	
2	Botões de operação: aplicar para todos *tokens*, aplicar somente para este *token* ou desfazer.	Ao clicar no botão, a ação correspondente é executada.



7.2.8. Edição - detalhe *join*

Esta tela representa a terceira aba de funcionalidade da edição do *token*. O usuário deve informar a palavra que pode substituir os valores dos *tokens* selecionados para junção.

	descrição	*evento*
1	Campo de entrada para a palavra. Este campo é opcional.	
2	Botões de operação: aplicar para todos *tokens*, aplicar somente para este *token* ou desfazer.	Ao clicar no botão, a ação correspondente é executada.



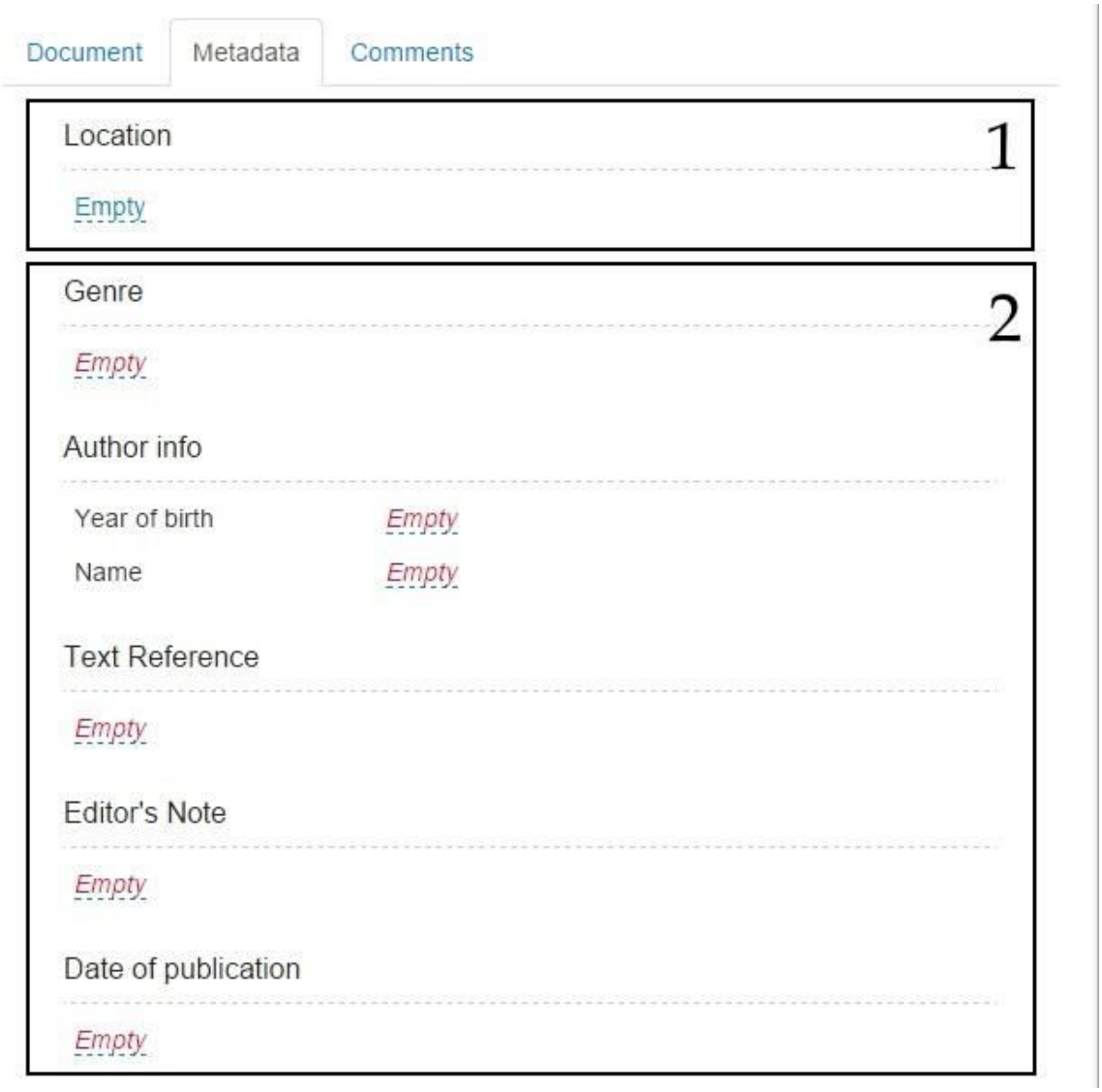
7.2.9. Metadados do documento

Esta tela representa a aba de metadados exibida na lista de documentos (7.2.2). Os metadados podem ser configurados por corpus e podem ser de diversos tipos:

- Referência geolocalizada
- Lista de valores
- Ano
- Data

- Texto livre curto
- Texto livre longo

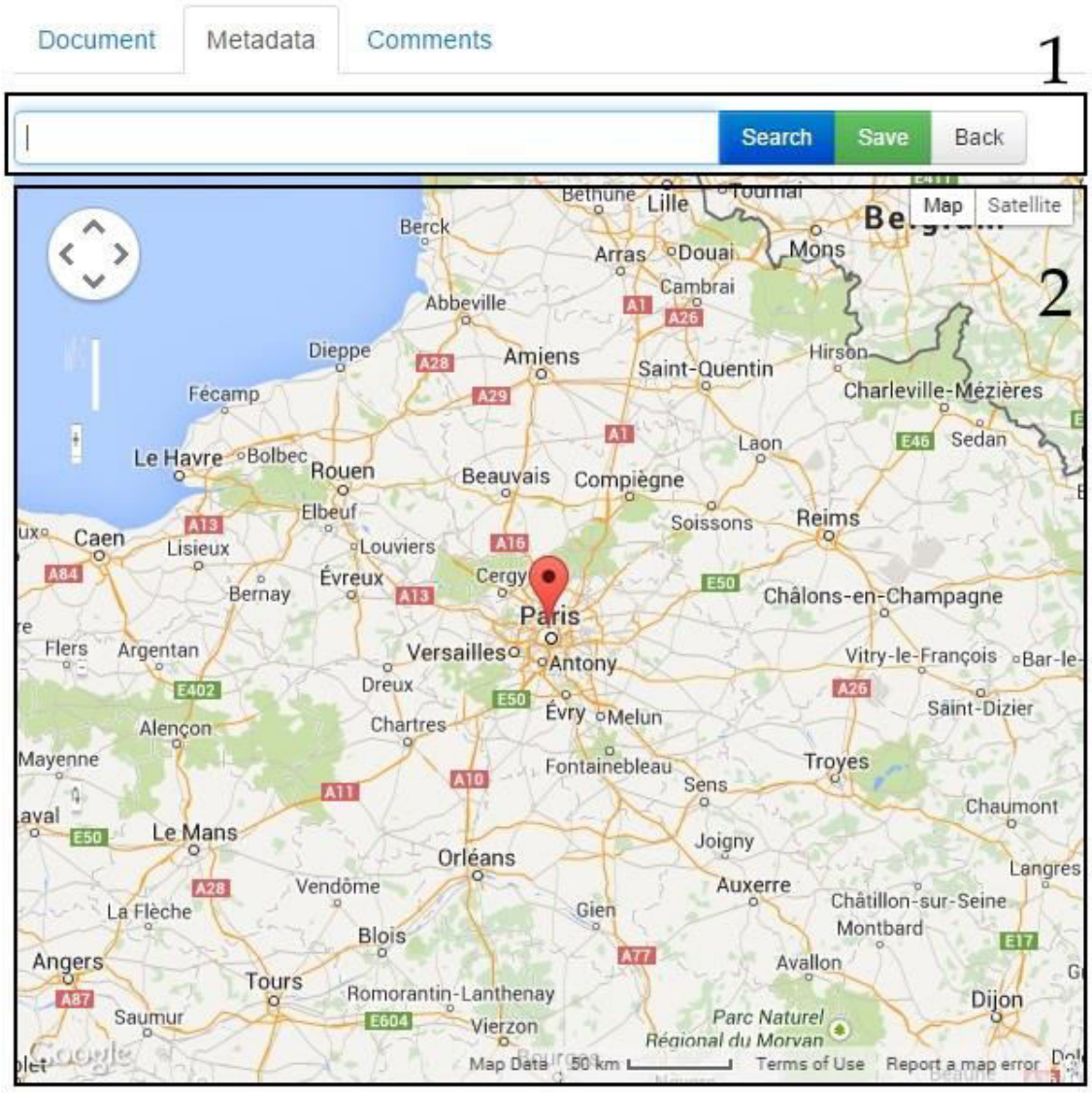
	descrição	*evento*
1	Campo de entrada para valor geolocalizado.	Ao clicar no campo editável, o usuário é direcionado para 7.2.10.
2	Outros campos de metadados.	Ao clicar nos elementos editáveis, eles podem ser modificados.



7.2.10. Metadados geolocalizados

Tela para registro da informação de metadado geolocalizado. O usuário pode arrastar o marcador no mapa para a posição desejada ou realizar uma busca pelo endereço.

	descrição	*evento*
1	Funções de busca de endereço.	O botão de busca realiza a pesquisa pelo endereço informado. O botão de salvar atribuir ao documento o valor de latitude e longitude do local selecionado. O botão de voltar leva o usuário para a tela descrita em 7.2.9.
2	Funcionalidades de mapa.	Usuário pode navegar pela mapa, utilizar o zoom e arrastar o marcador para selecionar a posição desejada.



7.2.11. Comentários

Tela que lista os comentários possibilitando a edição e remoção. Um comentário pode estar associado a um *token*, um conjunto de *tokens* ou ao documento como um todo.

	descrição	*evento*
1	Texto digitado como comentário.	Ao clicar nos elementos editáveis, eles podem ser modificados.
2	Dados do usuário e data de inclusão.	
3	Botão de exclusão.	
4	Campo de entrada para novo comentários e botão de inclusão.	

7.2.12. Mapa de caracteres

Mapa de caracteres que possibilita que o usuário insira no texto os caracteres que não são comuns ao teclado que utiliza. Ao clicar no símbolo desejado, ele é inserido no texto que está sendo transrito

.

7.3. APÊNDICE C: código-fonte do algoritmo de reversão dos chunks

Neste anexo incluo o código-fonte, em Java, do algoritmo que realiza a reversão dos chunks sintáticos para a estrutura de árvore apresentada na interface. Este algoritmo está implementado na classe *SentenceWrapperHelper*.

```
package tycho.edictor.tree.helper;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

import tycho.common.map.HashMapList;
import tycho.common.token.TychoTokenizer;
import tycho.common.tree.converter.CytoscapeTreeConverter;
import tycho.common.tree.converter.SyntacticTreeLayoutConverter;
import tycho.common.tree.cytoscape.CytoscapeEdge;
import tycho.common.tree.cytoscape.CytoscapeNode;
import tycho.common.tree.cytoscape.CytoscapeTree;
import tycho.common.tree.layout.SyntacticTreeLayout;
import tycho.core.helper.SexpressionHelper;
import tycho.core.structure.Chunk;
import tycho.core.structure.SplitToken;
import tycho.core.structure.Struct;
import tycho.core.structure.Token;
import tycho.core.util.StructUtils;
import tycho.core.util.TokenUtils;
import tycho.edictor.tree.vo.SentenceWrapperVO;

/**
 * Converts a chunk into a sentence wrapper.
 *
 * @author Luiz Veronesi
 */
public class SentenceWrapperHelper {

      // TODO: parameters? repeated.
      private static final Integer MIN_NODE_WIDTH = 70;
      private static final Integer MIN_NODE_HEIGHT = 50;

    /**
     * Executes the helper.
```

```
 *
 * @param expression
 * @return SyntacticTreeWrapper
 */
public static SentenceWrapperVO execute(Struct struct, Chunk sentence) throws Exception {

  SentenceWrapperVO wrapper = new SentenceWrapperVO();
  wrapper.setSentence(convertSentence(struct, sentence));

  // get root
  Chunk root = StructUtils.getSyntacticRoot(struct, sentence);
  if (root != null) {
        CytoscapeTree tree = convert(struct, root);
        SyntacticTreeLayout layout = SyntacticTreeLayoutConverter.convert(tree);
        SyntacticTreeLayoutConverter.calculatePositions(layout);
        wrapper.setTree(CytoscapeTreeConverter.convert(layout));
        wrapper.setExpression(SexpressionHelper.execute(layout));
  }

  return wrapper;
}

/**
 * Converts a sentence to a string format.
 *
 * @param struct
 * @param sentence
 */
private static String convertSentence(
  Struct struct, Chunk sentence) {

  StringBuffer sb = new StringBuffer();

  for (Token token : struct.getTokens(sentence.getInitialToken(), sentence.getFinalToken())) {
        sb.append(token.getValue());
        sb.append(TychoTokenizer.WHITESPACE);
  }
```

```
  return sb.toString();
}

/**
 * Converts a chunk corresponding to the root node into a CytoscapeTree.
 *
 * This operation consists of 3 steps:
 * 1- Initializes the tree connecting all leaves to the root.
 * 2- Operates the tree inserting the middle nodes and adjusting edges.
 * 3- Sort the edges.
 *
 * @param struct
 * @param root
 * @return CytoscapeTree
 */
private static CytoscapeTree convert(Struct struct, Chunk root) {
  CytoscapeTree tree = initTree(struct, root);

  operate(tree, struct, root);

  // Sort nodes list
  Collections.sort(tree.getEdges(), new Comparator<CytoscapeEdge>()
  {
        @Override
        public int compare(CytoscapeEdge o1, CytoscapeEdge o2) {
              // comparing edges linking nodes - TODO: test!
              if (o1.getLevel() != null && o2.getLevel() != null) {
                    return o1.getLevel().compareTo(o2.getLevel());
              }

              return o1.getInitialToken().compareTo(o2.getInitialToken());
        }
    });


  return tree;
```

```
}

/**
 * Operates the tree removing all dead edges.
 *
 * @param tree
 * @param struct
 * @param root
 */
private static void operate(CytoscapeTree tree, Struct struct, Chunk root) {

  HashMapList<Integer, Chunk> chunksByLevel = getChunksByLevel(struct, root);

  List<Chunk> chunks = getInnerChunks(struct, root);
  String rootId = tree.getRoot().getId();
  HashMapList<Integer, Token> tokensMapList = createTokensMapList(struct, root);

  for (Chunk chunk : chunks) {

        String nodeId = generateNodeId(chunk);

        // creates new
        CytoscapeNode node = new CytoscapeNode();
        node.setId(nodeId);
        node.setLabel(chunk.getLogical().iterator().next());
        node.setHeight(MIN_NODE_HEIGHT);
        node.setWidth(MIN_NODE_WIDTH);
        tree.addNode(node);

        // find to parent to connect parent and child by an edge
        Chunk parent = findParent(chunksByLevel, chunk);
        String parentId = generateNodeId(parent);

        // add edge between parent and new node
        CytoscapeEdge edge = new CytoscapeEdge(parentId, nodeId);
        edge.setInitialToken(chunk.getInitialToken());
        edge.setFinalToken(chunk.getFinalToken());
```

```
edge.setLevel(parent.getLevel());
tree.addEdge(edge);

// for each chunk, operates adding edges and
// removing edges from root and parent
for (int i = chunk.getInitialToken(); i < chunk.getFinalToken() + 1; i++) {

      // this an error, it can't happen
      if (!tokensMapList.containsKey(i)) continue;

      Token token = tokensMapList.first(i);
      String tokenId = generateTokenId(token);

      CytoscapeEdge intermediateEdge = new CytoscapeEdge(nodeId, tokenId);
      intermediateEdge.setInitialToken(token.getPos());
      tree.addEdge(intermediateEdge);

      tree.removeEdge(new CytoscapeEdge(parentId,tokenId));
      tree.removeEdge(new CytoscapeEdge(rootId, tokenId));

      // the token must be removed from the
      // array after it is appended to the tree
      if (chunksByLevel.containsKey(chunk.getLevel()+1)) {
            List<Chunk> list = chunksByLevel.get(chunk.getLevel() + 1);

            Boolean remove = Boolean.FALSE;
            for (Chunk chunkNextLevel : list) {
                  if (chunkNextLevel.getInitialToken().intValue() >  chunk.getFinalToken().intValue()
                   && chunkNextLevel.getFinalToken().intValue() <= chunk.getFinalToken().intValue())
                        remove = Boolean.FALSE;

                  if (tokensMapList.get(i).size() > 1
                              && chunkNextLevel.getInitialToken().equals(chunk.getInitialToken())
                              && chunk.getInitialToken().equals(token.getPos()))
                        remove = Boolean.TRUE;
                  }
```

```
                    if (remove) tokensMapList.remove(token.getPos(), token);

                } else {
                    tokensMapList.remove(token.getPos(), token);
                }
            }
    }
}


/**
 * Initializes the tree.
 *
 * @param struct
 * @param chunk
 * @return CytoscapeTree
 */
private static CytoscapeTree initTree(Struct struct, Chunk chunk) {
  List<CytoscapeNode> nodes = new ArrayList<>();
  List<CytoscapeEdge> edges = new ArrayList<>();

  CytoscapeNode root = new CytoscapeNode();
  root.setId(generateNodeId(chunk));
  root.setLabel(chunk.getLogical().iterator().next());
  root.setHeight(MIN_NODE_HEIGHT);
  root.setWidth(MIN_NODE_WIDTH);
  root.setRoot(Boolean.TRUE);
  nodes.add(root);

  HashMapList<Integer, Token> tokensMapList = createTokensMapList(struct, chunk);

  for (int i = chunk.getInitialToken(); i < chunk.getFinalToken() + 1; i++) {
        Token token = tokensMapList.first(i);
        initToken(nodes, edges, root, token);
        tokensMapList.remove(i, token);

        if (tokensMapList.containsKey(i)) i--;
```

```
  }

  CytoscapeTree tree = new CytoscapeTree();
  tree.setRoot(root);
  tree.setNodes(nodes);
  tree.setEdges(edges);

  return tree;
}

/**
 * Converts a token into a node at initialization, i.e., connects the token to the root.
 * A token is a pair of nodes, one containing morphological and value data and other
 * containing only value data.
 *
 * @param nodes
 * @param edges
 * @param root
 * @param token
 */
private static void initToken(List<CytoscapeNode> nodes, List<CytoscapeEdge> edges,
        CytoscapeNode root, Token token) {

  CytoscapeNode node = new CytoscapeNode();
  node.setId(generateTokenId(token));
  node.setLabel(token.getTag());
  node.setHeight(MIN_NODE_HEIGHT);
  node.setWidth(MIN_NODE_WIDTH);
  nodes.add(node);

  CytoscapeEdge edge = new CytoscapeEdge(root.getId(), node.getId());
  edge.setInitialToken(token.getPos());
  edges.add(edge);

  CytoscapeNode leaf = new CytoscapeNode();
  leaf.setLabel(token.getValue());
  leaf.setHeight(MIN_NODE_HEIGHT);
```

```
  leaf.setWidth(MIN_NODE_WIDTH);
  leaf.setId(generateLeafId(token));
  nodes.add(leaf);

  edge = new CytoscapeEdge(node.getId(), leaf.getId());
  edge.setInitialToken(token.getPos());
  edges.add(edge);
}

/**
 * Get all chunks associated with this chunk (including it) by level.
 *
 * @param struct
 * @param chunk
 * @return HashMapList<Integer, Chunk>
 */
private static HashMapList<Integer, Chunk> getChunksByLevel(Struct struct, Chunk chunk) {

  HashMapList<Integer, Chunk> chunks = new HashMapList<>();
  chunks.put(chunk.getLevel(), chunk);

  for (Chunk innerChunk : getInnerChunks(struct, chunk)) {
        chunks.put(innerChunk.getLevel(), innerChunk);
  }

  return chunks;
}

/**
 * Get all chunks associated with this chunk.
 *
 * @param struct
 * @param chunk
 * @return List<Chunk>
 */
private static List<Chunk> getInnerChunks(Struct struct, Chunk chunk) {
```

```
  List<Chunk> chunks = new ArrayList<>();

  for (Chunk innerChunk : struct.getChunks()) {
        if (innerChunk.isSyntacticSentence()
                    && innerChunk.getLevel() != null
                    && innerChunk.getLevel().compareTo(chunk.getLevel()) > 0
                    && innerChunk.getInitialToken().compareTo(chunk.getInitialToken()) >= 0
                    && innerChunk.getFinalToken().compareTo(chunk.getFinalToken()) <= 0) {
              chunks.add(innerChunk);
        }
  }

  // Sort nodes list
  Collections.sort(chunks, new Comparator<Chunk>() {
     @Override
     public int compare(Chunk o1, Chunk o2) {
        return o1.getInitialToken().compareTo(o2.getInitialToken());
     }
    });

  return chunks;
}

/**
 * Creates a map list where key is the position of a token for the tree, i.e., split
 * tokens are lists of tokens for a specified position.
 *
 * @param struct
 * @param chunk
 * @return HashMapList<Integer, Token>
 */
private static HashMapList<Integer, Token> createTokensMapList(Struct struct, Chunk chunk) {

  // Creates a map relating all tokens
  List<Token> tokens = struct.getTokens(chunk);

  // Set all tokens by id within a map
```

```
  HashMapList<Integer, Token> mapList = new HashMapList<>();
  for (Token token : tokens) {
        if (TokenUtils.hasSplitTag(token)) {
              for (int i = 0; i < token.getSplitTokens().size(); i++) {
                    SplitToken st = token.getSplitTokens().get(i);
                    st.setIdx(i);
                    st.setPos(token.getPos());
                    mapList.put(token.getPos(), st);
              }
        } else {
              mapList.put(token.getPos(), token);
        }
  }

  return mapList;
}

/**
 * Finds a chunk parent, i.e., the chunk to which the given one is entrenched to.
 *
 * @param chunksByLevel
 * @param chunk
 * @return Chunk
 */
private static Chunk findParent(HashMapList<Integer, Chunk> chunksByLevel, Chunk chunk) {

  for (Chunk parent : chunksByLevel.get(chunk.getLevel() - 1)) {

        int tf = chunk.getFinalToken().intValue();
        int ti = chunk.getInitialToken().intValue();

        int ptf = parent.getFinalToken().intValue();
        int pti = parent.getInitialToken().intValue();

        if (tf <= ptf && ti >= pti) return parent;
  }
```

```
      return null;
    }

    /**
     * Generates a node id.
     *
     * @param chunk
     * @return String
     */
    private static String generateNodeId(Chunk chunk) {
      String label = chunk.getLogical().iterator().next();
      return chunk.getInitialToken() + TychoTokenizer.UNIVERSAL_SPLITTER  + chunk.getFinalToken()
                  +  TychoTokenizer.UNIVERSAL_SPLITTER  +  chunk.getLevel()  +  TychoTokenizer.UNIVERSAL_SPLITTER  +
label;
    }

    /**
     * Generates an id code for a token.
     *
     * @param token
     * @return String
     */
    private static String generateTokenId(Token token) {
      String id = token.getPos().toString() + TychoTokenizer.UNIVERSAL_SPLITTER + token.getTag();

      if (token instanceof SplitToken) {
            id += TychoTokenizer.UNIVERSAL_SPLITTER + ((SplitToken) token).getIdx();
      }

            return id;
    }

    /**
     * Generates an id code for a token leaf.
     *
     * @param token
     * @return String
```

```
     */
    private static String generateLeafId(Token token) {

      String id = token.getPos().toString();

      if (token instanceof SplitToken) {
            id += TychoTokenizer.UNIVERSAL_SPLITTER + ((SplitToken) token).getIdx();
      }

            return id;
    }
}
```