UNIVERSIDADE FEDERAL DA BAHIA
INSTITUTO DE MATEMÁTICA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

MATEUS CHAGAS SOUSA

UM MODELO DE PADRONIZAÇÃO DE DOM PARA EDITORES DE TEXTO CONTENTEDITABLE

SALVADOR
2014

MATEUS CHAGAS SOUSA

UM MODELO DE PADRONIZAÇÃO DE DOM PARA EDITORES DE TEXTO CONTENTEDITABLE

Trabalho de Conclusão de Curso apresentado ao Colegiado do Curso de Bacharelado em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção de título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Ecivaldo de Souza Matos

SALVADOR

2014

MATEUS CHAGAS SOUSA

UM MODELO DE PADRONIZAÇÃO DE DOM PARA EDITORES DE TEXTO CONTENTEDITABLE

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Graduação em Ciência da Computação da Universidade Federal da Bahia - UFBA, como requisito parcial para obtenção de título de Bacharel em Ciência da Computação.

Aprovado em: __/ __/ __

BANCA EXAMINADORA

_____

Prof. Dr. Ecivaldo de Souza Matos
DCC/UFBA
Orientador

_____

Profa. Dra. Débora Abdalla Santos
Universidade Federal da Bahia
Examinador

_____

Prof. Dr. Frederico Araújo Durão
Universidade Federal da Bahia
Examinador

RESUMO

SOUSA, Mateus Chagas. Um modelo de padronização de DOM para editores de texto CONTENTEDITABLE. 2014. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Instituto de Matemática. Departamento de Ciência da Computação. Universidade Federal da Bahia. 2014.

Atualmente editores de texto baseados no atributo contenteditable são comuns, porém a árvore DOM resultante desses editores de texto é frequentemente inconsistente, ambígua e dificilmente personalizável em termos de estrutura. Este trabalho propõe um modelo de padronização de árvores DOM que classifica elementos em três categorias: *section*, *paragraphs* e *inlines*. Demonstra também o processo de implementação do modelo utilizando-se a linguagem JavaScript, os algoritmos resultantes e sua especificação formal em termos da teoria dos conjuntos. Dado que blogs, portais e sites que permitem publicações de terceiros podem se beneficiar ao estabelecerem um padrão para suas publicações, o modelo descrito ao longo do texto apresenta-se como possibilidade para aplicações em que tanto a foma quanto a estrutura do texto apresentado na web são importantes. As restrições do modelo podem ser pensadas também de modo a beneficiar aspectos de SEO, estética e acessibilidade para leitores automáticos de tela. A generalidade do modelo e suas categorias ainda carece de verficação. Uma análise quantitativa de forma e estrutura de textos na web bem como uma análise qualitativa com mantenedores de conteúdo textual na web ou usuários regulares de editores de texto poderia vir a evidenciar necessidades não identificadas neste trabalho.

Palavras-chave: web; DOM; contenteditable; html5; editor de texto.

ABSTRACT

SOUSA, Mateus Chagas. A model for standardization of DOM trees for CONTENTEDITABLE-based text editors. 2014. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Instituto de Matemática. Departamento de Ciência da Computação. Universidade Federal da Bahia. 2014.

Actually contenteditable-based text editors are very common, however the DOM tree resulting from such kind of text editors is frequently inconsistent, ambiguous and hardly customizable in structural concerns. This work proposes a model for standardization of DOM trees that classifies elements in one of three categories: *section*, *paragraph* and *inline*. Also demonstrates the process of implementation of such model using the JavaScript programming language, the resulting algorithms and its formal specification in terms of Set Theory. Given that blogs, portals and websites that allow third party publication can benefit from establishing standards to its publications, the model described throughout the text can possibly be a solution for applications in which both form and structure of the text showed on the web are important. The constraints within the model can be also used to benefit search engine optimization, aesthetics and automated screen readers accessibility. The model's and its categories generality must yet be verified. A quantitative analysis of form and structure of texts published on the web as well as a qualitative analysis with textual content maintainers or even usual text editors users, can point needs that were not addressed in this work.

Keywords: web; DOM; contenteditable; html5; text editor.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
DOM	Document Object Model
HTML	Hyper Text Markup Language
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
WYSISYG	What You See Is What You Get
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Com a chegada da especificação do HTML5 e mais especificamente com a introdução do atributo *contenteditable*, os navegadores de INTERNET, antes presos a caixas de texto comuns como *input* e *textarea*, puderam então abrigar editores de texto complexos que permitiram manipular a própria estrutura HTML da qual são feitas as páginas web. De fato, até mesmo IDEs são acessíveis via navegadores atualmente [Kimpan, 2013].

Apesar de toda flexibilidade provida pelo *contenteditable* este também é fonte de alguns problemas. De fato, documentos criados com esse tipo de editores de texto não possuem restrições e podem ser fontes de ambiguidades em relação à sua representação visual. Diferentes árvores podem ser renderizadas de maneira idêntica por um navegador de internet. Tal característica pode vir a causar dificuldades para plataformas de publicação de usuários que visam aumentar sua visibilidade utilizando técnicas de otimização para motores de busca, ou plataformas que possuam alguma rigidez estética na estrutura de suas publicações.

Tentativas de solução para esses problemas não são incomuns. Editores de texto como WYSIHTML5[1] e o Aloha-editor[2] procuram domar os aspectos do *contenteditable* sobrescrevendo algumas das interfaces de sua API, extendo suas funcionalidades e permitindo filtros e configurações sobre os elementos dos documentos criados por eles. A plataforma de plublicações de textos Medium[3] publicou um artigo, em sua própria plataforma, em que explica a solução adotada por eles para mitigar os problemas dos editores de texto *contenteditable*[4].

Baseado nisso, este trabalho **propõe um modelo de padronização de árvores DOM para ser utilizado em editores de texto que fazem uso da propriedade contenteditable**.

Esta monografia está estruturada da seguinte maneira: no capítulo 2, Fundamentação, são apresentados os temas pesquisados para elaboração desta

[1] https://github.com/xing/wysihtml5
[2] http://aloha-editor.org/
[3] https://medium.com
[4] https://medium.com/medium-eng/why-contenteditable-is-terrible-122d8a40e480

monografia. No capítulo 3, Proposta, está descrito o processo de elaboração da proposta deste trabalho. No capítulo 4, Resultados Obtidos, estão descritos os artefatos obtidos como resultado deste trabalho. Finalmente, no capítulo 5, Considerações Finais, são feitas considerações sobre os resultados obtidos, a aplicabilidade da proposta do trabalho bem como possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO

2.1 DOM

Originado a partir de uma especificação que permitisse scripts em Javascript e programas em Java serem portáveis entre diferentes browsers, o DOM (*Document Object Model*) é uma interface agnóstica, ou seja, pode ser instanciada independente de uma plataforma ou linguagem específica. Isso permite acesso e manipulação dinâmicos via linguagens de programação a diversos aspectos de um documento como conteúdo, estrutura e estilo[5]. Portanto, podemos dizer que se trata de uma API (*Application Programming Interface*) para documentos.

Os documentos representados no DOM possuem uma estrutura lógica semelhante à estrutura de uma árvore, porém, por poder possuir outras árvores dentro dessa mesma estrutura, é mais preciso afirmar que se trata de uma estrutura de floresta. Apesar disso, por se tratar de um modelo lógico para a interface de programação, em sua especificação não constam recomendações para que seja implementado utilizando-se especificamente uma estrutura de árvore ou floresta, logo a estrutura mais conveniente dependerá de cada implementação[6]. Entretanto, para simplificar seu entendimento, neste texto será utilizado o termo “árvore DOM” para se referir à estrutura lógica do DOM.

[5] http://www.w3.org/TR/WD-DOM/introduction.html
[6] http://www.w3.org/TR/DOM-Level-3-Core/introduction.html

```
<table>
  <tbody>
    <tr>
      <td>Shady Grove</td>
      <td>Aeolian</td>
    </tr>
    <tr>
      <td>Over the River, Charlie</td>
      <td>Dorian</td>
    </tr>
  </tbody>
</table>
```

Figura 2.1 - Exemplo de tabela HTML (fonte: http://www.w3.org/TR/DOM-Level-3-Core/introduction.html).

O DOM correspondente à Figura 2.1 está apresentado na Figura 2.2:

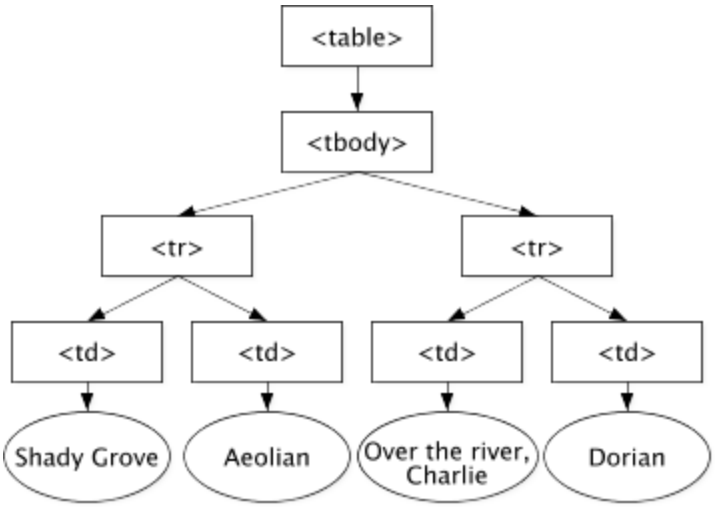


Figura 2.2 - Exemplo de árvore DOM (fonte: http://www.w3.org/TR/DOM-Level-3-Core/introduction.html).

É importante observar o isomorfismo estrutural como propriedade das árvores DOM, ou seja, caso quaisquer duas diferentes implementações do DOM sejam usadas para criar uma representação de um mesmo documento, ambas criarão a mesma árvore DOM[7].

Todo elemento encontrado em um documento, tal como exemplificado na Figura 2.2, faz parte do DOM daquele documento, portanto tais elementos podem ser

[7] http://www.w3.org/TR/DOM-Level-3-Core/introduction.html

acessados por meio de uma árvore DOM e de uma linguagem de script, como JavaScript[8].

A especificação do DOM provê uma série de APIs para interação com os objetos DOM, contudo, em termos de brevidade, abordaremos apenas a *API Core* que possui as interfaces fundamentais para lidar com uma árvore DOM:

- Document
- DocumentFragment
- DocumentType
- EntityReference
- Element
- Attr
- ProcessingInstruction
- Comment
- Text
- CDATASection
- Entity
- Notation

A API Core do DOM possui uma hierarquia de nós, cada um com suas interfaces mais especializadas. Semelhante à noção clássica de árvores em Ciência da Computação, um nó pode possuir nós-filhos ou ser um nó-folha[9]. Para o caso de HTML e XML, simplificadamente, podemos visualizar a estrutura hierarquica das interfaces da API Core no UML da Figura 2.3 .

[8] https://developer.mozilla.org/en-US/docs/Web/JavaScript
[9] http://www.w3.org/TR/DOM-Level-3-Core/introduction.html

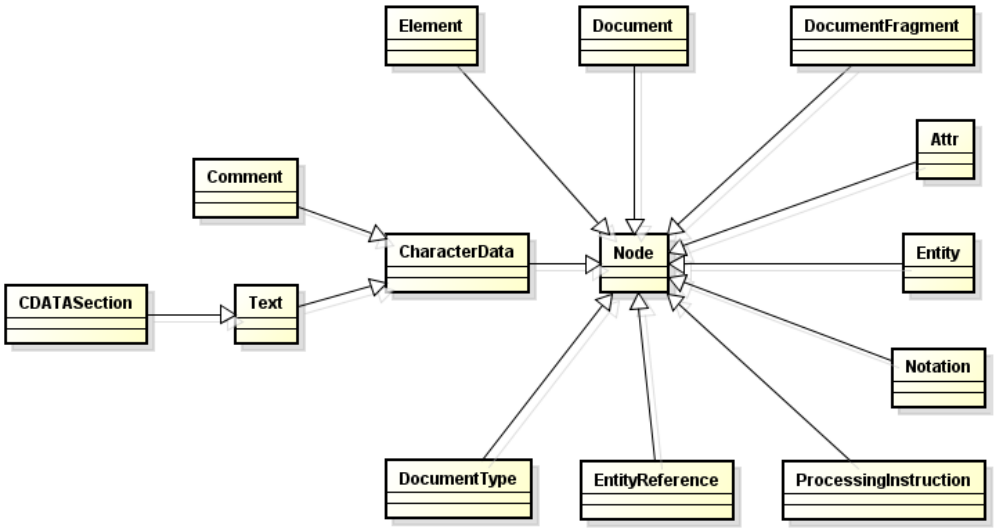


Figura 2.3 - Interfaces da API Core do DOM

Conforme o diagrama da Figura 2.3, observamos que a interface *Node* é a interface primária para o DOM. Essa interface representa um nó de árvore de um documento que pode possuir nós filhos ou ser nó-folha.

Das outras interfaces da API Core, é importante destacar:

1. **Document:** representa um documento em sua totalidade, ou seja, provê acesso à raiz da árvore do documento e a seus atributos. Deve existir no máximo uma por documento.
2. **DocumentFragment:** uma forma mais leve de um Document. Não está de maneira alguma atribuída a um documento, uma vez que é um documento em si. Ao ser adicionado a um documento, seus nós-filhos passam a ser nós-filhos daquele documento.
3. **Element:** representa um elemento em um documento, provendo acesso a seus atributos.

4. **Text**: representa o conteúdo textual de um elemento. É um nó-folha, logo não possui nós-filhos.

2.2 JAVASCRIPT e ECMASCRIPT

O JavaScript é uma linguagem de script desenvolvida na Netscape por Brendan Eich. A princípio foi implementada no Netscape Navigator 2.0 e subsequentemente adotada por outros browsers da época, incluindo o Internet Explorer 3.0 da Microsoft [ECMA, 1999].

Utilizada em milhões de páginas web em todo o mundo e sendo suportada por praticamente todos os navegadores, o JavaScript, junto com outras tecnologias como JScript da Microsoft, serviu como base para a definição do padrão Ecma [ECMA, 1999] e hoje se apresenta como um super-conjunto da 3ª edição da linguagem padrão de script ECMA-262 (ECMAScript)[10].

2.3 CONTENTEDITABLE

O contenteditable é um atributo HTML comum que é suportado, segundo sua especificação[11] por todos os elementos HTML. Elementos com este atributo devem permitir ao usuário realizar uma série de ações dentro do escopo do elemento, dentre elas:

1. Mover o cursor;
2. Mudar seleções, selecionar e retirar seleção de conteúdo;
3. Inserir texto;
4. Quebrar o texto em blocos;
5. Inserir separadores de linhas;
6. Deletar elementos;

[10] https://developer.mozilla.org/en/docs/Web/JavaScript/About_JavaScript
[11] http://www.w3.org/TR/2008/WD-html5-20080610/editing.html

Em suma, o conteúdo de um elemento que possua o atributo contenteditable deve se tornar editável.

2.4 TEORIA DAS CATEGORIAS

Na tentativa de modelar e formalizar as questões abordades neste trabalho, foi feita uma incursão em Teoria das Categorias uma vez que esta pode ser utilizada para para formalizar conceitos de abstração de alto-nível.

Teoria das Categorias consiste basicamente na abstração de conceitos em uma coleção de objetos e setas (também chamadas de morfismos) de modo a obedecer um conjunto de regras que definem sua estrutura.

Um morfismo f é uma estrutura direcional que mapeia de um objeto a (domínio ou origem) em um objeto b (contra-domínio ou alvo) [Rydeheard, 1988].

$$f : a \rightarrow b$$

Contudo, apesar da semelhança, objetos em uma categoria não são necessariamente conjuntos e por sua vez, morfismos não são necessariamente funções, dado que objetos e morfismos são conceitos primitivos em teoria das categorias. Na verdade qualquer acesso à estrutura interna de um objeto é proibido [Asperti, 1991]. É possível construir, por exemplo, uma categoria de deduções naturais em que os morfismos seriam a própria dedução e os o objetos as premissas.

Aplicam-se sobre um morfismo f duas operações: dom e cod que atribuem para cada morfismo um objeto, respectivamente seu domínio e contra-domínio.

Existe um tipo especial de morfismo em todo objeto de uma categoria qualquer, o morfismo identidade (id), onde $dom(id) = cod(id)$. Sendo assim, para um objeto *b*, $dom(id_a) = cod(id_a) = a$.

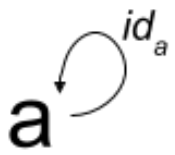


Figura 2.4 - Diagrama exemplificando identidade sobre um objeto

Entre quaisquer dois morfismos f e g, em uma categoria, deve haver uma operação de composição caso $dom(g) = cod(f)$, resultando em um novo morfismo $g \circ f$ em que $dom(g \circ f) = dom(f)$ e $cod(g \circ f) = cod(g)$.

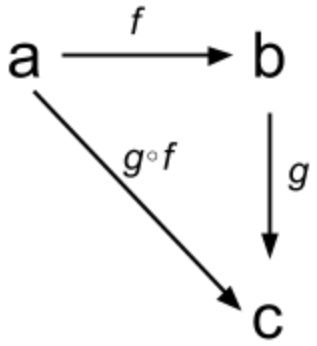


Figura 2.5 - Exemplo composição em uma categoria

Sobre a composição aplicam-se ainda duas propriedades. A primeira, associatividade, diz que para todo morfismo $f : a \to b$, $g : b \to c$ e $h : c \to d$ a igualdade $(h \circ g) \circ f = h \circ (g \circ f)$ é verdadeira. A segunda, a da identidade, diz que para todos os objetos a $f : a \to b$, existe uma identidade i_a tal que, para todo morfismo $f : a \to b$, a igualdade $f \circ i_a = f = i_b \circ f$ é verdadeira [Rydeheard, 1988].

Um morfismo pode ser classificado de diversas maneiras uma vez que satisfaça certas propriedades. Dentre elas está o epimorfismo:

Sejam $f : a \to b$ e $g, h : b \to c$. Existe um epimorfismo se, e somente se, $g \circ f = h \circ f \to g = h$.

3 PROPOSTA

3.1 CONTEXTUALIZAÇÃO

Dentre as características de árvores DOM geradas por editores de texto *contenteditable* que se quer evitar, está principalmente a ambiguidade em relação à sua representação visual. Dadas duas árvores DOM diferentes, é possível que ambas sejam renderizadas de forma idêntica.

Durante o desenvolvimento deste trabalho foram identificados alguns requisitos básicos de modo a mitigar o problema de ambiguidade no DOM gerado em editores de texto contenteditable:

1. Entidades HTML com papéis semelhantes devem ser tratadas de forma semelhante, se possível idêntica, seja para estruturação do documento como para estilização de texto.
2. Deve haver alguma flexibilidade de modo a atender as necessidades de possíveis diferentes aplicações da ferramenta.
3. Deve ser possível armazenar e manipular um documento de forma a não acoplar à uma árvore DOM específica.

De modo a atender esses requisitos, a princípio imaginou-se uma forma de classificação de elementos HTML em categorias abstratas que representasse seu papel em um documento. Desse modo, um elemento HTML passaria a ser menos importante para a estruturação um documento do que a categoria a qual pertence.

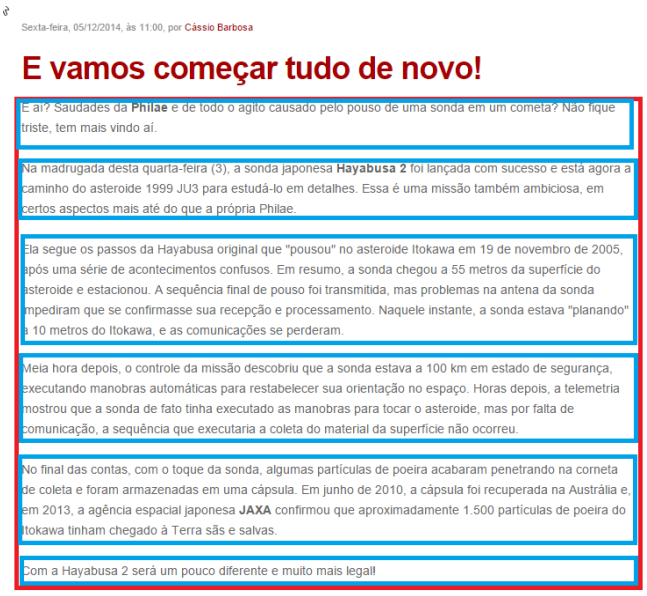
Então, propôs-se classificar elementos HTML em 4 categorias:

1. **DOCUMENT:** O documento em si; agrupamento de seções.
2. **SECTIONS:** Seções; agrupamento de parágrafos.

3. **PARAGRAPHS:** Parágrafos; não necessariamente texto, mas como conteúdo de uma seção.
4. **INLINES:** O texto em si e suas formatações em um parágrafo; utilizado para estilização, hiperlinks.

Uma rápida observação em blogs e portais ajudou também a propor tal classificação, porém vale ressaltar a necessidade de uma análise com uma amostragem mais significativa sobre esta questão. As figuras a seguir ajudam a ilustrar a aplicabilidade desta classificação. Os contornos em vermelho representam seções enquanto os contornos em azul representam parágrafos:

Figura 3.1 - Múltiplas seções com imagem em parágrafo (fonte: http://www.meiobit.com)


Sexta-feira, 05/12/2014, às 11:00, por Cássio Barbosa

E vamos começar tudo de novo!

E aí? Saudades da **Philae** e de todo o agito causado pelo pouso de uma sonda em um cometa? Não fique triste, tem mais vindo aí.

Na madrugada desta quarta-feira (3), a sonda japonesa **Hayabusa 2** foi lançada com sucesso e está agora a caminho do asteroide 1999 JU3 para estudá-lo em detalhes. Essa é uma missão também ambiciosa, em certos aspectos mais até do que a própria Philae.

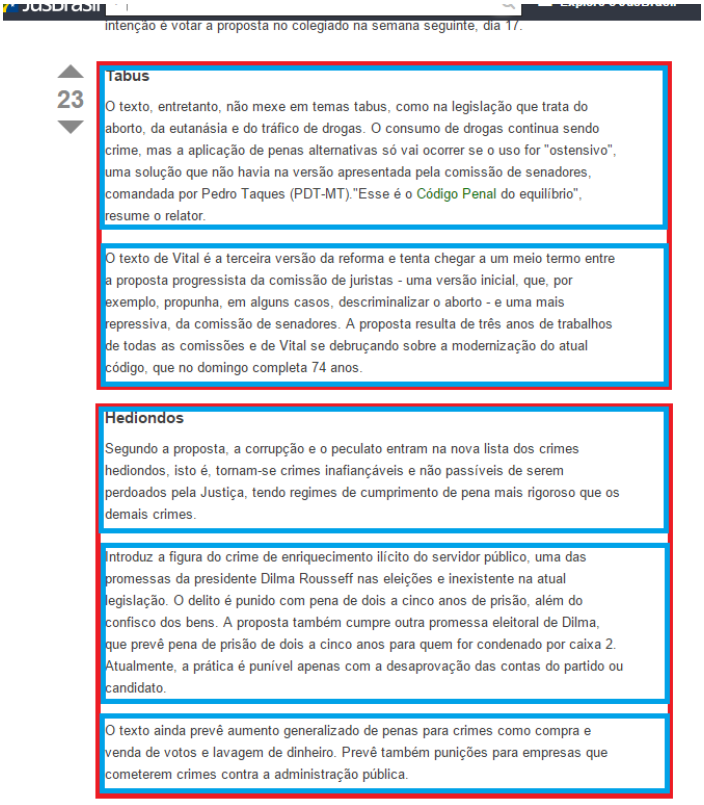
Ela segue os passos da Hayabusa original que "pousou" no asteroide Itokawa em 19 de novembro de 2005, após uma série de acontecimentos confusos. Em resumo, a sonda chegou a 55 metros da superfície do asteroide e estacionou. A sequência final de pouso foi transmitida, mas problemas na antena da sonda impediram que se confirmasse sua recepção e processamento. Naquele instante, a sonda estava "planando" a 10 metros do Itokawa, e as comunicações se perderam.

Meia hora depois, o controle da missão descobriu que a sonda estava a 100 km em estado de segurança, executando manobras automáticas para restabelecer sua orientação no espaço. Horas depois, a telemetria mostrou que a sonda de fato tinha executado as manobras para tocar o asteroide, mas por falta de comunicação, a sequência que executaria a coleta do material da superfície não ocorreu.

No final das contas, com o toque da sonda, algumas partículas de poeira acabaram penetrando na corneta de coleta e foram armazenadas em uma cápsula. Em junho de 2010, a cápsula foi recuperada na Austrália e, em 2013, a agência espacial japonesa **JAXA** confirmou que aproximadamente 1.500 partículas de poeira do Itokawa tinham chegado à Terra sãs e salvas.

Com a Hayabusa 2 será um pouco diferente e muito mais legal!

Figura 3.2 - Uma seção com múltiplos parágrafos (fonte: http://www.globo.com)


intenção é votar a proposta no colegiado na semana seguinte, dia 17.

23

Tabus

O texto, entretanto, não mexe em temas tabus, como na legislação que trata do aborto, da eutanásia e do tráfico de drogas. O consumo de drogas continua sendo crime, mas a aplicação de penas alternativas só vai ocorrer se o uso for "ostensivo", uma solução que não havia na versão apresentada pela comissão de senadores, comandada por Pedro Taques (PDT-MT)."Esse é o Código Penal do equilíbrio", resume o relator.

O texto de Vital é a terceira versão da reforma e tenta chegar a um meio termo entre a proposta progressista da comissão de juristas - uma versão inicial, que, por exemplo, propunha, em alguns casos, descriminalizar o aborto - e uma mais repressiva, da comissão de senadores. A proposta resulta de três anos de trabalhos de todas as comissões e de Vital se debruçando sobre a modernização do atual código, que no domingo completa 74 anos.

Hediondos

Segundo a proposta, a corrupção e o peculato entram na nova lista dos crimes hediondos, isto é, tornam-se crimes inafiançáveis e não passíveis de serem perdoados pela Justiça, tendo regimes de cumprimento de pena mais rigoroso que os demais crimes.

Introduz a figura do crime de enriquecimento ilícito do servidor público, uma das promessas da presidente Dilma Rousseff nas eleições e inexistente na atual legislação. O delito é punido com pena de dois a cinco anos de prisão, além do confisco dos bens. A proposta também cumpre outra promessa eleitoral de Dilma, que prevê pena de prisão de dois a cinco anos para quem for condenado por caixa 2. Atualmente, a prática é punível apenas com a desaprovação das contas do partido ou candidato.

O texto ainda prevê aumento generalizado de penas para crimes como compra e venda de votos e lavagem de dinheiro. Prevê também punições para empresas que cometerem crimes contra a administração pública.

Figura 3.3 - Múltiplas seções com múiltiplos parágrafos (fonte: http://www.jusbrasil.com.br)

Um conjunto de operações sobre essas categorias seria então definido, de modo a permitir manipulação e evolução sobre o mesmo sem dependência de sua árvore DOM. Portanto, deveria ser possível:

1. adicionar e remover seções;
2. adicionar e remover parágrafos;
3. adicionar e remover *inlines*;
4. importar uma árvore DOM para o modelo;
5. exportar o modelo para uma representação DOM.

3.2 REPRESENTAÇÃO FORMAL

De modo a explicitar o problema e a especificar de forma clara o que se espera da solução, decidiu-se pela formalização de ambos. A princípio, devido à semelhança de termos encontrados, foi feito uma tentativa de modelar o problema utilizando *teoria das categorias*.

Nessa abordagem definimos uma categoria C_1 que com três objetos:

1. **D:** o conjunto de todas as árvores DOM possíveis em um editor de texto.
2. **M:** o conjunto de todas as instâncias de um modelo abstrato de padronização de DOM.
3. **V:** o conjunto de todas as representações visuais geradas por um *browser*.

Quanto aos morfismos, por questão de brevidade, vamos abstrair as operações de identidade e assumir que essas são triviais para o nosso caso e focalizar em duas classes de morfismos:

$$f : D \rightarrow M$$

Que mapeia o DOM para um modelo de padronização de árvores DOM.

$$g : M \rightarrow V$$

Que mapeia do anteriormente citado modelo para a representação visual e podemos considerar como abstração da operação de renderização efetuada pelo *browser*.

Estabelecido isso, é possível compor estes dois morfismos, uma vez que $dom(g) = cod(f)$, e portanto, podemos representar nosso mapeamento DOM-Visual como $g \circ f$.

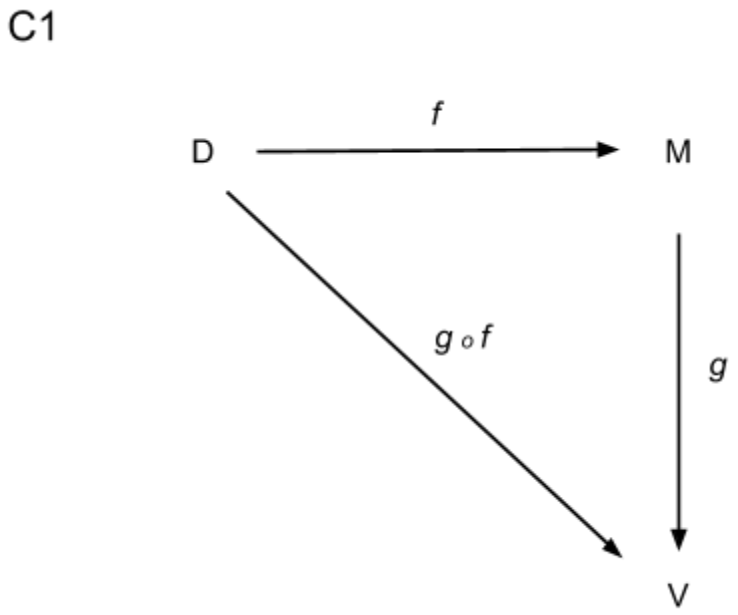


Figura 3.4 - Composição do mapeamento DOM-Visual

Para que seja atingido o objetivo do Modelo proposto, é necessário que haja um epimorfismo de modo que, para $f : DOM \rightarrow MOD$ e $g1, g2 : MOD \rightarrow VIS$ se $g1 \circ f = g2 \circ f$ então $g1 = g2$.

Entretanto, modelar utilizando-se Teoria das Categorias mostrou-se mais difícil do que o esperado. Dada à propriedade atômica dos objetos, não era possível expandi-los para demonstrar as propriedades desejadas. A pouca familiarização prévia com o tema e o pouco tempo para aprofundamento acabaram por limitar o processo de modelagem seguindo tal abordagem. Contudo, foi observada forte semelhança com Teoria dos Conjuntos e dado que havia maior familiarização com o tema decidiu-se então por esta nova abordagem de modo a evitar erros desnecessários durante a formalização.

3.3 IMPLEMENTAÇÃO

Uma vez que a intenção da solução é ser utilizada na implementação de editores de texto WYSISYG que fazem uso da propriedade *contenteditable*, a linguagem escolhida foi o Javascript, uma vez que é o padrão adotado pelos navegadores modernos.

A implementação iniciou-se com a feitura de um arquivo de configuração no formato JSON[12], permitindo uma visão geral da solução e servindo como um artefato concreto deste modelo. Além disso, ter como entrada um arquivo de configuração viria a permitir a flexibilidade desejada, de modo que cada implementação do modelo pudesse definir seu próprio subconjunto do DOM sobre o qual iria operar.

Seguiu-se, então, uma abordagem *bottom-up*, em que foram trabalhados primeiro os níveis mais internos da arquitetura e depois os níveis mais externos. Desse modo, a primeira etapa foi implementar a transformação e ordenação de *inlines* dentro de um parágrafo. A etapa seguinte foi de classificar os nós nas categorias e gerar o modelo.

3.3.1 INLINES

Ao desenvolver o algoritmo de ordenação de inlines observou-se primeiramente uma série de comportamentos desejados:

[12] http://www.json.org/

1. toda subárvore deve ter como folhas nós de texto e logo devem ser eliminadas caso o contrário;
2. após a ordenação a sequência de nós de texto deve se manter igual à da árvore original, de modo a não modificar o sentido do texto;
3. o estilo aplicado sobre os nós de texto deve ser igual ao da árvore original.

Visando atender os requisitos acima, procurou-se desenvolver um algoritmo que permitisse transformar uma árvore desordenada em uma árvore ordenada, dada uma determinada configuração de prioridade de nós que fossem encaixados na categoria de *inline*. A Figura 3.5 ilustra uma árvore desordenada e a árvore para qual esta deve ser transformada.

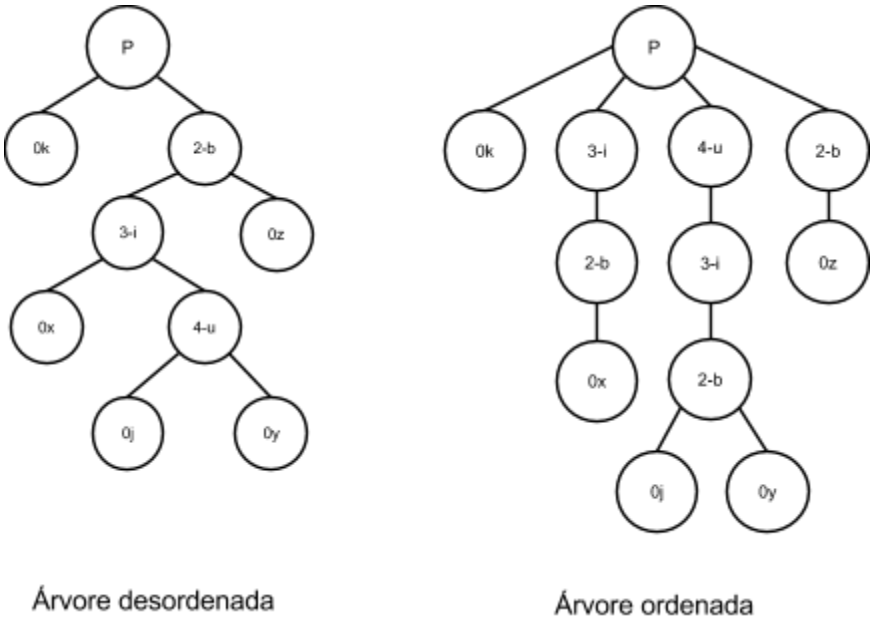


Figura 3.5 - Comparação de árvores ordenada e desordenada

É importante observar que os nós com índice 0 são nós de texto. Ao percorrer os nós-folha, da esquerda para a direita, é possível obter o texto presente na árvore

que deve ser preservado após a transformação, no caso *kxjyz*. Os nós com índice maior que zero, representam nós de estilo e devem ser ordenados de forma crescente, segundo seu índice, no sentido folha-raiz.

Foi encontrada dificuldade em relacionar árvores clássicas e suas transformações. Ao tentar aplicar rotações para ordenar além de se perder a ordem dos nós-folha, observou-se um número excessivo de condições a serem consideradas, o que levou ao desenvolvimento de um algoritmo direcionado à resolução deste tipo de ordenação.

3.3.2 CATEGORIZANDO O DOCUMENTO

A etapa seguinte à implementação envolveu categorizar os elementos de um documento pré-existente. Logo, foi adotada uma abordagem recursiva que leva em consideração a classificação definida no arquivo de configuração.

Dessa maneira, foi desenvolvido um algoritmo que lê uma árvore DOM a partir de sua raiz e a separa em instâncias de uma das três categorias, podendo assim, depois ser reconstruída como uma nova árvore.

3.3.3 TESTES

A cada etapa foi adicionado um conjunto de testes unitários, de modo a garantir o funcionamento pelo menos para entradas e saídas conhecidas. Tais entradas e saídas foram escolhidas visando atender diversos casos gerais de transformação de documentos e diferentes combinações de árvores e elementos HTML. Dentre os cenários previstos nos testes estão:

- verificar se a sequência do texto é mantida após a conversão;
- verificar a ordenação resultante dos inlines;
- verificar se são preservador os estilos de cada nó de texto;
- verificar se os parágrafos são corretamente identificados;
- verificar se as seções são corretamente identificadas;
- verificar se o HTML resultante da transformação está correto;

4 RESULTADOS OBTIDOS

4.1 REPRESENTAÇÃO FORMAL

Após a tentativa de demonstrar o problema utilizando-se teoria das categorias demonstrar-se infrutífera, uma abordagem com teoria dos conjuntos foi adotada com os resultados apresentados nas linhas a seguir.

Sejam D o conjunto de todas as árvores DOM possíveis em um documento de um editor-de-texto contenteditable, V o conjunto de todas as representações visuais possíveis de serem renderizadas por um browser e $r : D \rightarrow V$ uma função que mapeie uma árvore $d \in D$ em uma representação visual $v \in V$.

Suponha que r é injetora, então $\forall di, dj \in D$ se $r(di) = r(dj)$ então $di = dj$. Tal afirmativa é falsa uma vez que, basta utilizar-se de dois nós como âncora (<*a*>) e negrito (<*b*>) sobre um texto e inverter suas hierarquias para obter duas árvores que produzam a mesma representação visual. De fato, é exatamente esta característica que queremos evitar ao propor uma forma de padronizar o DOM para editores de texto.

Deste modo, explicitam-se as propriedades que o modelo devem possuir da seguinte maneira:

Sejam D o conjunto de todas as árvores DOM possíveis em um documento de um editor-de-texto contenteditable, V o conjunto de todas as representações visuais possíveis de serem renderizadas por um browser, M o conjunto de todas instâncias de um modelo de padronização do DOM, $f : D \rightarrow M$ uma função não-injetora que mapeie uma árvore DOM para uma instância do modelo, $g : M \rightarrow V$ uma função injetora que mapeie do modelo para a representação visual.

Uma vez que não lidaremos diretamente com as operações de renderização, deixando isso a cargo da implementação dos browsers, então deve existir um conjunto D', uma função $r' : D' \rightarrow V$ e uma função $t : M \rightarrow D'$, tal que $D' \subset D$ e r' é injetora e t é injetora de modo que $g = t \circ r'$.

O diagrama na Figura 4.1 ajuda a visualizar melhor as propriedades que se deseja obter com o modelo:

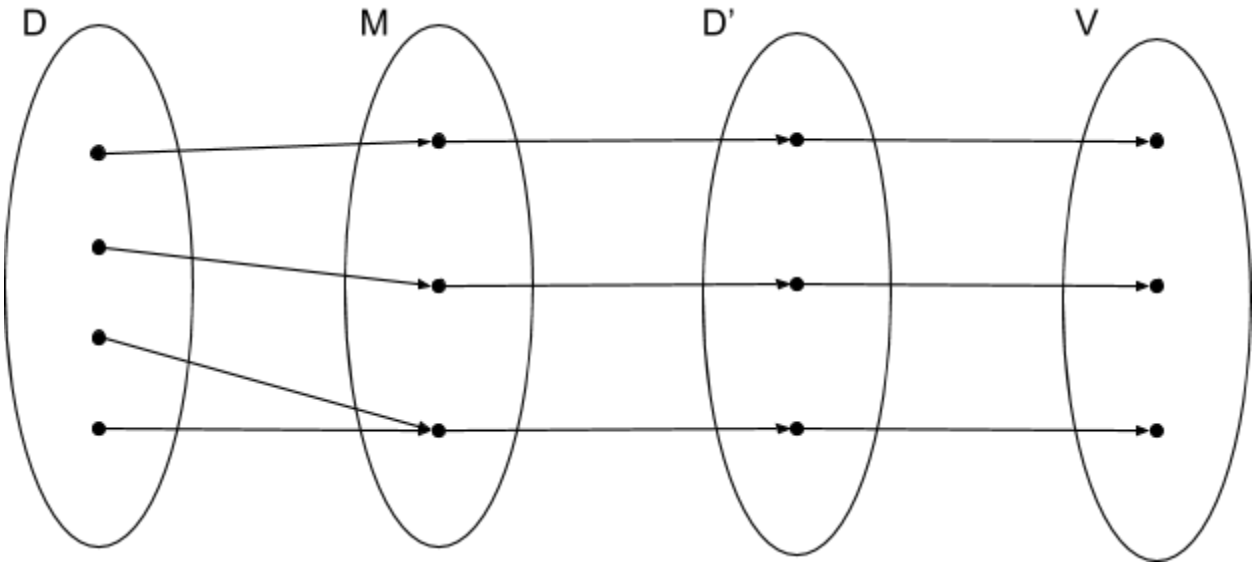


Figura 4.1 - Diagrama de Venn - Injeção entre M e V

Uma vez definidas as propriedades desejadas, podemos finalmente definir um modelo M.

Seja $D1$ o conjunto de árvores DOM com apenas um elemento, ou seja, um conjunto de nós tal que $D1 \subset D$. Definimos portanto os elementos de um M como 5-tuplas (S, P, I, b, r) tal que $S, P, I \subset D1$ e são conjuntos de nós classificáveis em seções, parágrafos e *inlines* respectivamente, enquanto b e r são funções $b : D \rightarrow M$ e $r : M \rightarrow D'$ tais que, b é necessariamente não-injetora e r é necessariamente injetora.

4.2 ARQUITETURA DA SOLUÇÃO

Para cada uma das quatro classificações propostas, foi criada uma classe correspondente conforme UML na Figura 4.2:

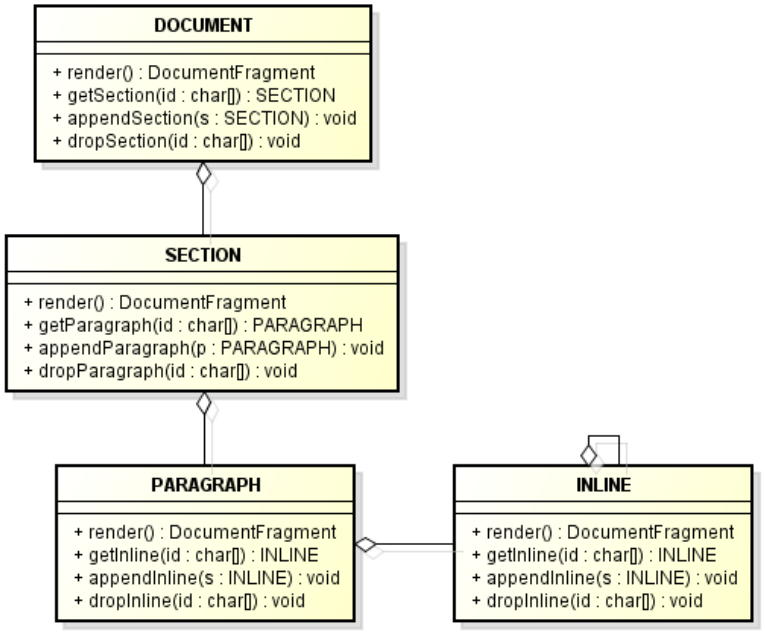


Figura 4.2 - UML modelo de padronização DOM

4.2.1 DOCUMENT

DOCUMENT é a classe que representa um documento em um editor de texto em si. Tal classe possui uma agregação com SECTION e métodos para adicionar e remover seções, repectivamente *appendSection* e *dropSection*, além de um método *render* que retorna a árvore DOM do documento em um DocumentFragment.

Há também outros métodos como *getSection*, que dado um id retorna um seção em um documento e *getSectionList*, que retorna todas as seções de um documento.

4.2.2 SECTION

SECTION é a classe que representa uma seção de documento em um editor de texto. Tal classe possui uma agregação com PARAGRAPAH e métodos para adicionar e remover parágrafos, repectivamente *appendParagraph* e *dropParagraph*, além de um método *render* que retorna a árvore DOM da seção em um DocumentFragment. Toda instância da classe SECTION possui um identificador alfanumérico de dez caracteres gerado de forma aletórea.

Há também outros métodos como *getParagraph*, que dado um *id* retorna um parágrafo daquela seção e *getParagraphList*, que retorna todos os parágrafos de uma seção.

4.2.3 PARAGRAPH

PARAGRAPH é a classe que representa um parágrafo documento em um editor de texto. Tal classe possui uma agregação com INLINES e métodos para adicionar e remover inlines, repectivamente *appendInline* e *dropInline*, além de um método *render* que retorna a árvore DOM do parágrafo em um DocumentFragment.

Há também outros métodos como *getParagraph*, que dado um *id* retorna um parágrafo daquela seção e *getParagraphList*, que retorna todos os parágrafos de uma seção.

4.2.4 INLINE

INLINE é a classe que representa o conteúdo em um documento, bem como estilização de texto. Tal classe possui uma agregação consigo mesma, de modo a abstratir a hierarquia entre INLINES e métodos para adicionar e remover inlines análogos aos métodos de PARAGRAPH. *O* método *render*, por sua vez, retorna a árvore DOM do INLINE em um DocumentFragment.

4.2.5 OUTROS MÉTODOS E CONSIDERAÇÕES SOBRE O MÉTODO RENDER

Há um método global chamado *build* que, dado um elemento raiz de um documento DOM, retorna uma instância completa de DOCUMENT.

O método *render*, presente em todas as classes, funciona de modo a delegar a responsabilidade de renderizar as partes específicas de cada categoria do modelo às suas implementações específicas, ou seja, uma chamada no método render de um DOCUMENT resulta no encadeamento da chamada ao método render de todos os SECTIONS, PARAGRAPHS e INLINES respectivamente.

4.2.6 ARQUIVO DE CONFIGURAÇÃO

Um arquivo de configuração no formato JSON é utilizado para criar instâncias particulares do modelo. É a partir desse arquivo que é possível obter alguma flexibilidade. O arquivo de configuração possui a seguinte estrutura:

1. **defaultSection:** uma string com o nome do nó padrão para seções.
2. **defaultParagraph:** uma string com o nome do nó padrão para parágrafos
3. **sections:** uma lista de objetos JSON com a seguindo a seguinte estrutura
    a. **name:** o nome do nó.
    b. **paragraphs:** um array de strings com nós que devem ser considerados parágrafos dentro daquela seção. Caso este atributo seja vazio ou não fornecido, qualquer parágrafo será aceito, portanto, este é um atributo opcional.
    c. **remove:** um booleano que indica se, caso encontrados nós desse tipo, este deve ser removido. É um atributo opcional, caso não seja fornecido, o nó não será removido.
    d. **rename:** uma string que indica o nó para o qual este deve ser renomeado caso seja encontrado um nó deste tipo. Este é um atributo opcional. O atributo *remove* tem maior prioridade que este caso ambos sejam fornecidos.
    e. **keepContent:** um booleano que indica se, em caso de remoção, o conteúdo do nó deve ser mantido. Caso não seja fornecido este será considerado com o valor *true* e o conteúdo do nó será mantido.
4. **paragraphs:** uma lista de objetos JSON com estrutura semelhante a *sections*, mas no lugar do atributo *paragraphs*, vai o atributo *inlines*, que por sua vez, analogamente, é uma lista com os nós considerados INLINES deste parágrafo.
5. **inlines:** uma lista de objetos JSON com estrutura idêntica a *paragraphs*.

4.3 ALGORITMOS

O algoritmo de transformação dividiu-se em duas partes principais: (i) *identificação e transformação de parágrafos e seções* e (ii) *transformação e ordenação de inlines.*

4.3.1 TRANSFORMAÇÃO E ORDENAÇÃO DE INLINES

O algoritmo consiste basicamente de quatro etapas: identificação dos nós-folha, extração da árvode de *inlines*, ordenação dos nós de acordo com o que foi definido no arquivo de configuração e eliminação dos nós repetidos.

Para exemplificar o funcionamento do algoritmo, tomemos o arquivo de configuração da Figura 4.3 e um trecho de um parágrafo em HTML, Figura 4.4, como exemplo:

```
var config = {
    "inlines" : [
        {
            "name":"#text"
        },
        {
            "name":"B"
        },
        {
            "name":"U"
        },
        {
            "name":"I"
        }
    ]
};
```

Figura 4.3 - Excerto arquivo de configuração. Ordenação de inlines.

O arquivo acima descreve uma ordem de hierarquia na qual um nó de texto é tido como o de maior hierarquia e um nó itálico como o de menor hierarquia.

```
<p>
  I am not in danger, <b>Skyler</b>. <b><u>I am <i>the danger</i></u></b>.
</p>
```

Figura 4.4 - Exemplo HTML de parágrafo

A árvore correspondente ao html acima será:

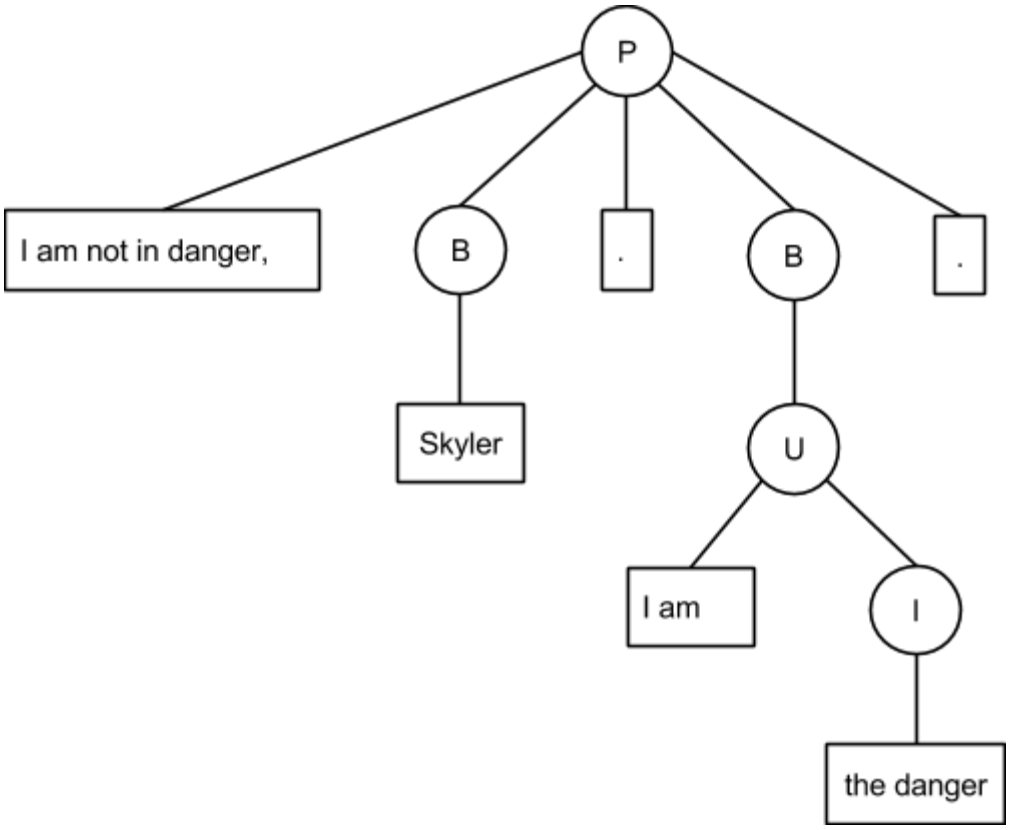


Figura 4.5 - Exemplo de árvore DOM de um parágrafo

Que por fim será renderizada com a seguinte formatação demonstrada na Figura 4.6:

I am not in danger, **Skyler**. **<u>I am *the danger*</u>**.

Figura 4.6 - Exemplo de texto com formatação e estilo HTML

O primeiro passo é identificar os nós-folha, ou seja, os nós de texto. Para tal primeiro é preciso percorrer toda a árvore em busca dos nós folha e separá-los em uma lista. Para o exemplo da Figura 4.5, a lista com os nós de texto está apresentada na Figura 4.7, a seguir.

Figura 4.7 - Exemplo de texto com formatação e estilo HTML

Em seguida, utilizando-se do atributo *parentNode*[13], percorre-se a árvore das folhas para a raiz. A cada nó percorrido insere-se de forma ordenada, de acordo com o arquivo de configuração (Figura 4.3) uma nova instância de INLINE em uma matriz em que cada índice da primeira dimensão horizontal da matriz representa um ramo ordenado da árvore. A Figura 4.8 ilustra o resultado dessa operação:

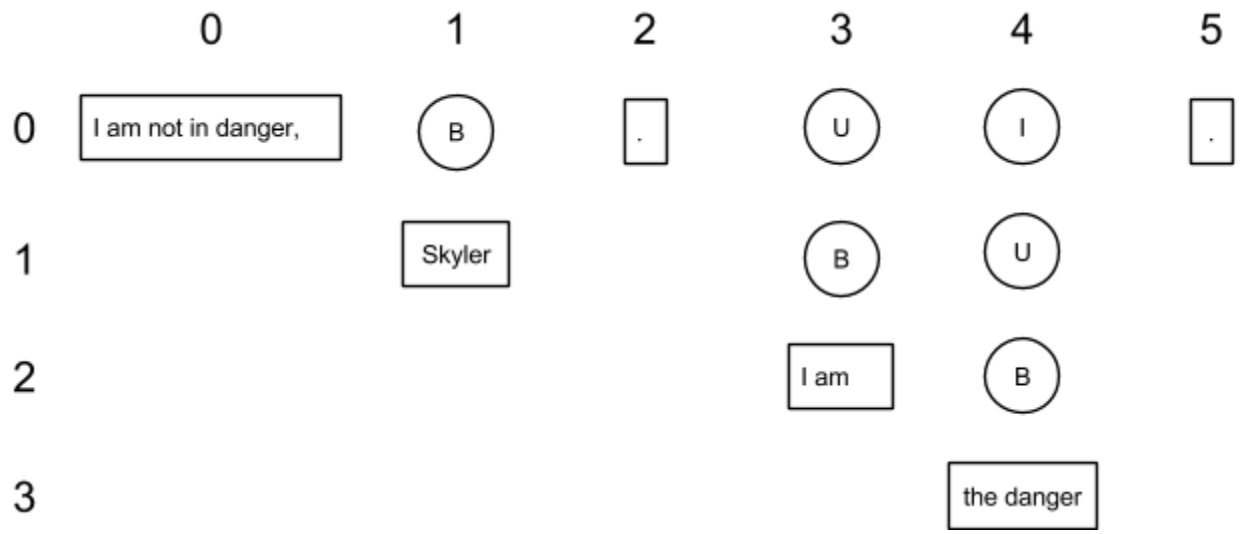


Figura 4.8 - Matriz de inlines ordenados

[13] https://developer.mozilla.org/en-US/docs/Web/API

O algoritmo descrito pode ser visualizado no seguinte pseudo-código:

```
declare function extrairInlines(NODE raiz) returning array<array<INLINE>> as
    var array<NODE> folhas = getFolhas(raiz);
    var array<array<INLINE>> nos;

    foreach folhas as indice => folha do
        var NODE noAtual = folha;
        while noAtual is not null do
            insereOrdenado(nos[indice], new INLINE(noAtual));
            noAtual = noAtual.parentNode;
        endwhile;
    endforeach;

    return nos;
end;
```

Os INLINES então são agrupados utilizando-se o método *appendInline* e depois adicionados a um PARAGRAPH.

O caso em que a primeira etapa do algoritmo exerce mais comparações é o caso em que há uma árvore completa. Para esse caso, o número de acessos a nós será dado em função do produto da largura pela altura da árvore. Sendo então esse o primeiro termo do polinômio para análise da complexidade assintótica de tempo no pior caso. Seja w a largura da árvore e h sua altura, o primeiro termo do polinômio é:

$$w \, . \, h$$

Na segunda etapa, será gerada sempre uma matriz completa de largura igual a

w e altura igual a h uma vez que a matriz é gerada percorrendo-se a partir de todas as folhas para a raiz de uma árvore completa. O algoritmo de ordenação executa em tempo de ordem h, pois este sempre insere ordenado em uma lista vazia, uma vez para cada coluna, ou seja w vezes. Logo, o segundo termo do polinômio é também $w \,.\, h$, resultando no seguinte polinômio:

$$2wh$$

Seja $n = w \,.\, h$, logo o algoritmo de ordenação de inlines tem complexidade de tempo linear no pior caso, ou seja $O(n)$.

4.3.2 IDENTIFICAÇÃO E CLASSIFICAÇÃO DE NÓS

O outro algoritmo importante é o de identificação e classificação de nós. A partir dele é possível converter uma árvore DOM para o modelo.

O primeiro passo do algoritmo em questão consiste em, dado um nó raiz de um documento, percorrer os nós-filho imediatamente abaixo da raiz e identificá-los de acordo com arquivo de configuração.

Se o nó é classificável como uma seção, cria-se uma nova instância de SECTION e partir daí percorrem-se os seus nós-filhos em busca de parágrafos.

Se o nó é classificável como parágrafo, uma SECTION padrão é criada, conforme definido no arquivo de configuração, e o parágrafo adicionado a ela. Segue-se então explorando todos os nós irmãos desse parágrafo de modo a adicioná-los como parágrafos da seção recém-criada até que se encontre uma nova seção ou o fim do documento.

Se um nó não é identificado como parágrafo e nem como seção é então classificado como INLINE, uma nova instância de PARAGRAPH é criada com o parágrafo padrão e o INLINE é adicionado a ela. Segue-se então a exploração de todos os nós irmãos daquele, de modo a identificar novos INLINES até que se encontre um novo parágrafo.

Sempre que um parágrafo é identificado ou criado, este deve ser submetido ao algoritmo de transformação e ordenação de INLINES. Porém, antes disso é necessário que se faça uma limpeza no parágrafo de modo a remover os nós que não se encaixarem na classificação de inline. Portanto, quaisquer nós encontrados que não forem classificados como INLINES são substituídos por seu conteúdo textual apenas.

5 CONSIDERAÇÕES FINAIS

Espera-se que o modelo seja útil a aplicações em que tanto a foma quanto a estrutura do texto apresentado na web são importantes. Blogs, portais e sites que permitam publicações de terceiros podem se beneficiar ao estabelecerem um padrão para suas publicações. As restrições do modelo podem ser pensadas também de modo a beneficiar aspectos de otimização para motores de busca, estética e acessibilidade para leitores automáticos de tela.

Contudo, em seu estado atual, o modelo, tanto em sua forma abstrata quanto em sua implementação, carece ainda de maturação. É possível aplicá-lo para transformação e padronização de documentos completos sem muito esforço (por exemplo, após concluir a edição do documento e clicar no botão de salvar), porém sua aplicação em um editor de texto em tempo real pode vir a causar problemas de usabilidade uma vez que não foram implementadas funções que levem em consideração, por exemplo, a *SelectionAPI*[14], posicionamento do cursor, o que reduz em muito a aplicabilidade do modelo para editores de texto reais.

É importante levar em consideração também que nós não são a única forma de definir conceitualmente uma divisão entre uma seção ou parágrafo. Espaços em branco, quebras de linhas, cabeçalhos, divisões horizontais também podem agir como fronteiras entre seções. Portanto, é necessário implementar esse conceito de modo a aumentar a flexibilidade do modelo.

A implementação[15] do modelo ainda depende que o arquivo de configuração seja definido de de forma correta, não só sintaticamente, mas semanticamente. Não

[14] http://www.w3.org/TR/selection-api/
[15] https://github.com/matchs/thoth-wysiwyg

há uma etapa de verificação de duplicidades, dependências cíclicas e afins, de modo que é fácil que uma definição errada cause inconsistências no modelo e a verificação das inconsistências passa a ser essencialmente empírica.

A generalidade do modelo e suas categorias ainda carece de verficação. Uma análise quantitativa de forma e estrutura de textos na web bem como uma análise qualitativa com mantenedores de conteúdo textual na web ou usuários regulares de editores de texto poderia vir a evidenciar necessidades não identificadas neste trabalho.

6 REFERÊNCIAS

[Kimpan, 2013] Kimpan, W.; Meebunrot, T.; Sricharoen, B. Online code editor on Private cloud computing. *Computer Science and Engineering Conference (ICSEC), 2013 International* 2013. Bangkok, Tailândia. P 4-6

[ECMA, 1999] ECMAScript Language Specification, Third Edition. European Computer Manufacturers Association, Standard ECMA-262, Dezembro 1999. Disponível em http://www.ecma-international.org/ecma-262/5.1/

[Rydeheard, 1988] Rydeheard, D.E.; Burstall, R.M. Computational Category Theory. Prentice Hall. 1988. P. 257

[Asperti, 1991] Asperti, Andrea. Longo, Giuseppe. Categories, Types And Structures: An Introduction to Category Theory for the working computer scientist. MIT Press. 1991. P. 300