UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

WEBCODE: COMPONENTE WEB PARA EDIÇÃO DE CÓDIGO FONTE

LEONARDO ZORZO CARBONE

BLUMENAU
2008

2008/2-13

LEONARDO ZORZO CARBONE

WEBCODE: COMPONENTE WEB PARA EDIÇÃO DE CÓDIGO FONTE

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Adilson Vahldick - Orientador

BLUMENAU
2008

2008/2-13

WEBCODE: COMPONENTE WEB PARA EDIÇÃO DE CÓDIGO FONTE

Por

LEONARDO ZORZO CARBONE

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Adilson Vahldick, M.Sc. – Orientador, FURB
Membro:	Prof. Wilson Pedro Carli, M.Eng. – FURB
Membro:	Prof. Joyce Martins, M.Sc. – FURB

Blumenau, 13 de fevereiro de 2009

Dedico este trabalho aos meus pais, meus amigos, meu orientador e especialmente para aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça e pela vida maravilhosa que tenho.

Aos meus pais, Odilon e Maria de Lourdes pela oportunidade de estar cursando esta Universidade e por sempre acreditarem em mim.

À minha irmã Ângela e meu cunhado Diogo, pelas brigas que tivemos, as quais só me ajudaram a crescer e pelos ótimos momentos de descontração e diversão.

À minha namorada, Sabrina, pela paciência e apoio.

A todos meus amigos e colegas de curso, pelos bons momentos de risadas durante o intervalo das aulas e pelas dicas e sugestões para este trabalho.

Ao meu orientador, Adilson Vahldick, por ter acreditado na conclusão deste trabalho.

Dominar a agressividade, suavizar as arestas, moderar as palavras.

Masaharu Taniguchi

RESUMO

Este trabalho apresenta o desenvolvimento de um componente web para edição de código fonte. O componente, representado por uma *tag* HTML customizada, disponibiliza recursos de *syntax highlighting*, auto complemento de código, numeração de linhas e um mecanismo de *plugins* para o reconhecimento das linguagens. Também disponibiliza uma barra de ferramentas com opções para abrir, salvar e criar um novo código e ações de refazer e desfazer e copiar, colar e recortar. O componente foi desenvolvido com as linguagens de programação Java e JavaScript, o *frame*work de componentes Java Server Faces e a biblioteca FlexJSON. Para demonstrar sua utilização, o componente foi adicionado em uma aplicação web desenvolvida em outro TCC, que possui um editor de código sem funcionalidades.

Palavras-chave: AJAX. JSF. DOM. Análise léxica. *Plugin*.

ABSTRACT

This paper presents the development of a web component to edit source code. The component, represented by a custom HTML tag, offers features for syntax highlighting, code completion, line numbers and a plugin mechanism for the recognition of program languages. It also offers a toolbar with options to open, save and create a new code and actions of undo and redo, copy, paste and cut. The component was developed with the programming languages Java and JavaScript, the Java Server Faces component framework and the FlexJSON library. To demonstrate its use, the component was added in an web application developed in another TCC that contains a code editor without functionalities.

Key-words: AJAX. JSF. DOM. Lexical analysis. Plugin.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

AJAX – *Asynchronous Javascript And XML*

CSS – *Cascading Style Sheets*

DOM – *Document Object Model*

DWR – *Direct Web Remoting*

HTML – *HyperText Markup Language*

HTTP – *HyperText Transfer Protocol*

JCP – *Java Community Process*

JEE – *Java Enterprise Edition*

JSF – *Java Server Faces*

JSON – *JavaScript Object Notation*

JSP – *Java Server Pages*

MVC – *Model View Controller*

RF – Requisito Funcional

RNF – Requisito Não-Funcional

TLD – *Tag Library Descriptor*

UML – *Unified Modeling Language*

URI – *Uniform Resource Identifier*

XML – *eXtensible Markup Language*

XSLT – eXtensible Stylesheet Language Transformation

SUMÁRIO

1 INTRODUÇÃO

Soares (2006, p. 11) afirma que a internet está em constante evolução e atualmente é um dos principais meios de comunicação. Juntamente com a evolução da internet, as linguagens de programação evoluíram e popularizaram-se, e novas ferramentas de programação surgiram no mercado (CARDOSO, 2004, p. 5). Estas ferramentas oferecem em um único ambiente diversas funcionalidades necessárias para a implementação de sistemas como compilação, depuração, execução e funcionalidades para edição de código fonte como auto complemento e *syntax highlighting*[1], funções muito úteis durante o desenvolvimento.

Atualmente umas das ferramentas de programação mais populares é o Eclipse, que se diferencia de outras ferramentas por ser extensível a várias linguagens de programação através de um sistema de *plugins*[2] (OLIVEIRA, 2005).

As tecnologias para o desenvolvimento de aplicações web também passaram por uma grande evolução: criadas novas ferramentas e aperfeiçoadas as tecnologias existentes, proporcionando maior controle e interatividade (GONÇALVES, E., 2006, p. 3).

O uso de linguagens como Java e *Personal Home Page* (PHP) proporcionou a criação de aplicações web mais robustas e com acesso a banco de dados, mas foi com o uso de tecnologias já existentes nos navegadores de internet, como JavaScript e *eXtensible Markup Language* (XML), que os desenvolvedores trocaram o modelo tradicional de desenvolvimento, onde toda a página *HyperText Markup Language* (HTML) era recarregada a cada requisição, por um modelo dinâmico onde pequenas partes da página HTML são recarregadas (GONÇALVES, B., 2006, p. 11). Gonçalves B. (2006, p. 11) ainda cita que esta nova abordagem de desenvolvimento possibilitou trazer as características das aplicações *desktop* para as aplicações web, tornando-as mais atraentes e interativas. Com isso numerosos produtos gratuitos e comerciais surgiram a fim de tirar vantagem deste novo modelo de desenvolvimento que foi chamado de *Asyncronous JavaScript And XML* (AJAX).

Observando o acima descrito, este trabalho apresenta o desenvolvimento de um componente web para edição de código fonte, utilizando o *framework*[3] *Java Server Faces* (JSF). Este componente cria na página onde está inserido, um editor de código fonte com as funcionalidades de numeração de linhas, auto complemento de código, *syntax highlighting* e

[1] Segundo Carlos (2007), é uma funcionalidade dos editores de código no qual o código é realçado com cores.
[2] É um pequeno programa que aumenta as funcionalidades de outros programas (ANSWERS, 2006).
[3] Conjunto de classes e interfaces que cooperam para resolver um problema de software (ARAÚJO, 2005).

uma barra de ferramentas com opções para criar, abrir e salvar código, copiar, colar e recortar e ações de refazer e desfazer. Além destas funcionalidades, o componente também possui um mecanismo de *plugins* que possibilita o reconhecimento de várias linguagens de programação. A utilização do componente é feita através de *tags*[4] HTML customizadas inseridas no corpo do documento HTML.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um componente web para edição de código fonte, com suporte a várias linguagens de programação por meio de *plugins*.

Os objetivos específicos do trabalho são:

a) disponibilizar um mecanismo para definir e anexar *plugins*;
b) disponibilizar meios para abrir e salvar o código digitado;
c) criar meios para inserir o componente em uma página HTML;
d) disponibilizar recursos de *syntax highlighting*, auto complemento de código e numeração de linhas.

1.2 ESTRUTURA DO TRABALHO

A estrutura deste trabalho está dividida em capítulos que serão explicados a seguir.

O primeiro capítulo apresenta a contextualização e objetivos a serem alcançados no desenvolvimento do trabalho.

O segundo capítulo apresenta a fundamentação teórica do trabalho, arquitetura AJAX, arquitetura do *Document Object* Model (DOM), o *framework* JSF, formato JavaScript *Object Notation* (JSON), análise léxica e por último a apresentação dos trabalhos correlatos.

O terceiro capítulo apresenta os requisitos do componente, sua especificação, implementação, bem como sua utilização. O quarto capítulo apresenta as conclusões, limitações e sugestões para futuros trabalhos.

[4] *Tags* são símbolos especiais que dizem como o texto deve ser exibido no navegador (RAMALHO, 1996, p. 6).

2 FUNDAMENTAÇÃO TEÓRICA

A seguir são explanados os principais assuntos relacionados ao desenvolvimento do trabalho: AJAX, DOM, *framework* JSF, JSON, análise léxica e os trabalhos correlatos.

2.1 AJAX

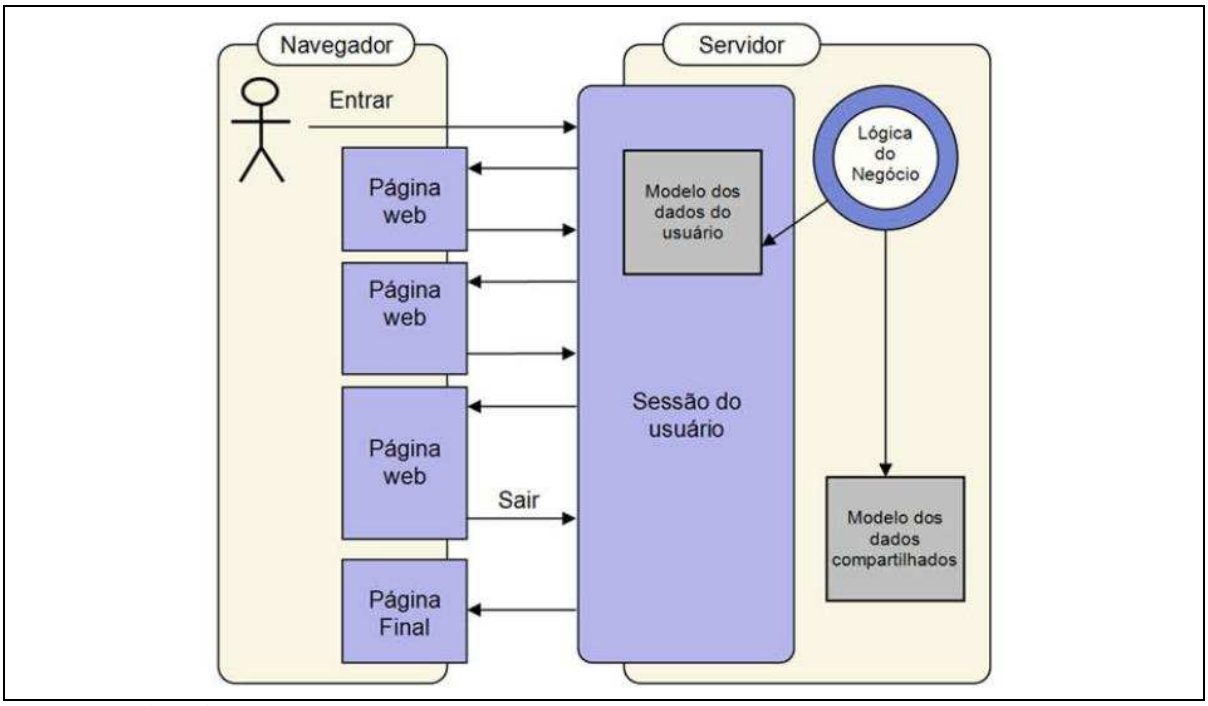
O termo AJAX foi definido por Jesse James Garret em fevereiro de 2005 em um artigo intitulado “AJAX, A New Approach to Web Applications” (GARRET, 2005).

Segundo Garret (2005), AJAX não é uma, mas um conjunto de tecnologias onde cada uma delas evolui por si própria, que quando usadas em conjunto possibilitam o desenvolvimento de aplicações web com interfaces mais sofisticadas. Para Jacobi e Fallows (2007, p. 174), AJAX é um termo que descreve uma técnica de desenvolvimento web.

No modelo tradicional de aplicações web, a interação com o usuário é feita de forma síncrona (CRANE; PASCARELLO; JAMES, 2007, p. 20), ou seja, o usuário cria uma requisição através de uma solicitação que pode ser feita por um formulário ou *link*, o navegador de internet gera uma solicitação ao servidor usando o protocolo *HyperText Transfer Protocol* (HTTP), o servidor recebe a requisição, processa-a e gera uma resposta (KURNIAWAN, 2002, p. vi). Para cada solicitação feita atualiza-se todo o conteúdo novamente, não importando se apenas parte dele tenha sido atualizado. Para Jacobi e Fallows (2007, p. 178), este processo de requisição-resposta produz um efeito indesejado no qual a página HTML pisca ao ser recarregada na janela do navegador.

Antes do AJAX, alguns recursos eram utilizados para amenizar este problema. O uso de pré-carregamento de dados e elementos HTML como `frames` e `iframes` proporcionavam ao desenvolvedor uma maneira de resolver o problema mas nunca funcionavam de forma eficaz e apenas davam a impressão de que a página não era totalmente recarregada (LIMEIRA, 2006, p. 12).

Na Figura 1 é mostrada a estrutura tradicional de uma aplicação web.



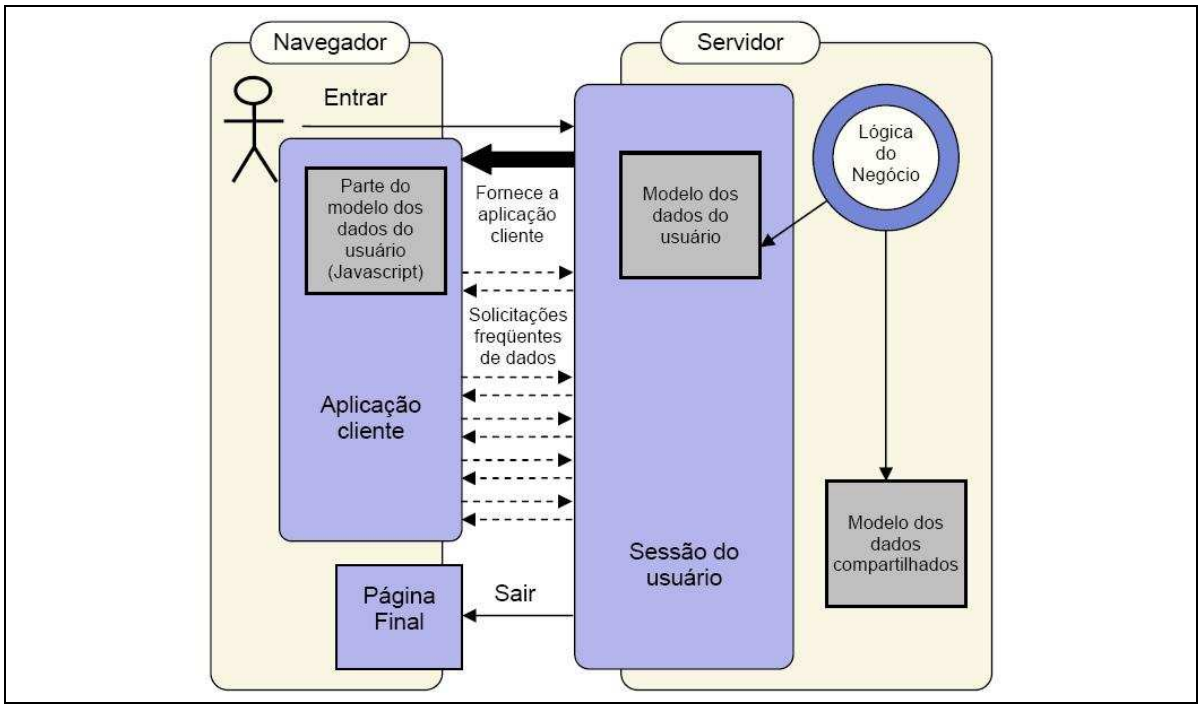
Fonte: Sousa (2006).

Figura 1 - Estrutura tradicional de uma aplicação web

Conforme Crane, Pascarello e James (2007, p. 21), com o uso do AJAX as aplicações tornam-se mais interativas e rápidas, visto que apenas os dados que realmente foram atualizados é que irão trafegar na rede. A interação dos usuários com a aplicação web passa a ser assíncrona, ou seja, não precisam esperar que toda a página seja processada a cada vez que fazem uma solicitação. O usuário pode interagir com uma parte da página e enquanto aguarda o retorno da solicitação pode continuar interagindo com as demais partes.

Ao contrário do modelo tradicional, o AJAX provê uma camada intermediária constituída de funções JavaScript que controlam as requisições ao servidor. Estas funções são chamadas sempre que uma informação precisa ser solicitada ou enviada ao servidor (GONÇALVES, E., 2006, p. 5). Dessa maneira toda página é carregada uma única vez no navegador e somente pequenas partes dela são atualizadas com os novos dados solicitados pelo usuário (GONÇALVES, E., 2006, p. 2).

Na Figura 2 é mostrada a estrutura de uma aplicação web utilizando AJAX.



Fonte: Sousa (2006).

Figura 2 – Estrutura de uma aplicação web utilizando AJAX

Garret (2005) afirma que várias tecnologias fazem parte de uma solução AJAX, sendo elas:

a) *Cascade Style Sheets* (CSS): segundo Soares (2006, p. 49), CSS, ou folhas de estilo, é uma ferramenta que permite definir toda a aparência de uma página web, tornando a experiência do usuário mais interessante. A estilização de uma página web pode ser modificada interativamente através de regras de estilos definidas em um arquivo separado ou até mesmo na própria página;

b) DOM: é uma interface que possibilita que documentos HTML e XML sejam manipulados como um conjunto de nós (SOARES, 2006, p. 67). Soares (2006, p. 67) ainda cita que com o uso do DOM e JavaScript é possível acessar diretamente qualquer elemento de um documento, pois a estrutura da página é apresentada como uma estrutura de árvore. A alteração dinâmica do documento só é possível com a utilização do DOM (TEAGUE, 2001, p. 177);

c) XML: para Gonçalves (2006b, p. 128), o XML possui como característica sua semelhança em *tags* e atributos com o HTML. Diferentemente do HTML, um documento XML não pode conter *tags* desordenadas, ou seja, se uma *tag* for aberta e outra *tag* for inserida dentro da mesma, é necessário que a *tag* atual seja fechada antes de se fechar a primeira *tag.* Ainda segundo Gonçalves E. (2006, p. 129), um

documento XML não é responsável pela formatação das informações, somente é responsável pelo conteúdo a ser apresentado;

d) *eXtensible Stylesheet Language Transformation*: segundo Limeira (2006, p. 18), XSLT é uma linguagem para transformar documentos XML, onde é possível criar um documento XML a partir de outro documento, inserindo, alterando ou excluindo nós. Ainda cita que é uma linguagem declarativa no qual se podem especificar regras que irão transformar um arquivo XML no formato desejado. Gonçalves E. (2006, p. 142) afirma que o XSLT é uma linguagem baseada em modelos, onde o processador trabalha em um documento XML aplicando regras desse modelo;

e) JavaScript: é uma linguagem baseada em objetos desenvolvida para criar aplicações de internet, não sendo compilada e sim interpretada pelo navegador de internet (CARDOSO, 2004, p. 64). Segundo Silva (2003, p. 22), a utilização do JavaScript permite a validação de dados, exibição de mensagens e efeitos dinâmicos, dando maior enriquecimento a uma página web;

f) `XmlHttpRequest`: é o objeto que torna possível a comunicação assíncrona entre uma página e o servidor de aplicações web, sendo a tecnologia principal do AJAX (JACOBI; FALLOWS, 2007, p. 175). Seu detalhamento é apresentado na seção 2.1.1.

A Figura 3 apresenta como todas estas tecnologias trabalham em conjunto em uma requisição AJAX. Segundo Murray (2008), os passos apresentados na Figura 3 são os seguintes:

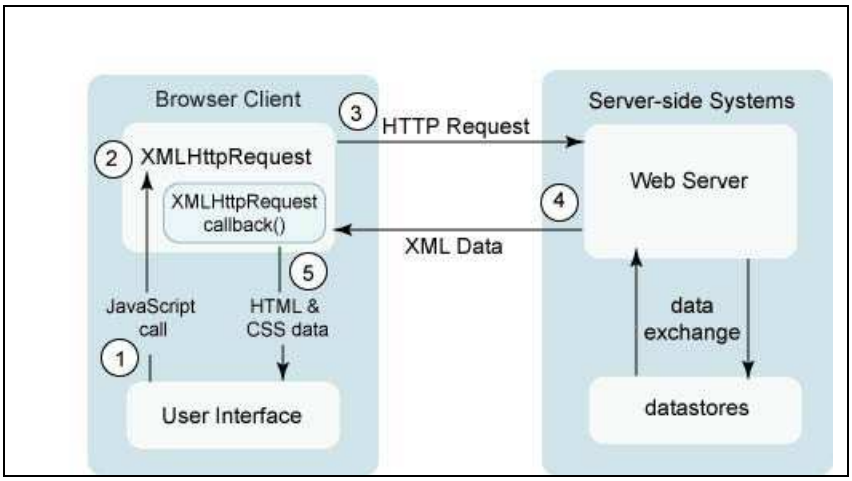
a) passo 1: o usuário gera um evento na interface da aplicação, como um clique em um botão. Isso resulta em uma chamada JavaScript;

b) passo 2: o objeto `XmlHttpRequest` é criado e configurado para fazer uma requisição;

c) passo 3: o objeto `XmlHttpRequest` realiza uma requisição assíncrona para o servidor. O servidor recebe a requisição e processa-a;

d) passo 4: o servidor retorna para o cliente o resultado do processamento através de um XML ou um formato texto;

e) passo 5: o objeto `XmlHttpRequest` recebe os dados do servidor e executa uma função *callback* que irá atualizar os dados na página.



Fonte: Murray (2008).

Figura 3 – Seqüência de uma requisição AJAX

2.1.1 *XmlHttpRequest*

Para Limeira (2006, p. 18), trata-se de um objeto JavaScript que pode ser usado para fazer requisições ao servidor, em segundo plano, sem congelar o navegador ou recarregar toda a página. Foi criado inicialmente no navegador Internet Explorer 5, sendo disponibilizado através de um componente ActiveX. Em outros navegadores como Mozilla Firefox, Konqueror, Safari e Opera, é disponibilizado como um objeto JavaScript nativo. Realiza a troca de informações com o servidor normalmente no formato XML, embora possa utilizar qualquer formato de texto como JSON.

Segundo Soares (2006, p. 83), este objeto é parte da especificação DOM nível 3, sendo necessária sua implementação em qualquer navegador que ofereça suporte aos padrões do *World Wide Web Consortium* (W3C).

O Quadro 1 apresenta uma função JavaScript que cria o objeto `XmlHttpRequest` no navegador Internet Explorer e nos demais navegadores.

```
function criaHttpRequest()
{
   if (window.XMLHttpRequest)//Mozilla Firefox, Konqueror, Safari, etc...
      return new XMLHttpRequest();
   else
      return new ActiveXObject("Microsoft.XMLHttp"); //Internet Explorer
}
```

Quadro 1 – Criação do objeto `XmlHttpRequest`

Limeira (2006, p. 19) afirma que o objeto `XmlHttpRequest` possui os seguintes métodos e atributos listados no Quadro 2.

Método	Descrição
`Open(método, url, asynch)`	Método que prepara um pedido HTTP ao servidor, especificando no terceiro parâmetro se a solicitação vai ser síncrona ou assíncrona.
`Send()`	Método que envia a solicitação ao servidor.
`SetRequestHeader()`	Método que configura o cabeçalho HTTP.
`GetResponseHeader(header)`	Método que retorna o valor da `string` do cabeçalho especificado.
`Abort()`	Método que interrompe o processamento atual do objeto.
`Status`	Atributo que contém o código do *status* enviado pelo servidor: 200 (OK) e 404 (não encontrado).
`statusText`	Atributo que contém o *status* enviado pelo servidor em formato texto.
`readyState`	Atributo que contém o estado da solicitação: 0 (não inicializada), 1 (carregando), 2 (carregada), 3 (interativa), 4 (concluída).
`responseText`	Atributo que contém a resposta do servidor em forma de `string`.
`responseXML`	Atributo que contém a resposta do servidor em formato XML.
`OnReadyStateChange`	Manipulador de eventos que recebe uma função que é executada sempre que o valor do atributo `readyState` é alterado.

Quadro 2 – Métodos e atributos do objeto `XmlHttpRequest`

2.2 DOM

É uma representação hierárquica dos objetos de um documento HTML ou XML (SOARES, 2006, p. 67). Crane, Pascarello, James (2007, p. 31) definem esta representação como "[...] uma estrutura de árvore onde cada elemento representa uma *tag* HTML[...]".

Esta estrutura de árvore é composta de elementos ou nós, que podem conter nós filhos dentro deles e assim sucessivamente. O nó raiz da árvore é exposto pelo JavaScript através da variável global `document` que é usada como ponto de partida para acessar os elementos da árvore (CRANE; PASCARELLO; JAMES, 2007, p. 32).

Para Teague (2001, p. 178), o DOM funciona como um mapa onde é possível localizar objetos dentro de um documento e manipulá-los através de funções JavaScript.

Por ser definido pelo W3C, o DOM pode ser agrupado em cinco categorias, que são:

a) DOM *core:* modelo genérico especificado para ver e manipular um documento;

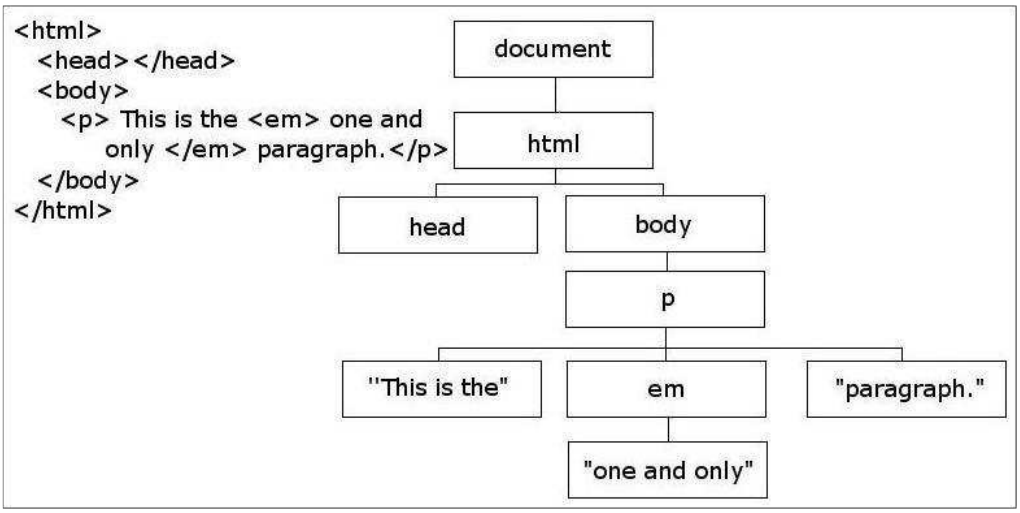
b) DOM HTML: extensão do DOM *core* para uso com HTML;

c) DOM CSS: provê interfaces necessárias para manipular regras de estilo;

d) DOM *events*: adiciona eventos de controle ao DOM. Os eventos podem ser de interface, como cliques do *mouse* ou eventos que ocorrem ao se modificar a estrutura de um documento;

e) DOM XML: extensão do DOM *core* para uso com XML.

A Figura 4 apresenta um trecho de código HTML e sua respectiva representação na árvore DOM.



Fonte: Goodman e Morrison (2004, p. 41).

Figura 4 – Exemplo de uma estrutura DOM

2.3 JSF

"*Java Server Faces* (JSF) é um *framework* de componentes de interface de usuário (*User Interface* – UI) para aplicações web Java Enterprise Edition (JEE)" (JACOBI; FALLOWS, 2007, p. 1). Jacobi e Fallows (2007, p. 15) ainda citam que o JSF é uma tecnologia que faz parte da especificação do padrão JEE, que incorpora características de uma arquitetura *Model View Controller* (MVC) para web e um modelo de interfaces gráficas baseadas em eventos. Por basear-se na arquitetura MVC, uma das melhores vantagens do JSF é a separação bem definida entre visualização e a regra de negócio.

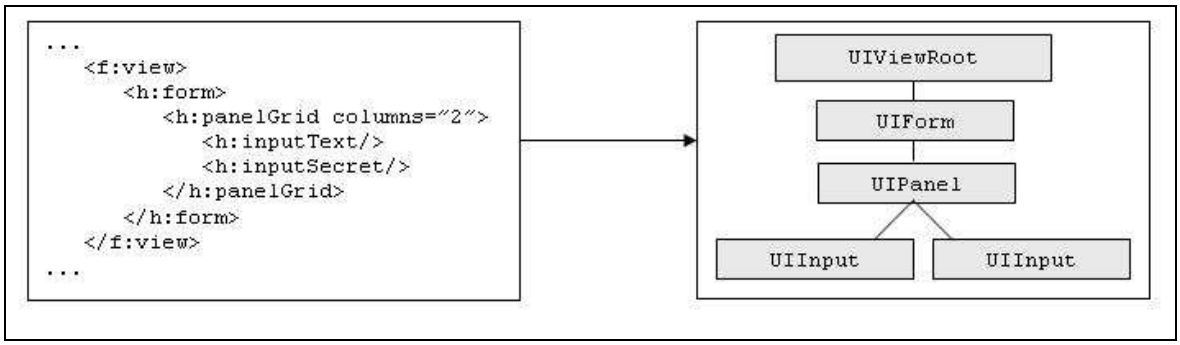
Burns e Kitain (2004, p. 47) afirmam que o JSF é uma especificação de um *framework* de componentes de interface de usuário definida pelo *Java Community Process* (JCP) e que existem várias implementações dessa especificação.

O JSF foi criado para simplificar o trabalho dos desenvolvedores web, fornecendo um caminho fácil para a criação de interfaces a partir de um conjunto de componentes

reutilizáveis. Estes componentes vêm em várias formas e com diferentes funcionalidades, variando desde botões, painéis e tabelas até componentes mais sofisticados como painéis com abas e componentes com suporte nativo a AJAX, possibilitando assim a criação de interfaces mais ricas e sofisticadas (JACOBI; FALLOWS, 2007, p. 8).

Jacobi e Fallows (2007, p. 12) dizem que no desenvolvimento da interface da aplicação o desenvolvedor deve declarar os componentes aninhando uns dentro dos outros. Esta estrutura aninhada de componentes é representada em tempo de execução através de uma hierarquia de componentes.

A Figura 5 apresenta a declaração dos componentes JSF em uma página HTML e sua representação em tempo de execução.



Fonte: Jacobi e Fallows (2007, p. 12).

Figura 5 – Componentes JSF declarados em uma página e sua representação hierárquica

Além dos componentes, o JSF também fornece um modelo de navegação declarativo, onde através de regras de navegação declaradas no arquivo `faces-config.xml` o desenvolvedor define a sequência de navegação entre as páginas (BURNS; KITAIN, 2004, p. 216). Um exemplo de regra de navegação é apresentada no Quadro 3.

```
<navigation-rule>
   <from-view-id>/login.jsp</from-view-id>
      <navigation-case>
         <from-outcome>success</from-outcome>
         <to-view-id>/result.jsp</to-view-id>
       </navigation-case>
</navigation-rule>
```

Fonte: Jacobi e Fallows (2007, p. 9).

Quadro 3 – Exemplo de regra de navegação

Segundo Pitanga (2004, p. 3), o uso do JSF no desenvolvimento de aplicações web Java oferece ganhos pois:

a) possibilita que o desenvolvedor crie interfaces por meio de um conjunto de componentes pré-definidos;
b) oferece um conjunto de *tags* para acessar os componentes;
c) oferece reutilização de componentes;

d) oferece separação de funções na construção de aplicações web.

Conforme Burns e Kitain (2004, p. 47), a especificação do JSF e a tecnologia que ela define é destinada a vários tipos de desenvolvedores, sendo eles:

a) autores de páginas: responsáveis por criar a interface de uma aplicação web. Conhecem linguagens de marcação e *script* como HTML e JavaScript assim como tecnologias de apresentação como JSP. Segundo a especificação, os autores de páginas não possuem conhecimentos em linguagens de programação para web como Java, Visual Basic e C#;
b) criadores de componentes: são responsáveis por criar bibliotecas reutilizáveis de componentes de interface que podem ser usadas pelos autores de página;
c) desenvolvedores de aplicação: responsáveis por criar no lado do servidor as funcionalidades de uma aplicação web que pode ou não estar relacionada a uma interface. São responsáveis por criar os mecanismos de persistência, classes de negócios e manipulação de eventos gerados por uma interface;
d) implementadores JSF: fornecem a implementação da especificação JSF. As implementações mais conhecidas são a Mojarra (implementação da própria SUN, também conhecida com JSF RI), MyFaces (implementação *open source* da Apache) e ADF Faces (implementação da Oracle).

2.3.1 Arquitetura JSF

O que torna o JSF diferente de outros *frameworks* como Struts e Webwork é a sua arquitetura, que permite a implementação de diferentes tecnologias de renderização, ou seja, é possível criar uma única aplicação que pode ter diferentes tipos de interfaces de saída, como HTML, *Wireless Markup Language* (WML), *Telnet* (modo caractere) e outros (JACOBI; FALLOWS, 2007, p. 9).

Para Jacobi e Fallows (2007, p. 10), por implementar o padrão *Model* 2 que é baseado no modelo MVC, uma aplicação JSF possui três elementos, que são:

a) *model* (modelo): camada constituída de classes Java simples que contém a lógica da aplicação;

b) *view* (visão): é a camada da aplicação onde se descreve o *layout* da interface e seu comportamento. A peça fundamental desta camada é o `UIComponent`, que é a base de todos os componentes do *framework*.;

c) *controller* (controlador): camada constituída por um *servlet* controlador.

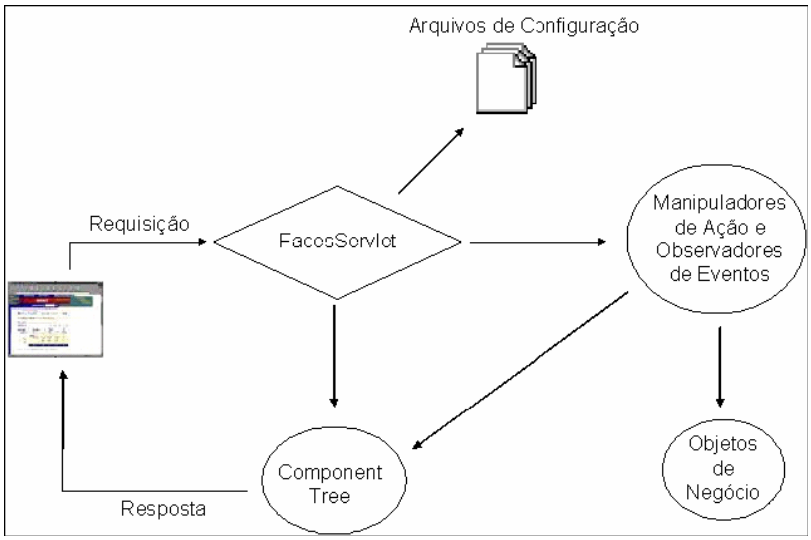
Pitanga (2004, p. 2) afirma que a camada controladora é composta pelos itens:

a) `FacesServlet`: *servlet* responsável por receber as requisições da web, redirecioná-las para a camada modelo e por fim remeter uma resposta;

b) arquivos de configuração: responsáveis por associações, mapeamento de ações e regras de navegação;

c) manipuladores de eventos: responsáveis por receber os dados vindos da camada de visão, acessar a camada modelo e devolver o resultado para o `FacesServlet`.

A Figura 6 apresenta a arquitetura JSF baseada no modelo MVC.



Fonte: Pitanga (2004, p. 2).

Figura 6 – Arquitetura JSF no modelo MVC

Segundo Jacobi e Fallows (2007, p. 13), a arquitetura interna do JSF é constituída pelos seguintes blocos:

a) `uicomponent`: é uma classe abstrata que define o comportamento dos componentes e o acesso ao modelo de dados;

b) `renderer`: é responsável pela apresentação de um componente na interface e pela conversão de uma informação vinda de um cliente em um dado legível para o componente;

c) `renderkit`: é uma coleção de renderizadores. O `RenderKit` padrão para todas as aplicações JSF é o de HTML, que contém renderizadores que produzem uma saída HTML 4.0.1.

2.3.1.1 Pacotes JSF

Segundo Burns e Kitain (2004, p. 51) a *Application Program Interface* (API) do JSF é subdividida em vários pacotes. O Quadro 4 apresenta essa subdivisão.

Pacote	Descrição
`javax.faces`	Este pacote contém as classes de nível superior da API do JSF. A classe mais importante deste pacote é a `FactoryFinder`, que é um mecanismo pelo qual os desenvolvedores podem substituir várias peças-chave da implementação.
`javax.faces.application`	Este pacote contém APIs que são usadas para vincular os objetos de regra de negócio com o JSF, bem como mecanismos para gerenciar a execução da aplicação. A classe mais importante deste pacote é a `Application`.
`javax.faces.component`	Este pacote contém as APIs fundamentais para os componentes de interface.
`javax.faces.component.html`	Este pacote contém as classes bases para cada combinação válida de componente+renderizador.
`javax.faces.context`	Este pacote contém classes e interfaces que representam a contextualização das informações associadas ao processamento de requisições recebidas e a criação das respectivas respostas. A classe mais importante é a `FacesContext`.
`javax.faces.convert`	Contém classes e interfaces definindo conversores. A classe principal é a `Converter`.
`javax.faces.lifecycle`	Contém classes e interfaces para manipulação do ciclo de vida do JSF. A classe mais importante é a `Lifecycle`.
`javax.faces.event`	Contém interfaces descrevendo eventos e manipuladores de eventos e a implementação de classes de eventos. Todos os componentes que geram com eventos estendem a classe `FacesEvent` e todos os manipuladores de evento estendem a interface `FacesListener`.
`javax.faces.render`	Contém classes e interfaces que definem o modelo de renderização. A classe principal do pacote é `RenderKit`.
`javax.faces.validator`	Contém interfaces que definem o modelo de validação e a implementação de classes de validação.
`javax.faces.webapp`	Contém classes requeridas para a integração com aplicações web, incluindo um *servlet* básico, *tags* JSP customizadas e implementação das *tags* principais do núcleo JSF.

Quadro 4 – Subdivisão da API do JSF

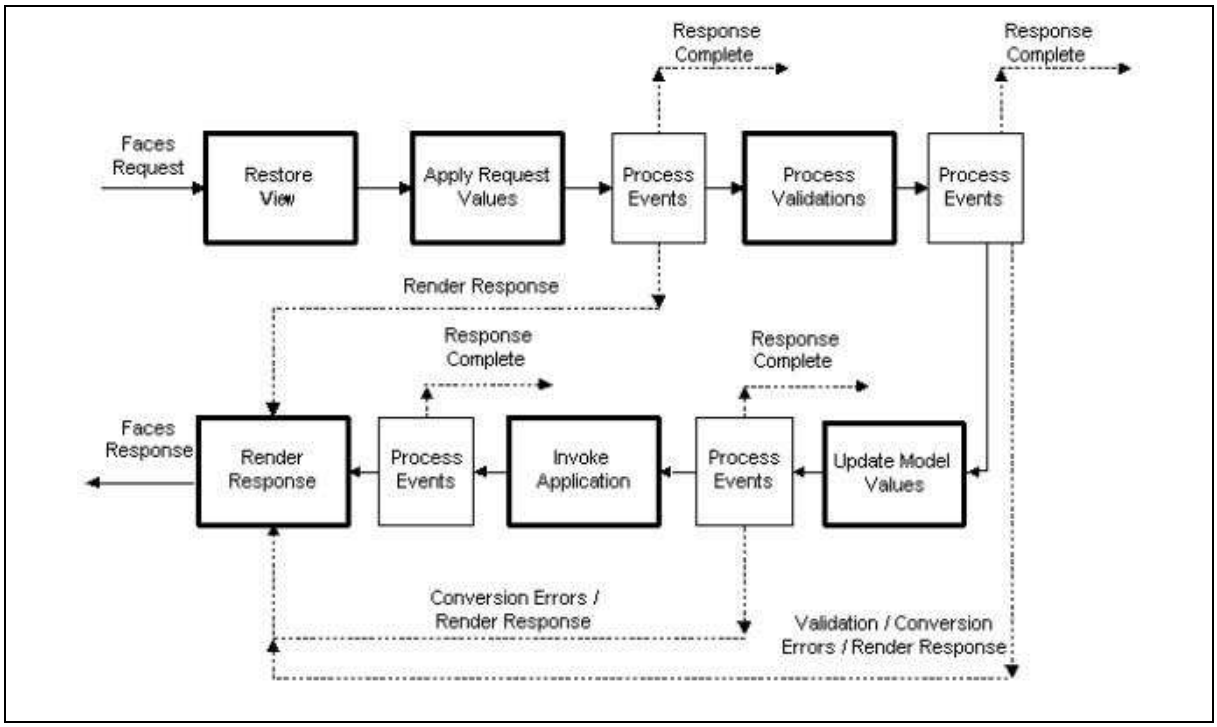
2.3.1.2 Ciclo de vida do JSF

Conforme Jacobi e Fallows (2007, p. 27), toda página construída com componentes JSF passa por um ciclo de vida de processamento de requisições bem definido.

Para Burns e Kitain (2004, p. 58), o ciclo de vida é composto por 6 fases que são:

a) *restore view*: fase responsável por restaurar a hierarquia de componentes da requisição e anexá-la ao `FacesContext`. Caso seja a primeira requisição da aplicação, esta fase é responsável por criar através do método `createView()` um novo `UIViewRoot` que é o nó raiz da hierarquia de componentes;
b) *apply request values*: nesta fase cada componente atualiza seu valor com os dados recebidos na requisição. A partir desta fase também são processados os eventos gerados pelos componentes;
c) *process validations*: esta fase é responsável por processar qualquer validador ou conversor anexado ao componente;
d) *update model values*: nesta fase todos os objetos de modelos de dados, ou *beans*, têm seus valores atualizados;
e) *invoke application*: fase responsável por processar qualquer evento disparado por um componente, que ainda não foi processado;
f) *render response*: fase responsável por renderizar a resposta da requisição para o cliente e armazenar o novo estado dos componentes para as próximas requisições.

A Figura 7 apresenta a sequência do ciclo de vida do JSF.



Fonte: Burns e Kitain (2004, p. 57).

Figura 7 – Ciclo de vida do JSF

2.3.2 Exemplo JSF

O exemplo a seguir apresenta uma simples página HTML contendo componentes JSF. A página possui dois campos, um para informar o *login*, outro para informar a senha e um botão que ao ser pressionado apresenta uma mensagem de alerta apresentando os dados digitados.

Burns e Kitain (2004, p. 247) relatam que um componente pode ser colocado em qualquer posição da página e que é necessário declarar através de diretivas *taglibs* as bibliotecas relativas ao componente. Por padrão a página deve conter a declaração das bibliotecas referentes aos componentes do núcleo do JSF e os componentes HTML básicos. Normalmente essa declaração é feita no topo da página e deve conter o *Uniform Resource Identifier* (URI) que é o identificador de localização da biblioteca, e o prefixo, usado para declarar um componente na página.

O Quadro 5 apresenta a declaração das bibliotecas através das diretivas *taglibs*.

```
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
```

Quadro 5 – Declaração das diretivas *taglibs*

Burns e Kitain (2004, p. 297) ainda ensinam que todos os componentes devem ser declarados dentro da *tag* `<f:view>` que é o *container* dos componentes.

O Quadro 6 apresenta o código completo da página exemplo e a Figura 8 apresenta o resultado da sua execução.

```
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>

<html>
    <head>
        <title>Exemplo JSF</title>
    </head>
    <body>
        <f:view>
              <center>
                 <h1>Informe Login e Senha </h1>
                 <h:outputText value="Login:" />
                 <h:inputText id="login"/> <br>
                 <h:outputText value="Senha:" />
                 <h:inputSecret id="senha"/>  <br>
                 <h:commandButton value="Acessar" onclick="alert('Login:' +
document.getElementById('login').value +
'\nSenha:' + document.getElementById('senha').value )"/>
              </center>
        </f:view>
    </body>
</html>
```

Quadro 6 – Código da página exemplo

Figura 8 – Resultado da execução da página

2.3.3 Componentes JSF customizados

A seguir é feita uma breve explanação sobre a criação de componentes customizados.

O modelo de componentes e a arquitetura extensível do JSF permitem ao desenvolvedor criar novos componentes quando necessário (JACOBI; FALLOWS, 2007, p. 49).

Conforme Jacobi e Fallows (2007, p. 53), a criação de um componente envolve as seguintes etapas:

a) definição da classe do componente: neste ponto deve-se criar a classe Java que representa o componente. A classe deve estender a classe base `UIComponentBase` e deve possuir a declaração de todos os atributos do componente bem como os métodos *getters* e *setters*. Deve implementar o método `getFamily()` para retornar o identificador da família do componente;

b) definição da classe renderizadora: definir a classe que contém o código que gera o *markup* para renderizar o componente na tela. Essa classe deve estender a classe base `Renderer`;

c) definição da classe de *tag*: definir a classe que irá fazer o processamento de *tags* do componente. A classe deve estender a classe base `UIComponentElTag` e deve implementar os métodos `getRenderType()` e `getComponentType()`;

d) definição do arquivo *Tag Library Descriptor* (TLD): nesse arquivo é feita toda a descrição da *tag* que representa o componente ou o conjunto de componentes no caso da criação de uma biblioteca de componentes. Deve-se especificar a versão da biblioteca de *tags*, o apelido do componente e a URI para utilização na declaração da diretiva *taglib*. O Quadro 7 apresenta um trecho da estrutura de um arquivo TLD;

e) definição do arquivo `faces-config.xml`: este arquivo deve conter os mapeamentos para o tipo do componente e sua classe através da *tag* `component` e das *tags* filhas `component-type` e `component-class`, e para o tipo de renderizador do componente através das *tags* `render-kit` e `renderer` e as *tags* filhas `component-family`, `renderer-type` e `renderer-class`. O valor definido nas *tags* filhas deve ser o mesmo valor retornado pelos métodos `getFamily()`, `getRendererType()` e `getComponentType()` citados anteriormente. O Quadro 8 apresenta um trecho da estrutura de um arquivo `faces-config.xml`.

```
...
<display-name> </display-name>
<tlib-version> </tlib-version>&gt;
<jsp-version> </jsp-version>
<short-name> </short-name>
<uri> </uri>

<tag>
<name> </name>
<tag-class> </tag-class>
<body-content>JSP</body-content>

<attribute>
  <name> </name>
  <deferred-value>
     <type> </type>
  </deferred-value>
</attribute>
...
```

Quadro 7 – Trecho do arquivo .TLD

```
<component>
    <component-type> </component-type>
    <component-class> </component-class>
</component>

<render-kit>
  <renderer>
    <component-family> </component-family>
    <renderer-ype> </renderer-type>
    <renderer class> </renderer-class>
  </renderer>
</render-kit>
```

Quadro 8 – Trecho do arquivo `faces-config.xml`

2.4 JSON

JSON é um formato para intercâmbio de dados de fácil leitura e escrita. Baseado em um subconjunto da linguagem JavaScript, JSON é independente de linguagem por usar convenções familiares às linguagens C, C++, C#, Java, Perl, Python e outras (JSON, 2006).

Segundo JSON (2006), o formato é constituído de duas estruturas de dados universais, que são:

a) coleção de pares nome/valor, semelhante a estruturas do tipo `object, record e hash table;`
b) uma lista ordenada de valores, semelhante a estruturas do tipo `array`, lista ou vetor.

Para Jaconete (2006), através do JSON é possível a representação de objetos complexos. Ainda segundo Jaconete (2006), JSON é bastante utilizado na execução de chamadas AJAX, onde a programação do lado servidor gera `strings` no formato JSON possibilitando através da função `eval()` que o JavaScript crie um objeto que represente os dados recebidos.

O Quadro 9 apresenta um trecho de código JavaScript que define uma `string` representando um objeto Pessoa no formato JSON e sua conversão para objeto JavaScript. O objeto possui apenas 2 atributos: nome, idade e seus respectivos valores.

```
{
 var stringJSON = '({"Pessoa":{'nome':'Leonardo','idade':'26'}})';
 var novoObjeto = eval(stringJSON);
 alert(novoObjeto.nome);
}
```

Quadro 9 – Criação de um objeto JavaScript a partir de uma `string` no formato JSON

2.4.1 FlexJSON

FlexJSON é uma pequena biblioteca para serialização de objetos Java em JSON. De fácil utilização, a biblioteca permite a serialização desde simples objetos Java até objetos mais complexos que contenham coleções ou classes associadas (HUBBARD, 2007). Hubbard (2007) ainda cita que a principal classe da biblioteca é a `JSONSerializer` e que os principais métodos da classe são:

a) `serialize`: realiza a serialização superficial da classe, analisando apenas os atributos simples e deixando de lado atributos mais complexos como coleções, mapas e outros objetos. Para cada atributo encontrado é executado seu respectivo método *get* para recuperar seu valor;

b) `deepSerialize`: realiza a serialização em profundidade, ou seja, além de analisar os atributos simples, analisa coleções, mapas e também as classes associadas a classe principal. Para cada classe encontrada é feita uma nova análise e assim sucessivamente.

O Quadro 10 apresenta a definição de uma classe Java simples e o uso da biblioteca FlexJSON.

```
public class Pessoa
{
   private String nome;
   private int idade;
   private String apelido;

   public String getApelido()
   {
       return “Léo”;
   }

   public String getNome()
   {
       return “Leonardo”;
   }

   public int getIdade()
   {
      return 26;
   }
}
public String convertToJSON()
{
    Pessoa p = new Pessoa();
    JSONSerializer jSerializer = new JSONSerializer();
    return  jSerializer.serialize(p);
}
```

Quadro 10 – Definição de uma classe Java e uso da biblioteca FlexJSON

O Quadro 11 apresenta o resultado da execução do método `serialize`.

```
‘{ “class”: “Pessoa”,
    “nome”: “Leonardo”,
    “apelido” : “Léo”
}’
```

Quadro 11 – Resultado da execução do método `serialize`

2.5 ANÁLISE LÉXICA

A análise léxica é a fase do compilador que lê um programa fonte e o divide em marcas conhecidas como *tokens*[5].

"Compiladores são programas de computador que traduzem de uma linguagem para outra" (LOUDEN, 2004, p. 1). Para Louden (2004, p. 1), um compilador transforma um programa escrito em uma linguagem de alto nível como C ou Java, em um programa escrito com instruções de máquina.

Conforme Grune et al (2001, p. 88), o analisador léxico tem a tarefa de localizar os *tokens*, identificar sua classe e enviá-los para o analisador sintático. Para Delamaro (2006, p. 4), o processo de análise léxica apenas identifica os símbolos de um programa fonte, não se preocupando em verificar se a ordem desses símbolos está correta.

Lopes (2007, p. 1), afirma que o analisador léxico é o componente do compilador responsável por ler e interpretar as expressões do código fonte de uma determinada linguagem, reconhecendo respectivos *tokens* e atributos quando necessário. Ainda, afirma que as expressões reconhecidas devem ser formalmente definidas por expressões regulares que por sua vez, são reconhecidas pelo seu respectivo autômato determinístico. Conforme Leyendecker (2005, p. 27), as expressões regulares fornecem a principal descrição formal para a análise léxica, sendo uma fórmula que descreve um conjunto de palavras. A especificação léxica deve permitir a separação de cadeias de caracteres em classes de símbolos, permitindo assim utilizar estas classes de símbolos ao invés de apenas um símbolos (LEYENDECKER, 2005, p. 27).

Para Lopes (2007, p. 1), um analisador léxico deve ser capaz de:

a) reconhecer os *tokens;*
b) possuir uma tabela de símbolos com rotinas relacionadas ao armazenamento e recuperação de lexemas;
c) tratar as palavras reservadas da linguagem;
d) possuir gerenciamento de erros ocorridos;
e) dispor um buffer de entrada para o código.

[5] São palavras reservadas, identificadores e símbolos colhidos na análise léxica (DELAMARO, 2006, p. 6).

2.6 TRABALHOS CORRELATOS

Nesta seção são apresentadas duas ferramentas que desempenham um papel semelhante ao trabalho proposto. Na seção 2.6.1 é apresentado o ambiente de desenvolvimento Eclipse. Na seção 2.6.2 é apresentada a ferramenta CodePress.

2.6.1 Eclipse

Segundo Bystronski (2004, p. 14), o Eclipse é uma plataforma desenvolvida em Java que foi projetada para a construção de aplicações web. Ainda cita que a plataforma por si só não possui muitas funcionalidades. Sua característica é fornecer pontos de extensão para a implementação de *plugins* que agregam novas funcionalidades a ela.

Para Proulx (2005), a maneira mais fácil para se criar um *plugin* é através do assistente de criação de *plugins*, ferramenta iniciada quando um novo projeto de *plugin* é criado. A Figura 9 apresenta a tela inicial deste assistente e a Figura 10 apresenta a tela onde se define os dados básicos do *plugin*.

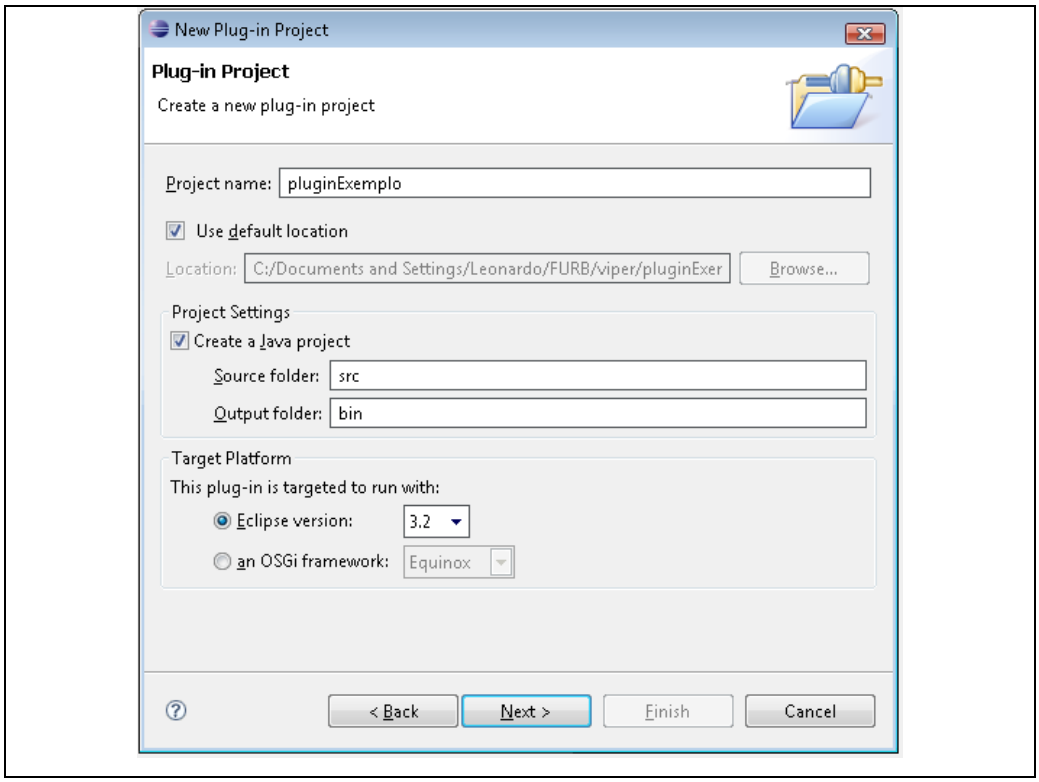


Figura 9 – Tela inicial do assistente de criação de *plugins*

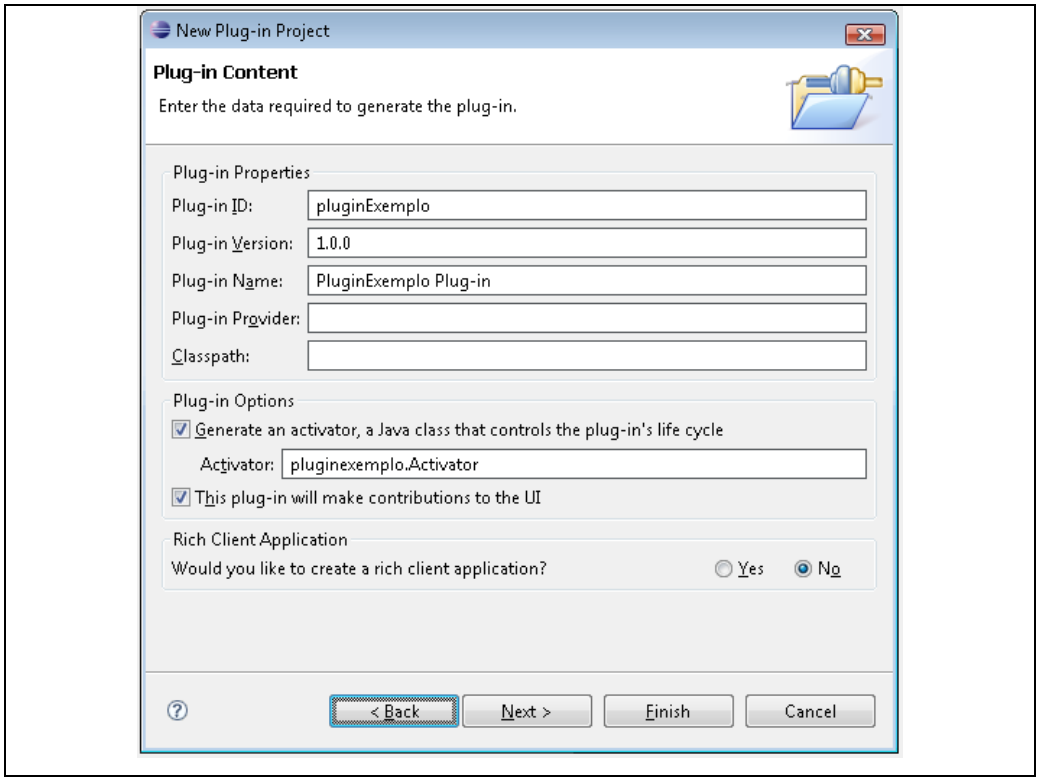


Figura 10 – Definição dos dados básicos do *plugin*

Após preencher os dados básicos, o assistente apresenta uma tela com opções de *templates* para a criação do *plugin*. A Figura 11 apresenta esta tela.

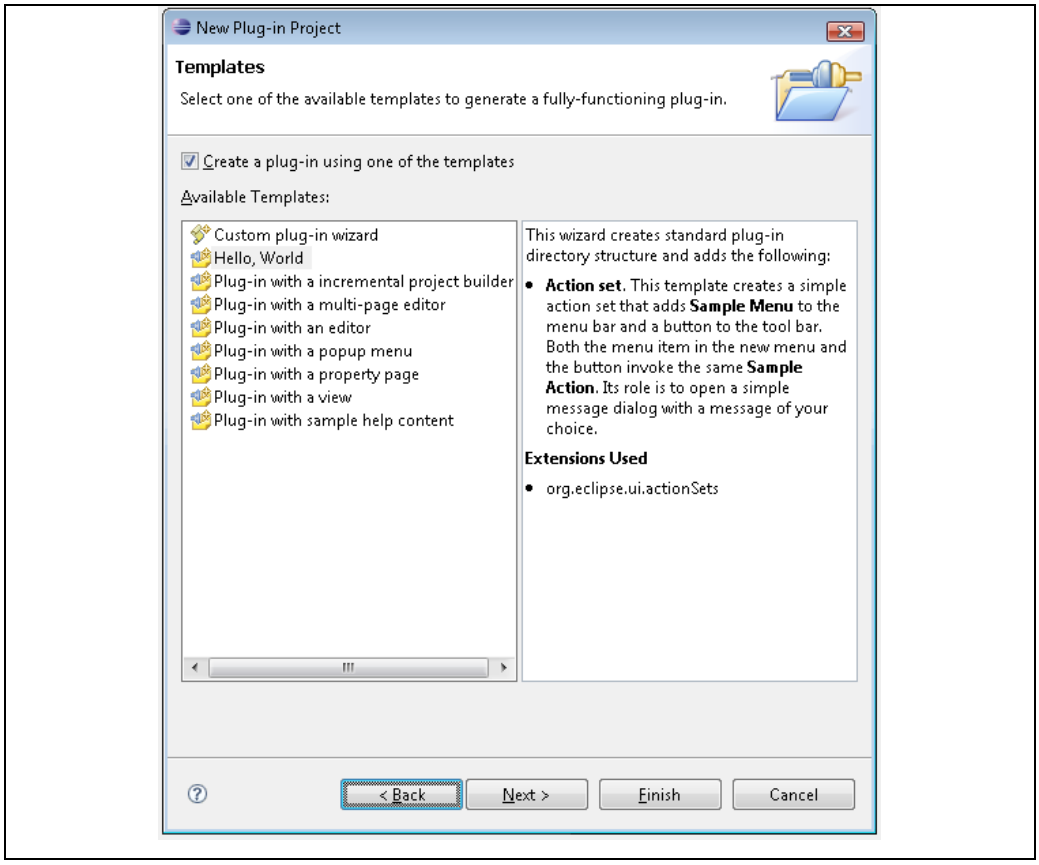


Figura 11 – Opções de *templates* para criação do *plugin*

Conforme Proulx (2005), ao finalizar o assistente são criados os seguintes arquivos:

a) `plugin.xml`: arquivo que descreve o *plugin*. Contém informações sobre bibliotecas, dependências, modelo de interconexões e geração de código. Conforme Bystronski (2004, p. 15), o modelo de interconexões é composto pelas extensões que o *plugin* oferece e as extensões que o *plugin* faz com outros *plugins* existentes na plataforma;

b) `build.properties`: arquivo usado para descrever o processo de *build* do *plugin*;

c) classe do *plugin*: classe Java da implementação do *plugin*;

d) classe de ações: classe Java implementado as ações do *plugin.*

Após a geração dos arquivos a plataforma Eclipse apresenta um ambiente para realizar a manutenção do *plugin* criado. Através deste ambiente é possível alterar todas as configurações do *plugin*, dependências, propriedades e extensões. A Figura 12 apresenta este ambiente.

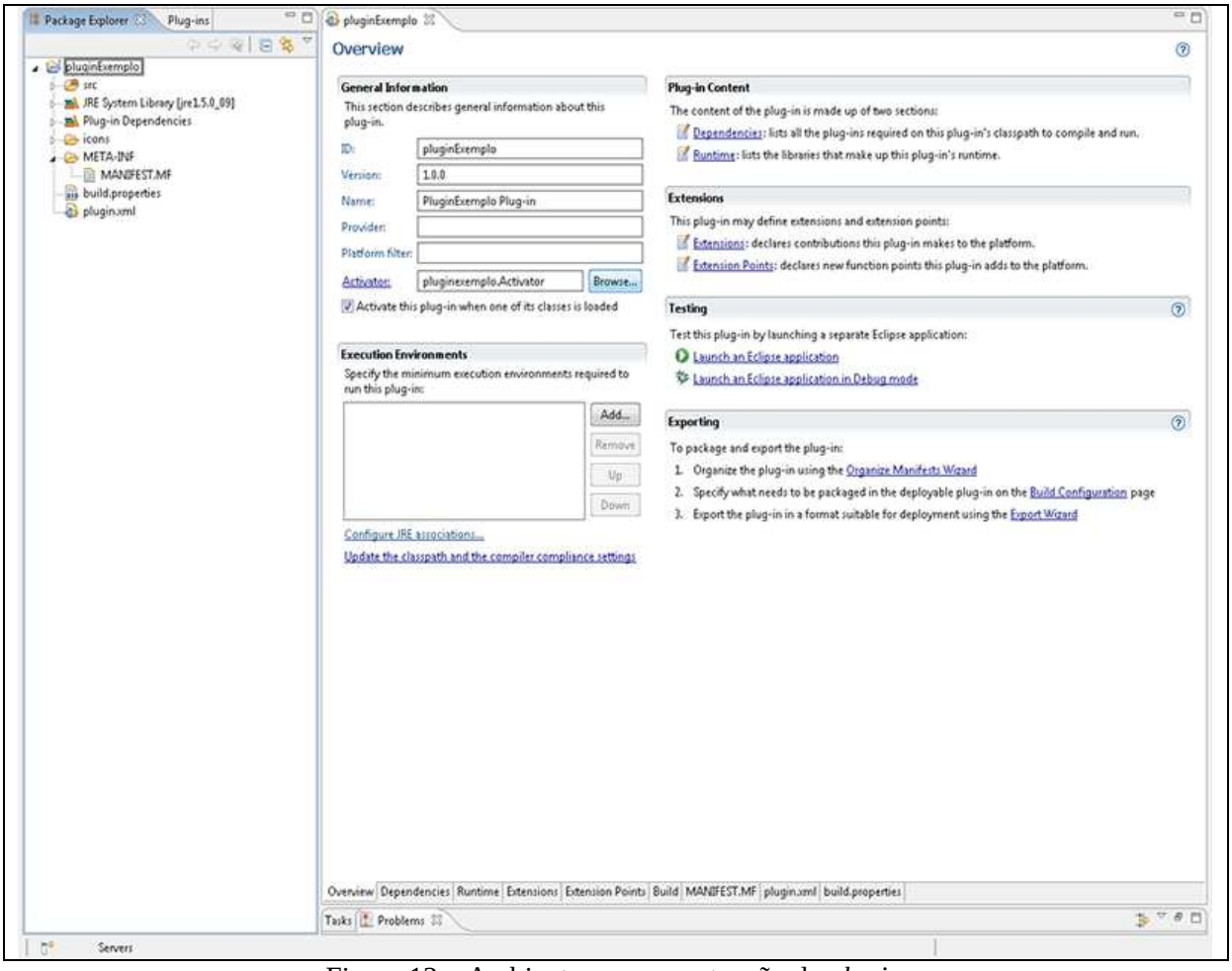


Figura 12 – Ambiente para manutenção de *plugins*

Segundo Oliveira (2005), a tecnologia de *plugins* permite que o ambiente de desenvolvimento possa ser personalizado, adequando as funcionalidades de acordo com o tipo

de projeto que está sendo criado, ou seja, se o desenvolvedor necessita criar um projeto para web usando a linguagem PHP por exemplo, ele adiciona *plugins* que irão configurar o ambiente e personalizar a área de edição de código e as opções de trabalho para a linguagem PHP.

O Eclipse é uma das ferramentas mais populares para o desenvolvimento em plataforma Java e é uma das grandes ferramentas da iniciativa *Open-Source*.

2.6.2 CodePress

Foi desenvolvido com a linguagem JavaScript e é distribuído sob a licença *Lesser General Public Licence* (LGPL) (CODEPRESS, 2007).

O editor disponibiliza funcionalidades de *syntax highlighting*, numeração de linhas, múltiplas janelas na mesma página e atalhos. Reconhece as linguagens PHP, JavaScript, Java, Perl, SQL, HTML e CSS mas não disponibiliza um meio para adicionar novas linguagens.

As linguagens são reconhecidas através de bibliotecas JavaScript e arquivos CSS definidos para cada linguagem e que são carregados quando o editor é executado pela primeira vez. Para definir o editor em uma página HTML o usuário deve declarar a biblioteca JavaScript `CodePress.js` e fazer a chamada para a classe `CodePress` especificando qual a linguagem deve ser reconhecida (CODEPRESS, 2007).

O Quadro 12 apresenta o código HTML e JavaScript que cria o editor na página. A Figura 13 apresenta o editor CodePress reconhecendo a linguagem PHP.

```
<head>
   <link type="text/css" href="languages/codepress-php.css"
rel="stylesheet" id="cp-lang-style" />
<script type="text/javascript" src="codepress.js"></script>
<script type="text/javascript">
   CodePress.language = 'PHP';
</script>
</head>
```

Quadro 12 – Código para criação do editor CodePress

```
<?php
// Very simple implementation of server side script

if(isset($_GET['file'])) {
        $file = basename($_GET['file']);
        $full_file = $path['server'].'/'.$path['webdocs'].'/'.$path['files']."/".$
        if(file_exists($full_file)) {
                $code = file_get_contents($full_file);
                $code = preg_replace("/>/","&gt;",$code);
                $code = preg_replace("/</","&lt;",$code);
                $language = getLanguage($file);
        }
}
?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xht
<html>
<head>
        <title>CodePress - Real Time Syntax Highlighting Editor written in JavaScr
        <link type="text/css" href="languages/codepress-<?=$language?>.css" rel="s
```

Figura 13 – Editor CodePress reconhecendo a linguagem PHP

3 DESENVOLVIMENTO

Neste capítulo são apresentados os requisitos, a especificação e implementação do trabalho e os diagramas de casos de uso, classes e sequência. Também são apresentados um estudo de caso para validação do trabalho e uma comparação com os trabalhos correlatos.

3.1 REQUISITOS

Os Requisitos Funcionais (RF) e os Requisitos Não Funcionais (RNF) do componente são:

a) disponibilizar meios para abrir e salvar o código digitado (RF);
b) disponibilizar mecanismo para definir e anexar *plugins* (RF);
c) disponibilizar recursos de numeração de linhas, auto complemento e *syntax highlighting* do código (RF);
d) reconhecer nativamente a linguagem Java (RF);
e) ser implementado utilizando o ambiente NetBeans 6.1 com as linguagens Java e JavaScript e o *framework* JSF (RNF);
f) ser compatível com os navegadores Internet Explorer e Mozilla Firefox (RNF).

3.2 ESPECIFICAÇÃO

Para o desenvolvedor o componente é uma simples *tag* que é inserida em uma página, ficando implícito para ele o conjunto de tecnologias que trabalham juntas para interpretá-la e gerar um código HTML que representa o componente.

Para especificar a estrutura do componente foram feitos com o auxílio da ferramenta Enterprise Architect os diagramas de casos de uso, classes e sequência, pertencentes a linguagem de modelagem *Unified Modeling Language* (UML).

3.2.1 Diagrama de casos de uso

Segundo Fowler (2005, p. 104), os casos de uso são uma técnica para captar os requisitos funcionais de um sistema, servindo para descrever as interações típicas entre os usuários de um sistema e o próprio sistema.

Conforme Bezerra (2002, p. 45), o modelo de casos de uso é importante, pois força os desenvolvedores a modelar o sistema de acordo com as necessidades do usuário e não o contrário, modelar o sistema de uma maneira que o usuário tenha que se adaptar.

A Figura 14 apresenta o diagrama de casos de uso do componente.

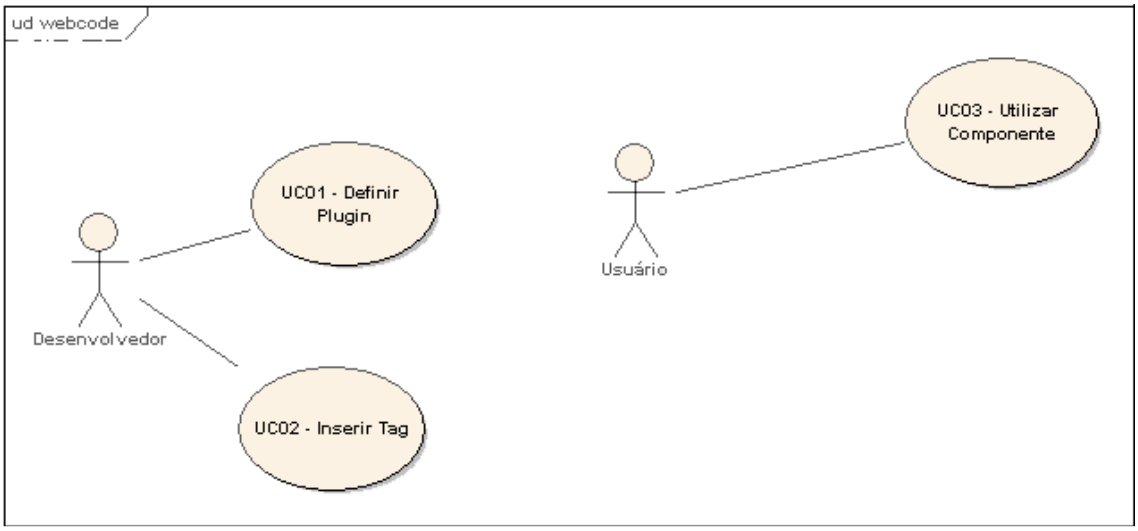


Figura 14 – Diagrama de casos de uso

O Quadro 13 apresenta o cenário para o caso de uso definir *plugin.*

UC01 – Definir Plugin

1. Descrição: Desenvolvedor define plugin para linguagem de programação desejada.

1.1 Atores: Desenvolvedor (Principal).

1.2 Pré-condições: Bibliotecas do JSF e do webcode inseridas no projeto.

1.3 Fluxo de eventos:

1.3.1 Fluxo básico: Definir plugin.
a)Desenvolvedor cria classe que implementa a interface WebCodePlugin
b)Desenvolvedor implementa método getPluginName().
c)Desenvolvedor implementa método getPluginVersion().
d)Desenvolvedor implementa método getWebCodeSyntaxDefinations().
e)Desenvolvedor implementa método getWebCodeComplete().

1.4 Pós-condição: Plugin definido.

Quadro 13 – Cenário do caso de uso definir *plugin*

O Quadro 14 apresenta o cenário para o caso de uso inserir *tag*.

UC02 – Inserir Tag

1. Descrição: Desenvolvedor insere tag do componente na página.

1.1 Atores: Desenvolvedor (Principal).

1.2 Pré-condições: Plugin definido.

1.3 Fluxo de eventos:

1.3.1 Fluxo básico: Inserir Tag.
a)Desenvolvedor define na página as bibliotecas de tags do JSF e do componente.
b)Desenvolvedor insere tag webcode na página.
c)Desenvolvedor define atributos da tag.
d)Desenvolvedor define funções JavaScript para abrir e salvar um código.
e)Desenvolvedor executa projeto.

1.4 Pós-condição: Editor criado no navegador.

Quadro 14 – Cenário do caso de uso inserir *tag*

O Quadro 15 apresenta o cenário para o caso de uso utilizar componente.

UC03 – Utilizar Componente

1. Descrição: Usuário utiliza componente.

1.1 Atores: Usuário (Principal).
Componente (Secundário)

1.2 Pré-condições: Tag inserida.

1.3 Fluxo de eventos:

1.3.1 Fluxo básico: Utilizar componente.
a)Usuário cria novo código.
b)Usuário digita código.
c)Componente executa função de syntax highlighting.

1.3.2 Fluxos Alternativos:

1.3.2.1 Criar código.
a)No item a do fluxo básico usuário clica no botão novo código.
b)Componente limpa área de edição para que um novo código seja digitado.

1.3.2.2 Abrir código
a)No item b do fluxo básico usuário clica no botão abrir código.
b)Componente executa função JavaScript para carregar o código.
c)Componente coloca o código na área de edição.

1.3.2.3 Salvar código
a)No item b do fluxo básico usuário clica no botão salvar código.
b)Componente executa função JavaScript para salvar o código.

1.3.2.4 Auto complemento
a)No item b do fluxo básico usuário pressiona simultaneamente as teclas [ctrl] e [espaço].
b)Componente solicita dados ao servidor via requisição AJAX.
c)Componente apresenta lista com dados na tela.
d)Usuário seleciona item.
e)Componente insere no código o item selecionado.

1.4 Pós-condição: Código digitado.

Quadro 15 – Cenário do caso de uso utilizar componente

3.2.2 Diagrama de pacotes e classes

Para Fowler (2005, p. 96), um pacote é uma construção que permite ao desenvolvedor agrupar qualquer construção UML em um nível mais alto. Seu uso mais comum é para o agrupamento de classes.

O componente desenvolvido é dividido em três módulos: módulo JSF, módulo JavaScript e módulo de execução. A Figura 15 apresenta o diagrama de pacotes do componente.

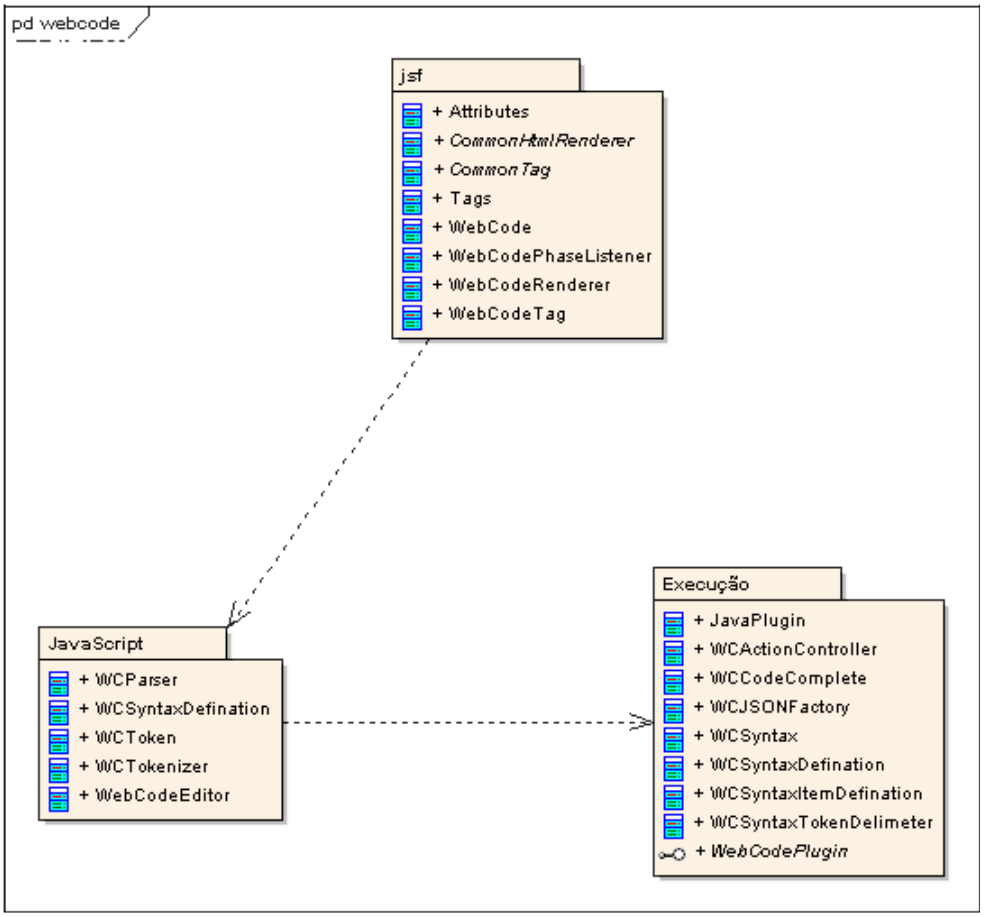


Figura 15 – Diagrama de pacotes

Segundo Fowler (2005, p. 52), um diagrama de classes descreve os tipos de objetos presentes no sistema, os vários tipos de relacionamentos existentes entre eles, suas propriedades e operações.

3.2.2.1 Módulo JSF

O módulo JSF implementa todos os requisitos para a criação de um componente JSF. É composto por classes responsáveis pela geração do código HTML para o navegador, inclusão de bibliotecas JavaScript, tratamento das requisições AJAX e processamento de *tags*. As classes estão distribuídas entre os pacotes `webcode.jsf.component`, `webcode.jsf.commons.tags`, `webcode.jsf.commons.renderer.html`.

A Figura 16 apresenta as classes que compõem os pacotes citados.

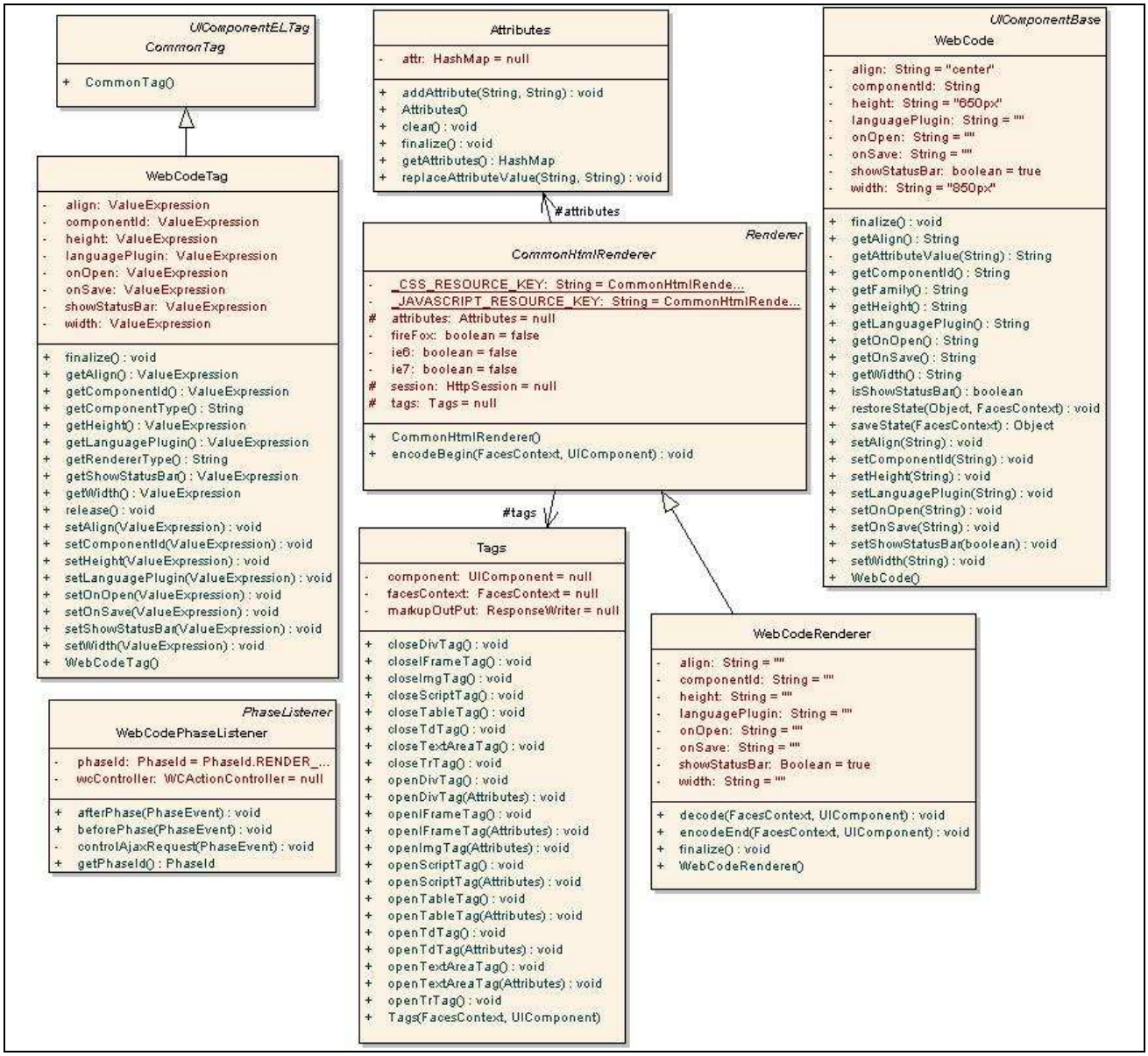


Figura 16 – Classes JSF

O Quadro 16 descreve cada classe e suas responsabilidades.

Classe	Descrição
`Attributes`	Classe base responsável pela manipulação dos atributos de uma *tag* HTML.
`CommonHtmlRenderer`	Classe de suporte para renderização do componente. Possui métodos para escrita de recursos como JavaScript e CSS, inserção de imagens e textos. Estende a classe base `Renderer`.
`CommonTag`	Classe de suporte para o processamento de *tags*. Possui métodos comuns para a manipulação de *tags* e estende a classe `UIComponentTag`, que é a classe base para todas as *tags* que representam os componentes JSF em uma página.
`Tags`	Classe base responsável por gerar o código das *tags* HTML.
`WebCode`	Classe que representa o componente. Estende a classe base `UIComponentBase`.
`WebCodePhaseListener`	Classe responsável pelo tratamento das requisições AJAX. Estende a classe `PhaseListener`.
`WebCodeRenderer`	Classe responsável pela renderização do componente. Utiliza as classes `Attributes` e `Tags` para geração do código HTML. Estende a classe `CommonHtmlRenderer`.
`WebCodeTag`	Classe responsável pelo processamento da *tag* que representa o componente. Estende a classe `CommonTag`.

Quadro 16 – Descrição das classes

3.2.2.2 Módulo JavaScript

O módulo JavaScript é composto pelas classes responsáveis por executar funções do componente no navegador, realizar as requisições AJAX e pelo recurso de *syntax highlighting*.

A Figura 17 apresenta estas classes e o Quadro 17 descreve as classes e suas responsabilidades.

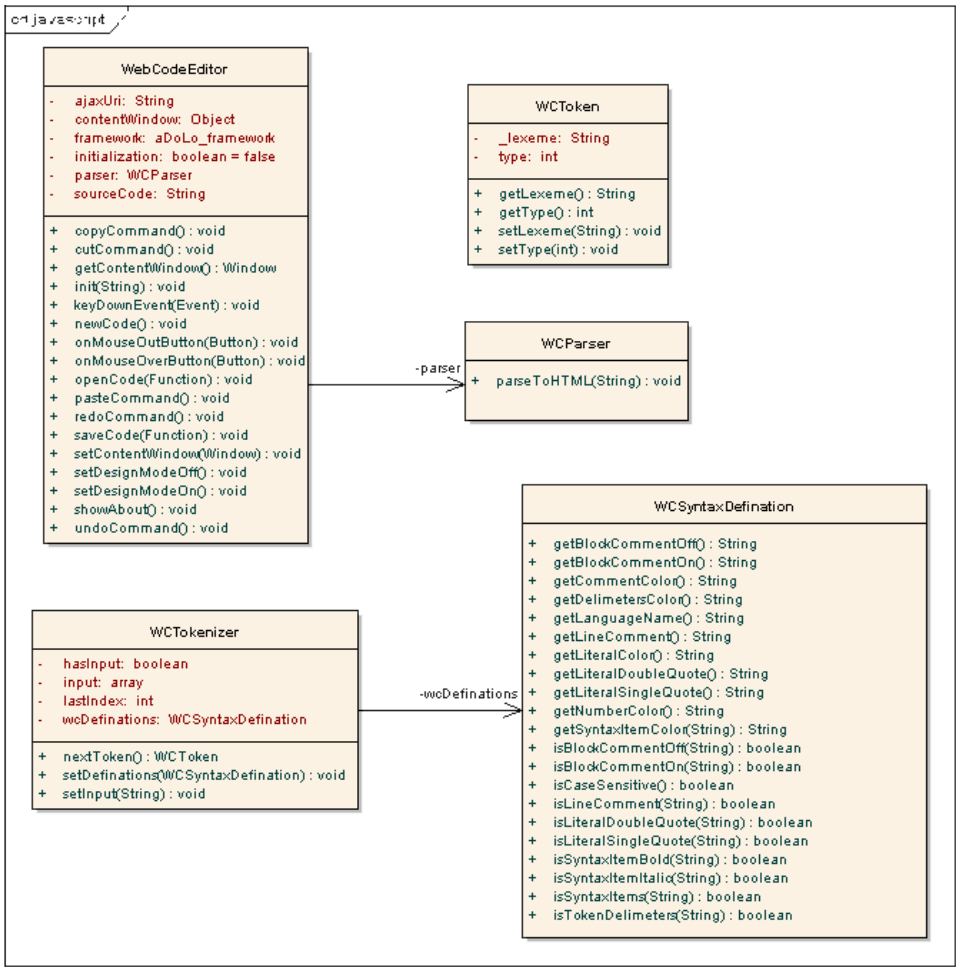


Figura 17 – Classes JavaScript

Classe	Descrição
`WCParser`	Classe principal do recurso de *syntax highlighting*. Responsável por fazer o *parser* do código e aplicar a formatação HTML nos *tokens*.
`WCSyntaxDefination`	Classe que contém as definições da linguagem e definições de formatação.
`WCToken`	Classe que representa um *token*.
`WCTokenizer`	Classe responsável por retornar os *tokens* do código através da análise léxica.
`WebCodeEditor`	Classe responsável pela inicialização das funções do componente do navegador, tratamento de eventos `keydown` e pelos recursos de copiar, colar, recortar, refazer e desfazer.

Quadro 17 – Descrição das classes

3.2.2.3 Módulo de execução

O módulo de execução engloba as classes responsáveis por executar as ações solicitadas pelo componente, a classe de geração de conteúdo JSON, as classes auxiliares para

criação dos *plugins* e a interface necessária para a implementação dos *plugins*. As classes estão distribuídas entre os pacotes `webcode`, `webcode.plugin`, `webcode.json`, `webcode.syntax` e `webcode.syntax.code`.

A Figura 18 apresenta estas classes e o Quadro 18 apresenta a descrição de cada classe.

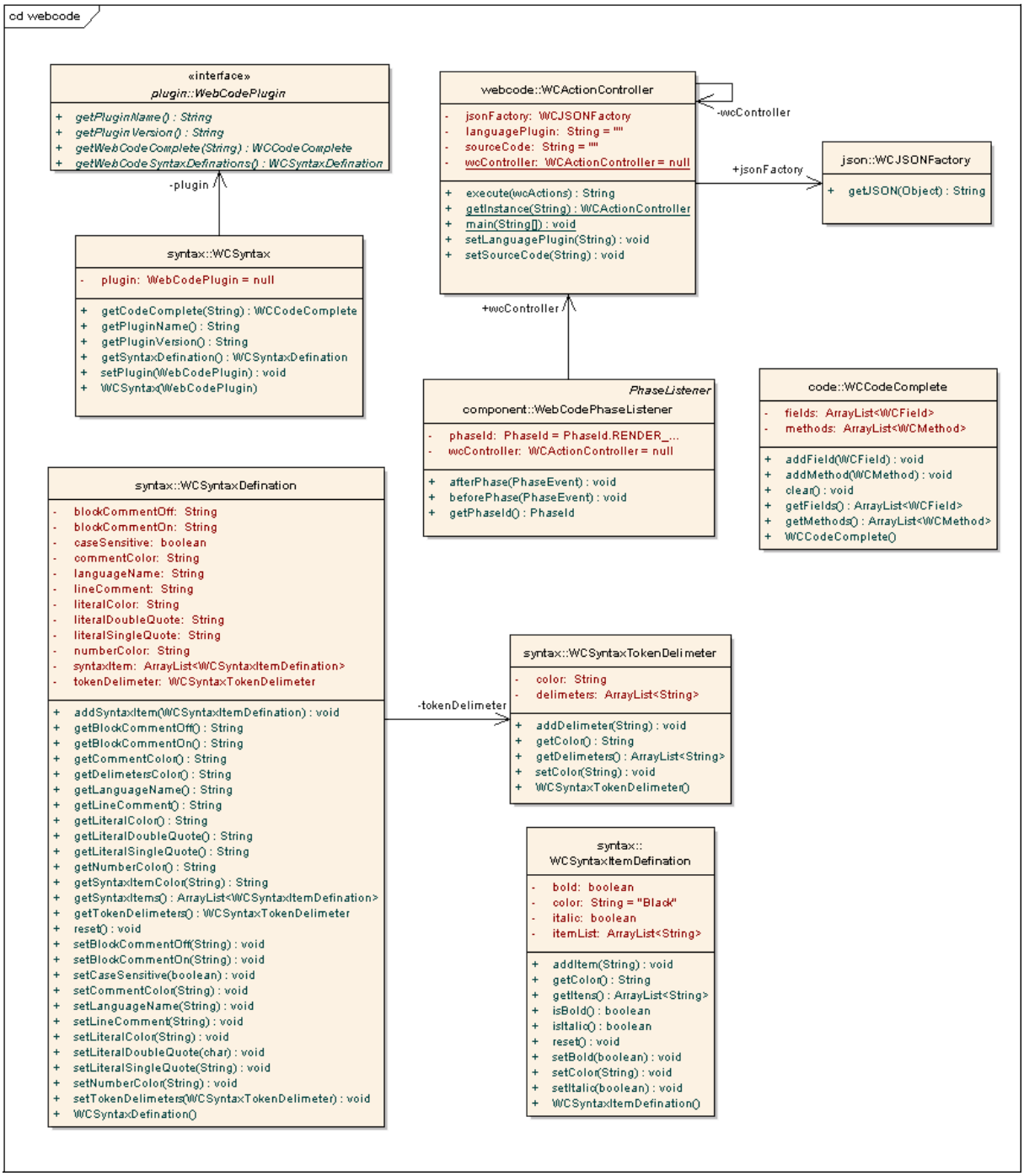


Figura 18 – Classes de execução

Classe	Descrição
`WCActionController`	Classe responsável por receber e executar as ações que chegam por requisições AJAX através da classe `WebCodePhaseListener`.
`WCCodeComplete`	Classe usada pelo *plugin* para o recurso de auto complemento de código.
`WCJSONFactory`	Classe responsável por gerar conteúdo JSON para ser enviado ao componente.
`WCSyntaxDefination`	Classe usada para implementação de um *plugin*. Mantém em sua estrutura um objeto `WCSyntaxTokenDelimeter` e uma coleção de objetos `WCSyntaxItemDefination`.
`WCSyntaxItemDefination`	Representa os itens de uma linguagem, como palavras reservadas e classes pré definidas.
`WCSyntaxTokenDelimeter`	Representa os tokens delimitadores de uma linguagem.
`WebCodePlugin`	Interface que define as operações que um *plugin* deve implementar.

Quadro 18 – Descrição das classes

3.2.3 Diagramas de sequência

Os diagramas de interação descrevem como grupos de objetos colaboram em algum comportamento. A UML define várias formas de diagramas de interação, sendo o de sequência o mais usado (FOWLER, 2005, p. 67). Para Fowler (2005, p. 67), um diagrama de sequência captura o comportamento de um cenário demonstrando a troca de mensagens entre os objetos que o compõe.

A Figura 19 apresenta o diagrama de sequência que demonstra os passos realizados pelo componente para realizar a função de *syntax highlighting*.

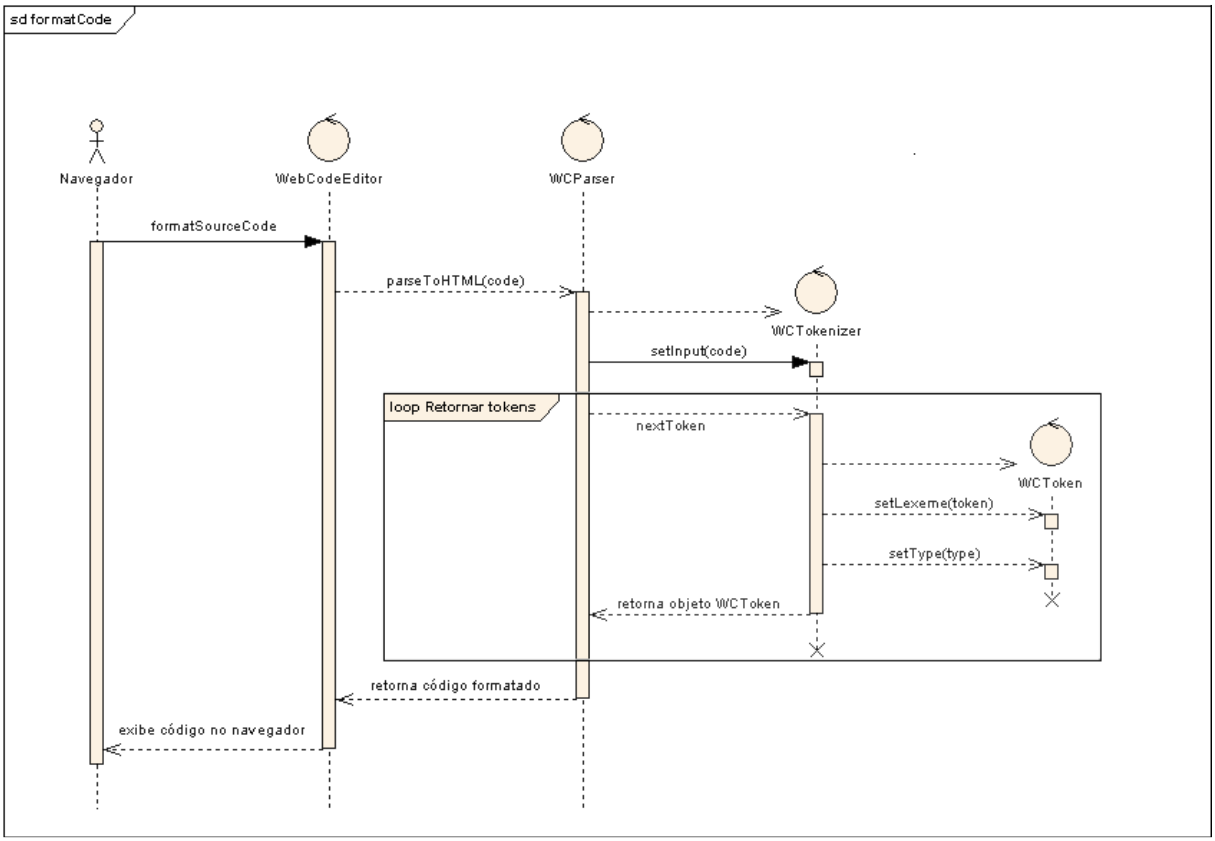


Figura 19 – Diagrama de sequência para recurso de *syntax highlighting*

A cada intervalo de três segundos o navegador executa o método `formatSourceCode()` do objeto `WebCodeEditor`. Este método recupera o código digitado pelo usuário e o envia para o objeto `WCParser` através do método `parseToHTML()`.

Ao ser executado o método `parseToHTML()` cria uma instância da classe `WCTokenizer` e através do método `setInput()` informa a ela o código que deve ser analisado.

A partir desse ponto a execução entra em um laço onde o método `parseToHTML()` realiza sucessivas chamadas ao método `nextToken()` do objeto `WCTokenizer` para que o mesmo crie e retorne um objeto `WCToken` contendo o próximo *token* do código e o seu tipo. Após isso o *token* é formatado conforme o seu tipo e concatenado com os demais *tokens* já analisados. A execução do laço termina quando todos os *tokens* forem lidos e formatados.

O resultado da execução do método `parseToHTML()` é o código formatado para ser exibido no navegador através do objeto `WebCodeEditor`.

O diagrama de sequência apresentado na Figura 20 demonstra a sequência de passos realizados pelo componente quando o usuário dispara a função de auto complemento do código.

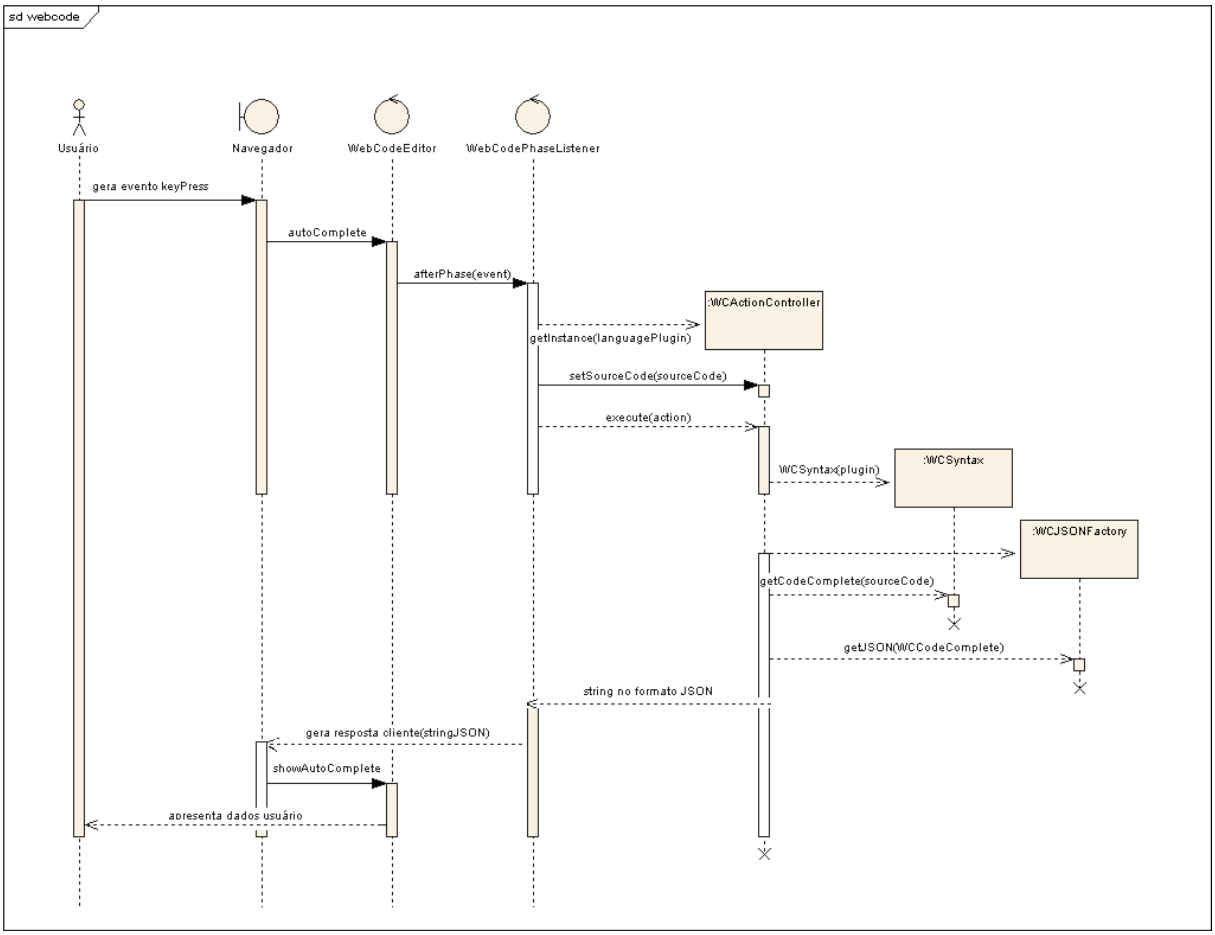


Figura 20 – Diagrama de sequência para o recurso de auto completar

O processo inicia quando ao digitar o código o usuário pressiona simultaneamente as teclas [ctrl] e [espaço] disparando então um evento `keydown` no navegador que por sua vez executa o método `autoComplete()` do objeto `WebCodeEditor`.

O método `autoComplete()` envia para o servidor uma requisição AJAX contendo o código digitado pelo usuário. A requisição é interceptada pelo *servlet* `FacesServlet` que por sua vez executa o método `afterPhase()` do objeto `WebCodePhaseListener`. Este objeto instancia a classe `WCActionController` que irá carregar o *plugin* anexado ao componente e depois executa o método `setSourceCode()` para informar ao objeto `WCActionController` o código recebido na requisição.

Após isso o objeto `WebCodePhaseListener` executa o método `executeAction()` do objeto `WCActionController` que por sua vez instancia as classes `WCJSONFactory` e `WCSyntax` e executa o método `getCodeComplete()` que irá retornar o objeto `WCCodeComplete` contendo os métodos e atributos do código. Este objeto é então passado como parâmetro na chamada do método `getJSON()` do objeto `WCJSONFactory`. Este método transforma o objeto `WCCodeComplete` em uma string no formato JSON que é repassada ao objeto `WebCodePhaseListener` para que o mesmo o envie ao navegador.

Ao receber a resposta o navegador executa o método `showAutoComplete()` do objeto `WebCodeEditor` para mostrar ao usuário os dados recebidos.

3.3 IMPLEMENTAÇÃO

Para o desenvolvimento do componente foram utilizadas as linguagens de programação Java e JavaScript, o *framework* Java Server Faces (JSF), a biblioteca FlexJSON, o ambiente de desenvolvimento NetBeans 6.1 e o servidor de aplicações web Apache-Tomcat 6.0.

3.3.1 Implementação do Módulo JSF

Para o desenvolvimento deste módulo foram realizados os seguintes passos:

a) definição da estrutura da *tag* `webcode`;
b) definição do arquivo de descrição de *tags* `webcode.tld`;
c) configuração do arquivo faces`-config.xml`;
d) implementação da classe `CommonTag`;
e) implementação da classe `WebCodeTag`;
f) implementação da classe `CommonHtmlRenderer`;
g) implementação da classe `Attributes`;
h) implementação da classe `Tags`;
i) implementação da classe `WebCodeRenderer`;
j) implementação da classe `WebCode`;
k) implementação da classe `WebCodePhaseListener`.

3.3.1.1 Definição e descrição da *tag* webcode

Para representar o componente em uma página HTML foi necessário especificar uma *tag* e um conjunto de atributos para ela. A definição da *tag* e seus atributos foi feita dentro do

arquivo `webcode.tld`.

A *tag* definida foi denominada `webcode` e os atributos especificados são os seguintes:

a) `componentId`: neste atributo pode-se especificar um id para o componente. Através do valor deste atributo é possível localizar o componente na árvore DOM. Se nenhum valor for definido o *framework* JSF gera e atribui um valor automaticamente;

b) `align`: neste atributo pode-se especificar o alinhamento do componente na tela. Os alinhamentos podem ser *left*, *center* ou *right*;

c) `languagePlugin`: através deste atributo pode-se especificar qual *plugin* o componente utilizará;

d) `onOpen`: neste atributo define-se a função JavaScript responsável por carregar um código existente. Esta função é disparada quando o usuário clicar no botão Abrir;

e) `onSave`: neste atributo define-se a função JavaScript responsável por salvar o código digitado. A função é disparada quando o usuário clicar no botão Salvar.

Seguindo a estrutura de um arquivo de descrição de *tags*, nas *tags* `name e tag-class` foram descritos respectivamente o nome da *tag* e a sua classe de processamento. O Quadro 19 apresenta a descrição destas duas *tags*.

```
<name>webCode</name>
<tag-class>br.furb.inf.tcc.imp.webcode.jsf.component.WebCodeTag</tag-class>
```

Quadro 19 – Definição do nome da *tag* e sua classe de processamento

Os atributos foram definidos através da *tag* `attribute`, que possui *tags* filhas para definir o nome do atributo, sua descrição e tipo.

O Quadro 20 apresenta a declaração do atributo `align`.

```
<attribute>
   <description>Alinhamento do editor na tela. Aceita
left, center, right. </description>
   <name>align</name>
   <deferred-value>
      <type>java.lang.String</type>
   </deferred-value>
</attribute>
```

Quadro 20 – *Tag* `webcode` e seus atributos

O Quadro 21 apresenta toda a especificação da *tag* `webcode`.

```
<tag>
    <name>webCode</name>
    <tag-class>br.furb.inf.tcc.imp.webcode.jsf.component.WebCodeTag</tag-class>
    <body-content>JSP</body-content>

    <attribute>
      <description>Alinhamento do editor na tela. Aceita left, center, right.
</description>
      <name>align</name>
      <deferred-value>
        <type>java.lang.String</type>
      </deferred-value>
    </attribute>

    <attribute>
      <description>Identificador do componente.</description>
      <name>componentId</name>
      <deferred-value>
        <type>java.lang.String</type>
      </deferred-value>
    </attribute>

    <attribute>
      <description>Define o plugin da linguagem a ser usado pelo
componente.</description>
      <name>languagePlugin</name>
      <deferred-value>
        <type>java.lang.String</type>
      </deferred-value>
    </attribute>

    <attribute>
      <description>Valor para o evento javaScript onSave.</description>
      <name>onSave</name>
      <deferred-value>
        <type>java.lang.String</type>
      </deferred-value>
    </attribute>

    <attribute>
      <description>Valor para o evento javaScript onOpen.</description>
      <name>onOpen</name>
      <deferred-value>
        <type>java.lang.String</type>
      </deferred-value>
    </attribute>
  </tag>
```

Quadro 21 – Definição da *tag* webcode e seus atributos

3.3.1.2 Configuração do arquivo faces.config.xml

Após definir a *tag* `webcode`, foi feita a configuração do arquivo `faces-config.xml`. No arquivo foram especificadas as *tags* `component`, `lifecycle` e `renderer`.

O Quadro 22 apresenta a definição do arquivo `faces-config.xml`.

```
<?xml version='1.0' encoding='UTF-8'?>

<!-- ========== FULL CONFIGURATION FILE ================================== -->

<faces-config version="1.2"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">

<component>
   <component-type>br.furb.inf.tcc.imp.webcode.jsf.component.WebCode</component-
type>
    <component-class>br.furb.inf.tcc.imp.webcode.jsf.component.WebCode</component-
class>
</component>

<lifecycle>
    <phase-
listener>br.furb.inf.tcc.imp.webcode.jsf.component.WebCodePhaseListener</phase-
listener>
</lifecycle>

<render-kit>
    <renderer>
        <component-family>javax.faces.Input</component-family>
        <renderer-type>br.furb.inf.tcc.imp.webcode.jsf.component.Text</renderer-
type>
        <renderer-
class>br.furb.inf.tcc.imp.webcode.jsf.component.WebCodeRenderer</renderer-class>
    </renderer>
</render-kit>

</faces-config>
```

Quadro 22 – Arquivo `faces-config.xml`

Na *tag* `component` foram definidos o tipo do componente e sua respectiva classe através das *tags* `component-type` e `component-class`. Na *tag* `lifecycle` foi utilizada a *tag* filha `phase-listener` para registrar a classe responsável por interceptar os eventos ocorridos após cada fase do ciclo de vida do JSF. A família do componente, o tipo de renderizador e a classe responsável pela renderização foram definidas respectivamente nas *tags* `component-family`, `renderer-type` e `renderer-class` que se enquadram dentro da *tag* `renderer` que por sua vez fica dentro da *tag* principal `render-kit`.

3.3.1.3 Implementação das classes de processamento de *tags*

Para o processamento da *tag* `webcode` foram definidas duas classes: `CommonTag` e `WebCodeTag`.

A classe `CommonTag` é a classe padrão desenvolvida para o processamento de *tags*. Possui métodos auxiliares para o processamento dos valores dos atributos das *tags*. O Quadro

23 apresenta os principais métodos desta classe.

Método	Parâmetros	Descrição
`setBooleanProperty()`	`UIComponent`, `String`, `ValueExpression`	Manipula atributos do tipo `boolean`.
`setStringProperty()`	`UIComponent`, `String`, `ValueExpression`	Manipula atributos do tipo `string`.
`setValueExpressionProperty()`	`UIComponent`, `String`, `ValueExpression`	Verifica se o valor de um atributo é uma expressão ou um literal. Se for uma expressão, a mesma é armazenada pelo componente para ser analisada posteriormente durante o ciclo de vida do JSF. Se for um literal, é testado o tipo de valor para depois chamar o método correspondente.

Quadro 23 – Principais métodos da classe `CommonTag`

O Quadro 24 apresenta a implementação destes métodos.

```
/* O valor é colocado diretamente no mapa de atributos do componente.
 * Trabalha com objetos do tipo String.
 */
private void setStringProperty(UIComponent component, String attributeName,
                               ValueExpression value)
{
   component.getAttributes().put(attributeName, value.getExpressionString());
}

/* O valor é colocado diretamente no mapa de atributos do componente.
 * Trabalha com objetos do tipo Boolean.
 */
private void setBooleanProperty(UIComponent component, String attributeName,
                                ValueExpression value)
{
   component.getAttributes().put(attributeName,
                               Boolean.valueOf(value.getExpressionString()));
}

/* Método que verifica se o valor do atributo do componente é literal ou uma
 * expressão. Caso seja uma expressão, a mesma será armazenada pelo componente para ser
 * analisada posteriormente durante o ciclo de vida do JSF. Se for literal é
 * testado o tipo do valor e chamado o método correspondete para o tipo.
 */
protected void setValueExpressionProperty(UIComponent component,
                                          String attributeName,
                                          ValueExpression value)
{
   if (value == null)
   {
      return;
   }
   if (value.isLiteralText())
   {
      if (value.getExpressionString().equalsIgnoreCase("true") ||
         value.getExpressionString().equalsIgnoreCase("false"))
      {
         this.setBooleanProperty(component, attributeName, value);
      } else
      {
         this.setStringProperty(component, attributeName, value);
      }
   } else
   {
      component.setValueExpression(attributeName, value);
   }
}
```

Quadro 24 – Implementação dos principais métodos da classe `CommonTag`

A classe `WebCodeTag` é a classe específica para o processamento da *tag* `webcode`, possuindo os atributos e seus respectivos métodos *getters* e *setters* que correspondem aos atributos definidos na *tag*.

A classe é responsável por receber o valor dos atributos, alocá-los no mapa de atributos do componente através do método `setValueExpressionProperty()` declarado na classe pai `CommonTag` e repassá-los para a classe `WebCode`. O Quadro 25 apresenta os principais métodos da classe `WebCodeTag`.

Método	Parâmetros	Descrição
`getComponentType()`	----	Método que retorna uma string contendo a identificação do tipo do componente. O valor de retorno deste método deve ser o mesmo valor declarado na *tag* `component-type` no arquivo `faces-config.xml`.
`getRendererType()`	----	Método que retorna uma string contendo o tipo de renderizador do componente. O valor de retorno deve ser o mesmo valor declarado na *tag* `renderer-type` no arquivo `faces-config.xml`.
`setProperties()`	`UIComponent`	Responsável por verificar se a instância do componente recebido no parâmetro é do tipo `WebCode` e por armazenar o valor dos atributos.

Quadro 25 – Principais métodos da classe `WebCodeTag`

O Quadro 26 apresenta a implementação destes métodos.

```
/**
 * Retorna string contendo identificação do tipo de componente.
 */
public String getComponentType()
{
   return WebCodeConstants.COMPONENT_TYPE;
}

/**
 * Retorna string contendo identificação do renderer do componente.
 */
public String getRendererType()
{
   return WebCodeConstants.RENDERER_TYPE;
}

/**
 * Método responsável por verificar se a instancia do componente recebido no
 * parâmetro é do tipo WebCode. Caso não seja gera uma exceção de componente
 * inválido.
 * É feita a chamada do método setValueExpressionAttribute para transferir o
 * valor dos atributos para o componente WebCode.
 */
   @Override
protected void setProperties(UIComponent component)
{
   if (!(component instanceof WebCode))
   {
      throw new IllegalArgumentException("Componente " + component.getClass().getName() +
         " não é um componente WebCode");
   }
   super.setProperties(component);

   this.setValueExpressionProperty(component, WebCodeConstants.ATTR_ALIGN, this.align);
   this.setValueExpressionProperty(component, WebCodeConstants.ATTR_COMPONENT_ID,
                                   this.componentId);
   this.setValueExpressionProperty(component, WebCodeConstants.ATTR_LANGUAGE_PLUGIN,
                                   this.languagePlugin);
   this.setValueExpressionProperty(component, WebCodeConstants.ATTR_SHOW_STATUSBAR,
                                   this.showStatusBar);
   this.setValueExpressionProperty(component, WebCodeConstants.ATTR_ONSAVE,
                                   this.onSave);
   this.setValueExpressionProperty(component, WebCodeConstants.ATTR_ONOPEN,
                                   this.onOpen);
   this.setValueExpressionProperty(component, WebCodeConstants.ATTR_WIDTH,
                                   this.width);
   this.setValueExpressionProperty(component, WebCodeConstants.ATTR_HEIGHT,
                                   this.height);
}
```

Quadro 26 – Implementação dos principais métodos da classe `WebCodeTag`

3.3.1.4 Implementação dos renderizadores do componente

O JSF disponibiliza para a geração de código de saída a classe `ResponseWriter` que estende a classe `java.io.Writer` e adiciona métodos para gerar elementos de markup. A instância desta classe é obtida através do método `getResponseWriter()` da classe `FacesContext` (JACOBI; FALLOWS, 2007, p. 66). Os principais métodos da classe `ResponseWriter` são `startElement()`, `endElement()` e `writeAttribute()`.

O Quadro 27 apresenta um exemplo do uso destes métodos e o Quadro 28 apresenta o código HTML resultante.

```
...
   // Recupera instância da classe ResponseWriter.
   ResponseWriter saida = context.getResponseWriter();
   saida.startElement(“input”,component);
   saida.writeAttribute(“name”,”inputExemplo”);
   saida.writeAttribute(“value”,”Exemplo”);
   saida.endElement(“input”);
...
```

Quadro 27 – Exemplo de uso dos métodos da classe `ResponseWriter`

```
<input name=”inputExemplo” value=”Exemplo”/>
```

Quadro 28 – Código HTML gerado pelos métodos

Para a renderização do componente, ou seja, para a geração do código HTML responsável por apresentar o componente no navegador foram desenvolvidas as classes `Attributes`, `Tags`, `CommonHtmlRenderer` e `WebCodeRenderer`.

A classe `Tags` foi desenvolvida para fornecer métodos que representam as *tags* HTML, como por exemplo os métodos `openDivTag()` e `closeDivTag()` que geram um código para abrir e fechar uma *tag* `div`. Usada em conjunto com a classe `Attributes` facilitou o processo de desenvolvimento do componente.

Cada método da classe `Tags` possui duas assinaturas sendo uma com e outra sem parâmetro. O método com parâmetro recebe uma instância da classe `Attributes`, classe responsável por definir e armazenar todos os atributos de uma *tag*. Os principais métodos da classe são `addAttribute()`, `replaceAttributeValue()` e `getAttributes()`.

O Quadro 29 apresenta um trecho de código da classe `Tags` que cria uma *tag* `table` com seus atributos.

```
/**
 * Método que insere um hash de atributos em uma tag html.
 */
private void insertAttribute(HashMap attributes) throws IOException
{
   if (attributes != null)
   {
      Object attribute = null;
      Object value = null;
      Iterator iterator = attributes.keySet().iterator();
      while (iterator.hasNext())
      {
         attribute = iterator.next();
         value = attributes.get(attribute);
         this.markupOutPut.writeAttribute(attribute.toString(), value.toString(), null);
      }
   }
}

/******************* Métodos para usar tag TABLE ************************/
public void openTableTag() throws IOException
{
   this.markupOutPut.startElement(HtmlConstants.TABLE, this.component);
}

public void openTableTag(Attributes attributes) throws IOException
{
   this.openTableTag();
   this.insertAttribute(attributes.getAttributes());
}

public void closeTableTag() throws IOException
{
   this.markupOutPut.endElement(HtmlConstants.TABLE);
}
```

Quadro 29 – Implementação dos métodos para criar a *tag* `table`

Para inserir recursos de JavaScript e CSS, inserir imagens e textos e para dar início ao processo de renderização do componente foi desenvolvida a classe `CommonHtmlRenderer`, classe base para o renderizador específico da *tag* `webcode`.

Através dos métodos `writeCSSResource()` e `writeJavaScriptResource()` é possível inserir recursos CSS e JavaScript na página gerada e também garantir que o mesmo recurso não vai ser escrito mais de uma vez. O Quadro 30 apresenta a implementação destes dois métodos.

```
/*
 * Método que escreve um recuro CSS no cliente garantindo que seja escrito apenas
 * uma vez.
 * Recupera um conjunto de recursos e caso o método add retorne false, um recurso
 * CSS já foi escrito.
 */
protected void writeCSSResource(FacesContext context,
                                String resourcePath) throws IOException
{
   Set cssResource = this.getCSSResources(context);

   if (cssResource.add(resourcePath))
   {
      String resourceURL = this.getResourceURL(context, resourcePath);
      ResponseWriter markupOutPut = context.getResponseWriter();
      markupOutPut.startElement(HtmlConstants.LINK, null);
      markupOutPut.writeAttribute(HtmlConstants.REL, "stylesheet", null);
      markupOutPut.writeAttribute(HtmlConstants.TYPE, "text/css", null);
      markupOutPut.writeAttribute(HtmlConstants.HREF, resourceURL, null);
      markupOutPut.endElement(HtmlConstants.LINK);
   }
}

/*
 * Método que escreve um recuro JavaScript no cliente garantindo que seja escrito
 * apenas uma vez.
 * Recupera um conjunto de recursos e caso o método add retorne false, um recurso
 * JavaScript já foi escrito.
 */
protected void writeJavaScriptResource(FacesContext context,
                                       String resourcePath) throws IOException
{
   Set javaScriptResource = this.getJavaScriptResources(context);

   if (javaScriptResource.add(resourcePath))
   {
      String resourceURL = this.getResourceURL(context, resourcePath);
      ResponseWriter markupOutPut = context.getResponseWriter();
      markupOutPut.startElement(HtmlConstants.SCRIPT, null);
      markupOutPut.writeAttribute(HtmlConstants.TYPE, "text/javascript", null);
      markupOutPut.writeAttribute(HtmlConstants.SRC, resourceURL, null);
      markupOutPut.endElement(HtmlConstants.SCRIPT);
   }
}
```

Quadro 30 - Métodos `writeCSSResource()` e `writeJavaScriptResource()`

O método `encodeBegin()` é responsável por dar início ao processo de renderização. Ele é invocado implicitamente pelo ciclo de vida do JSF e executa o método `encodeResources()` implementado na classe `WebCodeRenderer`, classe filha da classe `CommonHtmlRenderer`. O método também é responsável por instanciar as classes `Tags` e `Attributes`.

O método `encodeResources()` da classe `WebCodeRenderer` é responsável por executar o método `writeCSSResource()` para definir o arquivo CSS com as formatações do componente e por executar o método `writeJavaScriptResource()` para inserir as bibliotecas JavaScript necessárias para o funcionamento do componente.

O Quadro 31 apresenta a implementação deste método.

```
@Override
 protected void encodeResources(FacesContext context,
                                UIComponent component) throws IOException
 {
     if (!(component instanceof WebCode))
     {
         throw new IllegalArgumentException("Componente " +
                                            component.getClass().getName() +
                                            " não é um componente WebCode");
     }

    this.session.setAttribute("webcode", (WebCode)component);


    this.writeCSSResource(context,WebCodeConstants.RESOURCE_PATH + "/webCodeStyle.css");
    this.writeJavaScriptResource(context,WebCodeConstants.RESOURCE_JAVASCRIPT_PATH +
                                 "/aDoLo_framework.js");
    this.writeJavaScriptResource(context,WebCodeConstants.RESOURCE_JAVASCRIPT_PATH +
                                 "/WebCodeEditor.js");
    this.writeJavaScriptResource(context,WebCodeConstants.RESOURCE_JAVASCRIPT_PATH +
                                 "/WCParser.js");
    this.writeJavaScriptResource(context,WebCodeConstants.RESOURCE_JAVASCRIPT_PATH +
                                 "/WCSyntaxDefination.js");
    this.writeJavaScriptResource(context,WebCodeConstants.RESOURCE_JAVASCRIPT_PATH +
                                 "/WCToken.js");
    this.writeJavaScriptResource(context,WebCodeConstants.RESOURCE_JAVASCRIPT_PATH +
                                 "/WCTokenizer.js");
 }
```

Quadro 31 – Implementação do método `encodeResources()`

Após a execução do método `encodeResources()` o JSF executa o método `encodeEnd()` implementado na classe `WebCodeRenderer`. Este método executa os métodos responsáveis por montar a estrutura do componente. O Quadro 32 apresenta a descrição destes métodos.

Método	Parâmetros	Descrição
`beginRenderMainForm()`	`FacesContext`	Responsável por inserir o código JavaScript que instância a classe `WebCodeEditor`. Também cria a *tag* `div` que é o *container* que armazenas todas as outras partes do componente. O Quadro 33 apresenta a implementação deste método.
`endRenderMainForm()`	`FacesContext`	Finaliza a estrutura do componente.
`renderStatusBar()`	`FacesContext`	Método que cria a *status bar* do componente.
`renderWebCodeEditor()`	`FacesContext, UIComponent`	Método responsável por definir o `iframe` que corresponde à área de edição e por definir as funções JavaScript que serão executadas quando o componente for carregado no navegador. O Quadro 35 apresenta a implementação deste método.
`renderWebCodeToolBar()`	`FacesContext, WebCode`	Método responsável por criar a barra de ferramentas do componente e atribuir as funções JavaScript do objeto `WebCodeEditor` para cada botão da barra. O Quadro 34 apresenta um trecho da implementação deste método.

Quadro 32 – Principais métodos da classe `WebCodeRenderer`

```
protected void beginRenderMainForm(FacesContext facesContext) throws IOException
{
    this.attributes.clear();
    this.attributes.addAttribute(HtmlConstants.TYPE,"text/javascript");
    this.tags.openScriptTag(this.attributes);
    this.insertText(facesContext,"var wc = new WebCodeEditor();");
    this.tags.closeScriptTag();

    this.attributes.clear();
    this.attributes.addAttribute(HtmlConstants.ID, WebCodeIds.MainForm + this.componentId);
    this.attributes.addAttribute(HtmlConstants.ALIGN, this.align);
    this.tags.openDivTag(this.attributes);
}
```

Quadro 33 – Implementação do método `beginRenderMainForm()`

```
protected void renderWebCodeToolBar(FacesContext facesContext,
                                    WebCode component) throws IOException
{
 this.attributes.clear();
 this.attributes.addAttribute(HtmlConstants.ID, WebCodeIds.ToolBar);
 this.attributes.addAttribute(HtmlConstants.CLASS, "toolBar");
 this.attributes.addAttribute(HtmlConstants.ALIGN, "left");
 this.tags.openDivTag(this.attributes);

 this.attributes.clear();
 this.attributes.addAttribute(HtmlConstants.ON_MOUSE_OVER_IE,"wc.onMouseOverButton(this);");
 this.attributes.addAttribute(HtmlConstants.ON_MOUSE_OUT_IE,"wc.onMouseOutButton(this);");
 this.attributes.addAttribute(HtmlConstants.CLASS,"toolBarButtons");
 this.attributes.addAttribute(HtmlConstants.ALIGN,"center");

 this.tags.openTableTag();
 this.tags.openTrTag();

 this.attributes.addAttribute(HtmlConstants.ON_CLICK_IE,"wc.newCode();");
 this.tags.openTdTag(this.attributes);
 this.insertImage(facesContext,WebCodeConstants.RESOURCE_IMAGES_PATH + "/novo.gif",
                  "Novo Arquivo de Código");
 this.tags.closeTdTag();

 this.attributes.replaceAttributeValue(HtmlConstants.ON_CLICK_IE, "wc.saveCode(" +
                                       component.getOnSave() + ")");
 this.tags.openTdTag(this.attributes);
 this.insertImage(facesContext,WebCodeConstants.RESOURCE_IMAGES_PATH + "/salvar.gif",
                  "Novo Arquivo de Código");
 this.tags.closeTdTag();

 this.attributes.replaceAttributeValue(HtmlConstants.ON_CLICK_IE, "wc.openCode(" +
                                       component.getOnOpen() + ")");
 this.tags.openTdTag(this.attributes);
 this.insertImage(facesContext,WebCodeConstants.RESOURCE_IMAGES_PATH + "/abrir.gif",
                  "Abrir Arquivo de Código");
 this.tags.closeTdTag();
```

Quadro 34 – Trecho da implementação do método `renderWebCodeToolBar()`

```
protected void renderWebCodeEditor(FacesContext facesContext,
                                   UIComponent component) throws IOException
{
    String onLoad = "";
    onLoad = "wc.setContentWindow(this.contentWindow);" +
             "wc.setDesignModeOn();" +
             "wc.getContentWindow()." + this.getJS_AttachEvent(this.getJS_KeyDownEvent(),
             "wc.keyDownEvent") + ";" +
             "wc.init('" + WebCodeConstants.AJAX_URI + "');";

    this.attributes.clear();
    this.attributes.addAttribute(HtmlConstants.CLASS,"codeArea");
    this.attributes.addAttribute(HtmlConstants.WIDTH, this.width);
    this.attributes.addAttribute(HtmlConstants.HEIGHT, this.height);
    this.attributes.addAttribute(HtmlConstants.ID,WebCodeIds.CodeArea);
    this.attributes.addAttribute(HtmlConstants.SRC,this.getResourceURL(facesContext,
                                 WebCodeConstants.RESOURCE_PATH + "/codeArea.html"));
    this.attributes.addAttribute(HtmlConstants.ON_LOAD_IE,onLoad);



    this.tags.openIFrameTag(this.attributes);
    this.tags.closeIFrameTag();
}
```

Quadro 35 – Implementação do método `renderWebCodeEditor()`

3.3.1.5 Implementação das classes WebCode e WebCodePhaseListener

A classe `WebCode` é a classe que representa o componente e é responsável por armazenar e restaurar o valor dos atributos. Possui os mesmos atributos declarados na *tag* `webcode`. Os principais métodos da classe são `saveState()` e `restoreState()` que armazenam e recuperam o valor dos atributos. O Quadro 36 apresenta a implementação destes métodos.

```
/**
 * Método que restaura o estado armazenado do componente.
 */
public void restoreState(Object state, FacesContext facesContext)
{
   Object[] attributes = (Object[]) state;
   super.restoreState(facesContext, attributes[0]);
   this.setAlign((String) attributes[1]);
   this.setComponentId((String) attributes[2]);
   this.setLanguagePlugin((String) attributes[3]);
   this.setShowStatusBar(((Boolean) attributes[4]).booleanValue());
   this.setOnSave((String) attributes[5]);
   this.setOnOpen((String) attributes[6]);
   this.setHeight((String) attributes[7]);
   this.setWidth((String) attributes[8]);
}

/**
 * Método para gerenciar a preservação de estados do componente. Os valores dos
 * atributos são armazenados em um array.
 * Sobreescreve o método saveState da classe UIComponent.
 */
public Object saveState(FacesContext facesContext)
{
   Object[] attributesArray = new Object[9];
   attributesArray[0] = super.saveState(facesContext);
   attributesArray[1] = this.align;
   attributesArray[2] = this.componentId;
   attributesArray[3] = this.languagePlugin;
   attributesArray[4] = (Boolean) this.showStatusBar;
   attributesArray[5] = this.onSave;
   attributesArray[6] = this.onOpen;
   attributesArray[7] = this.height;
   attributesArray[8] = this.width;

   return attributesArray;
}
```

Quadro 36 – Implementação dos métodos `restoreState()` e `saveState()`

A classe `WebCodePhaseListener`, registrada no arquivo `faces-config.xml` para interceptar os eventos do ciclo de vida do JSF, é responsável por tratar as requisições AJAX e encaminhar as ações solicitadas para a classe `WCActionController`. Possui os métodos `beforePhase()` e `afterPhase()` que são executados pelo JSF antes e depois de uma fase do ciclo de vida. No método `getPhaseId()` foi definido que o JSF só iria executar a classe na última fase do ciclo de vida.

O método `afterPhase()` executa o método `controlAjaxRequest()`, responsável por recuperar o contexto da requisição, ler o conteúdo do parâmetro *action*, instanciar a classe `WCActionController` para executar a ação e montar a resposta que será devolvida ao navegador. O Quadro 37 apresenta a implementação do método `controlAjaxRequest()`.

```
private void controlAjaxRequest(PhaseEvent event)
{
 this.phaseId = PhaseId.RESTORE_VIEW;
 String requestType = event.getFacesContext().getViewRoot().getViewId();
 // Se não for uma requisição ajax, disparada pelo objeto javascript WebCodeEditor,
 // sai do método.
 if (!requestType.trim().equals(WebCodeConstants.AJAX_URI.trim()))
    return;
 // Recupera Request.
 HttpServletRequest request = (HttpServletRequest)event.getFacesContext().
                              getExternalContext().getRequest();
 // Recupera Response.
 HttpServletResponse response = (HttpServletResponse)event.getFacesContext().
                               getExternalContext().getResponse();
 // Se não vier o parâmetro action, sai do método e não faz nenhum processo.
 String actionParameter = request.getParameter("action");
 if (actionParameter == null)
     return;
 WebCode component = (WebCode)request.getSession().getAttribute("webcode");
 if (this.wcController == null)
   this.wcController = WCActionController.getInstance(component.getLanguagePlugin());

 String sourceCode = "";
 if (request.getParameter("code") != null)
     sourceCode = request.getParameter("code");

 this.wcController.setSourceCode(sourceCode);
 try
 {
    wcActions[] wcAction = wcActions.values();
    String resultAction =  this.wcController.execute(wcAction[Integer.parseInt(actionParameter)]);
    response.setContentType("text/xml");
    response.setHeader("Cache-Control", "no-cache");
    response.setHeader("Pragma", "no-cache");
    response.setDateHeader("Expires",-1);
    response.setCharacterEncoding("iso-8859-1");
    response.getWriter().write(resultAction);
    event.getFacesContext().responseComplete();
 }
 catch (WCException ex)
 {
    ex.printStackTrace();
 }
 catch (IOException exIO)
 {
    exIO.printStackTrace();
 }
}
```

Quadro 37 – Implementação do método `controlAjaxRequest()`

3.3.2 Implementação do módulo JavaScript

Entre as classes JavaScript desenvolvidas a principal é a `WebCodeEditor`. Esta classe, instanciada através do código JavaScript gerado pelo método `beginRenderMainForm()`, é responsável por executar as ações da barra de ferramentas, interceptar e tratar os eventos

gerados pelo teclado, fornecer os métodos *callback* para as chamadas AJAX e executar as funções responsáveis por abrir e salvar os códigos.

Ao carregar a página contendo o código HTML do componente gerado pelo JSF, o navegador executa o método `init()`. Este método solicita ao servidor através de requisições AJAX informações sobre o *plugin* como nome e versão e a definição da linguagem especificada no *plugin*, também registra no navegador através da função nativa do JavaScript `setInterval()` o método `formatSourceCode()` e o seu intervalo de execução. O Quadro 38 apresenta a implementação do método `init()`.

```
this.init = function(ajaxURI)
{

 ajaxUri = "faces"  + ajaxURI;
 if (!initialization)
 {

    try
    {
       framework.ajaxRequest("GET",false,getPluginInfoCallback,null,null,
                             onErrorCallback,ajaxUri+"?action=" +
                             wcActions.wcaGetPluginInfo,"");
       framework.ajaxRequest("GET",false,loadSyntaxDefinationCallback,null,null,
                             onErrorCallback,ajaxUri+"?action=" +
                             wcActions.wcaGetSyntaxDefination,"");
       initialization = true;
       contentWindow.focus();
       if (framework.isIE())
        {
          contentWindow.setInterval(formatSourceCode, 3000);
        }
       else
          {
             if (framework.isGecko())
                window.setInterval(formatSourceCode,3000);
          }
    }
    catch (exception)
    {
       onErrorCallback(exception.message);
    }
 }
};
```

Quadro 38 – Implementação do método `init()`

Quando a resposta da segunda requisição AJAX chega ao navegador é executado o método `loadSyntaxDefinationCallback()` que por sua vez executa o método `setSyntaxDefination()` passando como parâmetro a resposta recebida no formato JSON. O método `setSyntaxDefination()` carrega as definições da linguagem e cria uma instância da classe `WCParser`. O Quadro 39 apresenta a implementação deste método.

```
setSyntaxDefination = function(syntax)
{
   try
   {
      if (syntax == null)
         throw createErrorObject("WebCodeEditor setSyntaxDefination error:\n\
                                  syntaxDefination undefined.");

       var wcSyntaxDefination = new WCSyntaxDefination(syntax);
       parser = new WCParser(wcSyntaxDefination);
   }
   catch (exception)
   {
      onErrorCallback(exception.message);
   }
}
```

Quadro 39 – Implementação do método `setSyntaxDefination()`

Cada botão da barra de ferramentas está vinculado a um método da classe `WebCodeEditor`. Os métodos `openCode()` e `saveCode()` são responsáveis por executar as funções JavaScript informadas pelo usuário nos atributos `onOpen` e `onSave` da *tag* `webcode`. O método `newCode()` é responsável por limpar a área de edição para que um novo código seja digitado e os métodos `copyCommand()`, `pasteCommand()`, `cutCommand()`, `undoCommand()` e `redoCommand()` são responsáveis pelas funções de copiar, colar, recortar, desfazer e refazer. O Quadro 40 apresenta a implementação destes métodos.

```
//--------------- Funcionalidades da toolbar.
this.newCode = function()
{
  contentWindow.document.body.innerHTML = "";
};

this.saveCode = function(callbackFunction)
{
   callbackFunction();
};

this.openCode = function(callbackFunction)
{
   var code =  callbackFunction();
   contentWindow.document.body.innerHTML = code;
};

this.copyCommand = function()
{
   executeCommand("copy");
};

this.pasteCommand = function()
{
   executeCommand("paste");
};

this.cutCommand = function()
{
   executeCommand("cut");
};

this.undoCommand = function()
{
   executeCommand("undo");
};

this.redoCommand = function()
{
   executeCommand("redo");
};
```

Quadro 40 – Implementação das funcionalidades da barra de ferramentas

A cada intervalo de três segundos o método `formatSourceCode()`, responsável por inicializar o recurso de *syntax highlighting*, é executado pelo navegador. O método captura o código digitado pelo usuário, insere um código de controle para a localização do cursor, executa o método `clearCodeTags()` para remover as *tags* HTML e executa o método `parseToHTML()` da classe `WCParser` passando como parâmetro o código digitado. O código de controle usado para a localização do cursor é inserido através do método `createTextRange()` do JavaScript. A localização do cursor é feita através do método `findText()`. O Quadro 41 apresenta a implementação dos métodos `formatSourceCode()` e `clearCodeTags()`.

```
clearCodeTags = function(sourceCode)
{
   sourceCode = sourceCode.replace(/<br>/gi,'\n');
   sourceCode = sourceCode.replace(/<\/p>/g,'\n');
   sourceCode = sourceCode.replace(/<p>/gi,'\n');
   sourceCode = sourceCode.replace(/<.*?>/g,'');
   sourceCode = sourceCode.replace(/&lt;/g,'<');
   sourceCode = sourceCode.replace(/&gt;/g,'>');
   sourceCode = sourceCode.replace(/ /g,' ')
   sourceCode = sourceCode.replace(/ /g,' ');

   return sourceCode;
}

formatSourceCode = function()
{
   try
   {
   var code = contentWindow.document.body.innerHTML;
   if (code.trim().equals(""))
      return;

   contentWindow.document.selection.createRange().text = "\u2008";
   code = contentWindow.document.body.innerHTML;
   code = parser.parseToHTML(clearCodeTags(code))
   contentWindow.document.body.innerHTML = code;
   var range = contentWindow.document.body.createTextRange();
   if (range.findText("\u2008"))
      {
          range.select();
          range.text = '';
      }
   }
   catch (exception)
   {
       onErrorCallback(exception.message);
   }
};
```

Quadro 41 – Métodos `formatSourceCode()` e `clearCodeTags()`

A formatação do código no recurso de *syntax highlighting* é feita no método `parseToHTML()` que instância a classe `WCTokenizer` para fazer a análise léxica do código e retornar os *tokens*. Para cada *token* retornado é aplicada a formatação de acordo com a definição do *plugin*. O Quadro 42 apresenta a implementação deste método.

```
this.parseToHTML = function(sourceCode)
{
  try
  {
   var parserCode = "";
   var wcTokenizer = new WCTokenizer();
   wcTokenizer.setInput(sourceCode);
   wcTokenizer.setDefinations(syntaxDefination);

   var wcToken = wcTokenizer.nextToken();
   while (wcToken != null)
   {
     switch(wcToken.getType())
     {
      case wcTypes.wctEOL            : parserCode += "</br>";break;
      case wcTypes.wctNumber         : parserCode += parseToken(convertTokenToHTMLSpecialCharacter(
                                       wcToken.getLexeme()),syntaxDefination.getNumberColor(),
                                       false,false);break;
      case wcTypes.wctLiteral        : parserCode += parseToken(convertTokenToHTMLSpecialCharacter(
                                       wcToken.getLexeme()),syntaxDefination.getLiteralColor(),
                                       false,false);break;
      case wcTypes.wctLineComment    : parserCode += parseToken(convertTokenToHTMLSpecialCharacter(
                                       wcToken.getLexeme()),syntaxDefination.getCommentColor(),
                                       false,false);break;
      case wcTypes.wctBlockComment   : parserCode += parseToken(convertTokenToHTMLSpecialCharacter(
                                       wcToken.getLexeme()).replace("\n","</br>"),
                                       syntaxDefination.getCommentColor(),false,false);break;
      case wcTypes.wctWhiteSpace     : parserCode += " ";break;
      case wcTypes.wctWord           : parserCode += parseToken(convertTokenToHTMLSpecialCharacter(
                                       wcToken.getLexeme()),"BLACK",false,false);break;
      case wcTypes.wctTokenDelimeter : parserCode += parseToken(convertTokenToHTMLSpecialCharacter(
                                       wcToken.getLexeme()),syntaxDefination.getDelimetersColor(),
                                       false,false);break;
      case wcTypes.wctSyntaxItem     : parserCode += parseToken(convertTokenToHTMLSpecialCharacter(
                                       wcToken.getLexeme()),
                                       syntaxDefination.getSyntaxItemColor(wcToken.getLexeme()),
                                       syntaxDefination.isSyntaxItemBold(wcToken.getLexeme()),
                                       syntaxDefination.isSyntaxItemItalic(wcToken.getLexeme()));break;
     }
      wcToken = wcTokenizer.nextToken();
   }
   return parserCode;
  }
  catch (exception)
  {
      throw createErrorObject("WebCodeParser parseToHTML error.\n" + exception.message);
  }
}
}
```

Quadro 42 – Implementação do método `parseToHTML()`

O pressionamento simultâneo das teclas [ctrl] e [espaço] na área de edição de código gera uma evento `keyDown()` no navegador que é interceptado pelo método `keyDownEvent()` da classe `WebCodeEditor`. O método analisa as teclas pressionadas e dispara o método `autoComplete()`, responsável por enviar o código digitado através de AJAX ao servidor para que o recurso de auto complemento seja executado. O Quadro 43 apresenta o tratamento das teclas pressionadas na área de edição.

```
this.keyDownEvent = function(event)
{
   if (framework.isIE())
   {
     var cod = window.event.keycode;
     if (cod == 32 && window.event.ctrlKey == true)
        autoComplete();
   }
   else
      if (framework.isGecko())
      {
         if(event.charCode == 32 && event.ctrlKey == true)
            autoComplete();
      }
};
```

Quadro 43 – Tratamento do pressionamento de teclas

3.3.3 Implementação do módulo de execução

As classes de execução são responsáveis por executar as ações solicitadas pelo componente no navegador, que chegam através de requisições AJAX.

A classe `WebCodePhaseListener` ao receber uma requisição AJAX analisa os parâmetros para ver o tipo de ação que esta sendo solicitada e a encaminha para a classe `WCActionController` através do método `execute()`. Foram definidas três tipos de ações que o componente pode solicitar:

a) `wcaCodeComplete`: solicita execução do recurso de auto complemento;

b) `wcaGetSyntaxDefination`: solicita ao *plugin* as definições da linguagem;

c) `wcaGetPluginInfo`: solicita ao *plugin* suas informações de nome e versão.

O método `execute()` carrega o *plugin* anexado ao componente através do método `loadPlugin()`, executa a ação solicitada e depois repassa o objeto de retorno para a classe `WCJSOFactory` que converte o objeto em um `string` JSON. O Quadro 44 apresenta a implementação do método `execute()`.

```
public String execute(wcActions action)throws WCException
{
   if (action.equals(wcActions.wcaCodeComplete) ||action.equals(wcActions.wcaFormatCode))
      if (this.sourceCode.trim().equals(""))
         return this.jsonFactory.getJSON(new WCError("Error on execute action '" +
                                         action.name() + "': Source Code undefined."));

   try
   {
      String result = "";

      WebCodePlugin plugin = this.loadPlugin();
      WCSyntax wcSyntax = new WCSyntax(plugin);

      this.jsonFactory = new WCJSONFactory();
      switch (action)
      {
          case wcaCodeComplete        : result = this.jsonFactory.getJSON(
                                        wcSyntax.getCodeComplete(this.sourceCode));break;
          case wcaFormatCode          : result = this.jsonFactory.getJSON(
                                        wcSyntax.getFormatCode(this.sourceCode));break;
          case wcaGetSyntaxDefination : result = this.jsonFactory.getJSON(
                                        wcSyntax.getSyntaxDefination());break;
          case wcaGetPluginInfo       : result = this.jsonFactory.getJSON(plugin);break;
      }
      return result;
   }
   catch (WCException ex)
   {
      return this.jsonFactory.getJSON(new WCError("Error on execute action '" +
                                      action.name() + "'\n" + ex.toString()));
   }
}
```

Quadro 44 – Implementação do método `execute()`

O Quadro 45 apresenta a implementação do método `loadPlugin()`.

```
private WebCodePlugin loadPlugin() throws WCException
{
   try
   {
      Class<?> loadClass = Class.forName(this.languagePlugin);
      Constructor loadConstructor = loadClass.getConstructor();
      return (WebCodePlugin)loadConstructor.newInstance();
   }
   catch (Exception ex)
   {
      throw new WCException("WCActionController loadPlugin error: Plugin: '" +
      this.languagePlugin + "'\n" + ex.toString());
   }
}
```

Quadro 45 – Método `loadPlugin()`

A classe `WCSyntax`, instanciada no método `execute()`, implementa os métodos que executam as ações solicitadas. A implementação destes métodos é apresentada no Quadro 46.

```
public WCSyntaxDefination getSyntaxDefination()throws WCException
{
   return this.plugin.getWebCodeSyntaxDefinations();
}

public WCCodeComplete getCodeComplete(String sourceCode) throws WCException
{
    return this.plugin.getWebCodeComplete(sourceCode);
}
```

Quadro 47 – Métodos que executam as ações

A ação `wcaGetPluginInfo` não possui uma implementação na classe `WCSyntax`, pois a classe do *plugin* é passada diretamente para o método `getJSON()`.

Para o mecanismo de *plugins* foi definida a interface `WebCodePlugin` apresentada no Quadro 47. Também foram definidas as classes `WCSyntaxDefination`, `WCSyntaxItemDefination`, `WCTokenDelimeter`, `WCCodeComplete`, `WCField` e `WCMethod`.

```
package br.furb.inf.tcc.imp.webcode.plugin;

import br.furb.inf.tcc.imp.webcode.WCException;
import br.furb.inf.tcc.imp.webcode.syntax.WCSyntaxDefination;
import br.furb.inf.tcc.imp.webcode.syntax.code.WCCodeComplete;

public interface WebCodePlugin
{
   public abstract WCSyntaxDefination getWebCodeSyntaxDefinations() throws WCException;
   public abstract WCCodeComplete getWebCodeComplete(String sourceCode) throws WCException;
   public abstract String getPluginVersion() throws WCException;
   public abstract String getPluginName() throws WCException;
}
```

Quadro 47 – Interface `WebCodePlugin`

A classe que implementar a interface `WebCodePlugin` deve implementar os métodos descritos no Quadro 48.

Método	Parâmetros	Descrição
`getPluginName()`	`---`	Retorna nome do plugin.
`getPluginVersion()`	`---`	Retorna a versão do plugin.
`getWebCodeComplete()`	`String`	Retorna os dados para auto complemento.
`getWebCodeSyntaxDefinations()`	`---`	Retorna toda a definição da linguagem.

Quadro 48 – Métodos da interface `WebCodePlugin`

Através da classe `WCSyntaxDefination` o usuário que está criando o plugin pode definir a linguagem que ele deseja. A classe fornece métodos para definir comentários de linha e bloco, `strings` e `chars` e as cores que serão usadas para pintar as palavras no processo de *syntax highlighting*. Também é possível definir os *tokens* delimitadores da linguagem através da classe `WCSyntaxTokenDelimeter`. As palavras reservadas, classes pré-definidas e outros itens da linguagem podem ser definidos na classe

`WCSyntaxItemDefination`.

Como exemplo, o componente disponibiliza a classe `JavaPlugin` que é o plugin para o reconhecimento da linguagem Java. O Quadro 49 apresenta o método `setTokenSeparator()` que define os *tokens* delimitadores da linguagem.

```
private void setTokenSeparator()
{
   WCSyntaxTokenDelimeter delimeter = new WCSyntaxTokenDelimeter();
   delimeter.setColor("Red");


   delimeter.addDelimeter("+");
   delimeter.addDelimeter("-");
   delimeter.addDelimeter("*");
   delimeter.addDelimeter("/");
   delimeter.addDelimeter("=");
   delimeter.addDelimeter("!");
   delimeter.addDelimeter("(");
   delimeter.addDelimeter(")");
   delimeter.addDelimeter("[");
   delimeter.addDelimeter("]");
   delimeter.addDelimeter("{");
   delimeter.addDelimeter("}");
   delimeter.addDelimeter(":");
   delimeter.addDelimeter(">");
   delimeter.addDelimeter("<");
   delimeter.addDelimeter(".");
   delimeter.addDelimeter("&");
   delimeter.addDelimeter("|");
   delimeter.addDelimeter(".");
   delimeter.addDelimeter(";");
   delimeter.addDelimeter(",");

   this.wcDefination.setTokenDelimeters(delimeter);
}
```

Quadro 49 – Método para definir os *tokens* delimitadores da linguagem Java

O método `setPredefinedClasses()`, apresentado no Quadro 50, define algumas classes pré-definidas do Java.

```
private void setPredefinedClasses()
{
   this.wcSyntaxItem = new WCSyntaxItemDefination();
   this.wcSyntaxItem.setColor("Red");
   this.wcSyntaxItem.setItalic(true);

   this.wcSyntaxItem.addItem("String");
   this.wcSyntaxItem.addItem("Integer");
   this.wcSyntaxItem.addItem("Double");
   this.wcSyntaxItem.addItem("File");
   this.wcSyntaxItem.addItem("PrintStream");
   this.wcSyntaxItem.addItem("PrintWriter");

   this.wcDefination.addSyntaxItem(this.wcSyntaxItem);
}
```

Quadro 50 – Definição das classes pré-definidas

O Quadro 51 apresenta um trecho do método que define as palavras reservadas do

Java.

```
private void setReservedWords()
{
   this.wcSyntaxItem = new WCSyntaxItemDefination();
   this.wcSyntaxItem.setColor("Blue");
   this.wcSyntaxItem.setBold(true);
   this.wcSyntaxItem.setItalic(true);


   this.wcSyntaxItem.addItem("abstract");
   this.wcSyntaxItem.addItem("boolean");
   this.wcSyntaxItem.addItem("break");
   this.wcSyntaxItem.addItem("byte");
   this.wcSyntaxItem.addItem("case");
   this.wcSyntaxItem.addItem("catch");
   this.wcSyntaxItem.addItem("char");
   this.wcSyntaxItem.addItem("class");
   this.wcSyntaxItem.addItem("const");
   this.wcSyntaxItem.addItem("continue");
   this.wcSyntaxItem.addItem("default");
   this.wcSyntaxItem.addItem("do");
   this.wcSyntaxItem.addItem("double");
   this.wcSyntaxItem.addItem("else");
   this.wcSyntaxItem.addItem("extends");
   this.wcSyntaxItem.addItem("final");
   this.wcSyntaxItem.addItem("finally");
   this.wcSyntaxItem.addItem("float");
   this.wcSyntaxItem.addItem("for");
   this.wcSyntaxItem.addItem("if");
   this.wcSyntaxItem.addItem("implements");
   this.wcSyntaxItem.addItem("import");
   this.wcSyntaxItem.addItem("instanceof");
   this.wcSyntaxItem.addItem("int");
   this.wcSyntaxItem.addItem("interface");
   this.wcSyntaxItem.addItem("long");
   this.wcSyntaxItem.addItem("new");
   this.wcSyntaxItem.addItem("package");
   this.wcSyntaxItem.addItem("private");
   this.wcSyntaxItem.addItem("protected");
```

Quadro 51 – Trecho do método `setReservedWords()`

O Quadro 52 apresenta o método `getWebCodeComplete()` que retorna os atributos e métodos do código Java digitado.

```
public WCCodeComplete getWebCodeComplete(String sourceCode)
{
   try
   {
       WCCodeComplete wcComplete = new WCCodeComplete();
       String metodo = "";
       String atributo = "";
       this.createClass(sourceCode);
       Class c = Class.forName(getClassName(sourceCode));
       // Recupera os métodos da classe.
       Method methods[] = c.getDeclaredMethods();
       for(int count=0;count<methods.length;count++)
       {
          Method m = methods[count];
          metodo = m.getName() + " " + m.getReturnType();
          wcComplete.addMethod(new WCMethod(metodo));
       }
       // Recupera os atributos da classe.
       Field fields[] = c.getDeclaredFields();
       for(int count=0;count<fields.length;count++)
       {
          Field f = fields[count];
          atributo = f.getName() + " " + f.getType();
          wcComplete.addField(new WCField(atributo));
       }
       return wcComplete;
   }
   catch (Exception e)
   {
      return null;
   }
}
```

Quadro 52 – Implementação do método `getWebCodeComplete()`

3.3.4 Operacionalidade da implementação

Esta seção apresenta os passos e as bibliotecas necessárias para a utilização do componente.

Primeiramente é necessário adicionar ao projeto as bibliotecas que implementam a especificação do JSF, a biblioteca `flexjson.jar` e a biblioteca `webcode.jar`, biblioteca do componente.

Partindo da premissa que o *plugin* da linguagem já está definido ou seja, a classe que realiza a interface `WebCodePlugin` já está criada, o usuário deve adicionar na página as *taglibs* para as bibliotecas do JSF e a *taglib* do componente conforme apresenta o Quadro 53.

```
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@taglib prefix="t" uri="http://tcc.imp.webcode"%>
```

Quadro 53 – Definição das *taglibs* JSF e `WebCode`

Após definir as bibliotecas, o usuário pode inserir na página a *tag* do componente e configurar seus atributos, definido o *id* do componente, alinhamento, classe do *plugin* e as funções para abrir e salvar o código conforme apresenta o Quadro 54.

```
<f:view>
   <t:webCode componentId="webCode" align="center"
              languagePlugin="br.furb.inf.tcc.imp.webcode.plugin.JavaPlugin"
              onOpen="abrirDados" onSave="salvarDados">

   </t:webCode>
</f:view>
```

Quadro 54 – Definição da *tag* `webcode` na página

Ao executar o projeto será apresentada no navegador a tela de edição de código criada pelo componente. A Figura 21 apresenta a tela criada pelo componente. O usuário pode criar um novo código (1), salvar o código digitado (2), abrir um código (3), desfazer uma ação (4), refazer uma ação (5), copiar trecho de código selecionado (6), colar código (7), recortar trecho de código selecionado (8) e visualizar informações do componente (9).

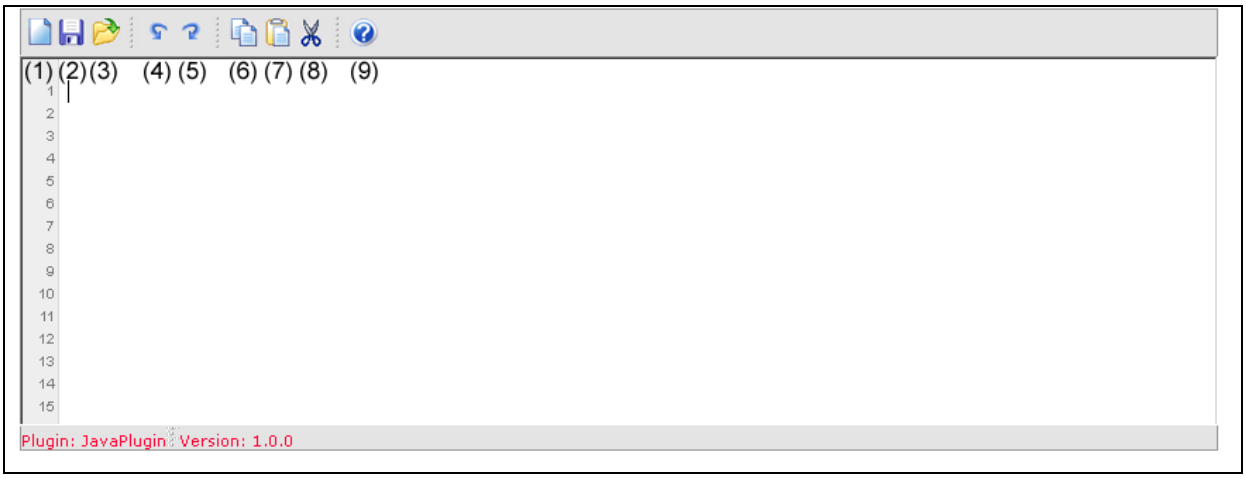


Figura 21 – Tela de edição de código

Após a tela ser criada pode-se começar a digitar o código. A Figura 22 apresenta um código Java sendo digitado na área de edição.

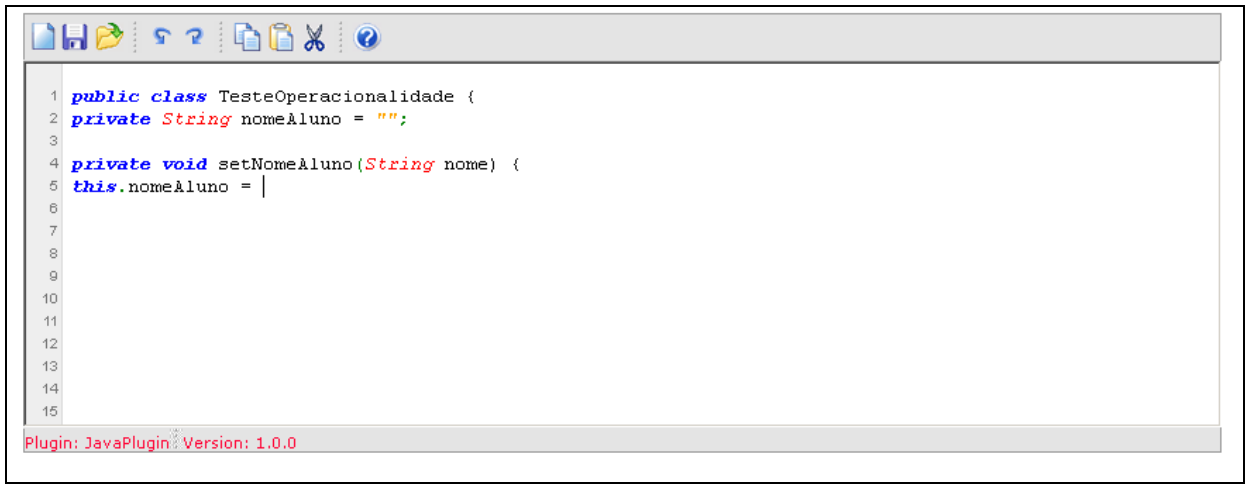


Figura 22 – Digitação de código Java no editor

No momento da digitação do código pode-se acionar o recurso de auto complemento de código pressionando simultaneamente as teclas [ctrl] e [espaço]. A Figura 23 apresenta o recurso de auto complemento.

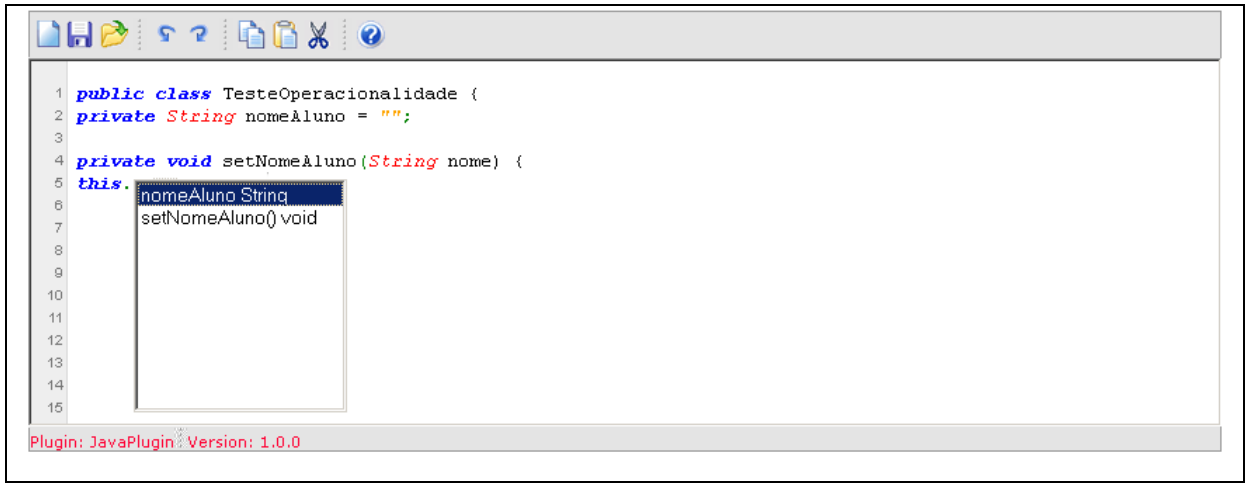


Figura 23 – Recurso de auto complemento

Ao clicar no botão de abrir na barra de tarefas o componente executa a função definida na *tag* do componente para carregar o código. Para exemplificar o recurso de abrir código, foi definida na página onde foi inserido o componente uma função JavaScript que carrega um código Java definido na própria função. O Quadro 55 apresenta a implementação desta função e a Figura 24 apresenta o código carregado no editor.

```
function abrirDados()
        {
           return 'public class ClasseTeste\n' +
                  '{\n' +
                  '   private String nome ="leonardo";\n' +
                  '   private double numero = 5.6;\n' +
                  '\n' +
                  '   public void setNome(String nome)\n' +
                  '   {\n' +
                  '      this.nome = nome;\n' +
                  '   }\n' +
                  '}\n'
        }
```

Quadro 55 – Função para abrir código

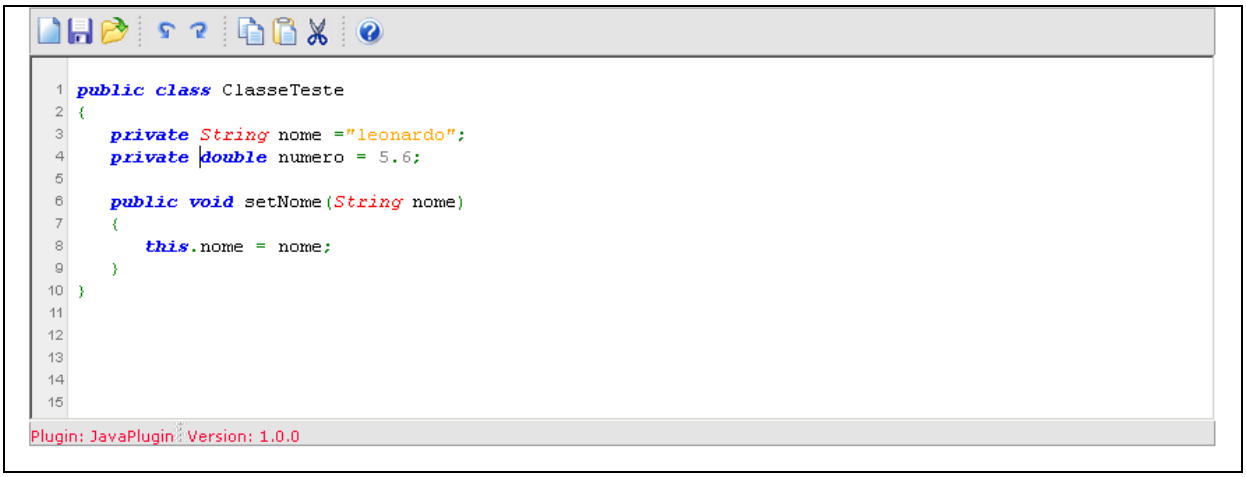


Figura 24 – Código carregado no editor

3.4 RESULTADOS E DISCUSSÃO

No Quadro 56 é apresentada uma comparação entre o componente desenvolvido e as ferramentas apresentadas na seção de trabalhos correlatos.

	Eclipse	**CodePress**	**WebCode**
Ambiente	Desktop	Web	Web
Recurso de plugins	Possui	Não Possui	Possui
Criação de plugins	Complexa	Não Possui	Simples
Abrir e salvar código	Possui	Não Possui	Possui
Auto complemento de código	Possui	Não possui	Possui
Syntax highlighting	Possui	Possui	Possui
Reconhecimento das linguagens	Por meio de plugins	CSS, Classes Java Script	Por meio de plugins

Quadro 56 – Comparação entre as ferramentas

Como visto no Quadro 56, o editor CodePress oferece um menor número de recursos, tornando-o limitado principalmente no reconhecimento de linguagens. Também pode ser visto que todas as ferramentas oferecem o recurso de *syntax highlighting* e apenas a plataforma Eclipse trabalha sobre um ambiente *desktop*.

É importante destacar que o WebCode possui os mesmos recursos do ambiente Eclipse, se considerar a comparação entre os três. Porém, o reconhecimento das novas linguagens é mais simplificada no WebCode.

4 CONCLUSÕES

O objetivo principal do trabalho, implementar um componente que possibilite a edição de código fonte e com suporte a *plugins* para reconhecimento das linguagens de programação, foi atingido. O componente permite sua utilização em uma página HTML através de uma *tag* customizada e disponibiliza entre outros recursos as funções de *syntax highlighting* e auto complemento de código.

Para a implementação do componente foi necessário um estudo sobre o *framework* JSF, desenvolvimento AJAX e DOM. Este último foi necessário para o recurso de auto complemento de código para criar dinamicamente uma lista com os dados a serem mostrados. Uma leitura sobre compiladores foi necessária visto que para o recurso de *syntax highlighting* foi implementado um analisador léxico que separa os *tokens* do código e os identifica para que depois seja aplicada a formatação conforme seu tipo.

A distribuição do componente através de um arquivo .jar possibilita sua utilização em diversos projetos. Pode-se por exemplo utilizar o componente como parte de um ambiente web completo para o desenvolvimento de aplicativos.

O componente não apresentou incompatibilidades nos navegadores Internet Explorer e Mozilla FireFox.

4.1 LIMITAÇÕES

O componente apresenta algumas limitações como:

a) não disponibilizar um meio para verificar a estrutura do código digitado, problemas com variáveis e chamadas de métodos com parâmetros incorretos através de análise sintática e semântica;
b) não disponibilizar opção para realizar a configuração do editor como tipo e tamanho de fonte;
c) não disponibilizar recurso para ocultar blocos de código;
d) não disponibilizar meios para alterar as dimensões de altura e largura do componente.

e) não manter a formatação da última digitada palavra após digitar um *token* delimitador;
f) não mostrar o cursor após o processo de *syntax highlighting*.

4.2 EXTENSÕES

Como possíveis extensões para o componente sugere-se:

a) alteração da interface do plugin para disponibilizar recursos de análise sintática e semântica;
b) opção para configuração do editor;
c) opção para adicionar novos botões de funcionalidades na barra de ferramentas;
d) aperfeiçoar o analisador léxico desenvolvido, melhorando o processo de reconhecimento de literais, números e comentários;
e) disponibilizar através de novos atributos da tag `webcode` uma opção para alterar as dimensões do componente;
f) aperfeiçoar a área de edição de código para fornecer recurso de ocultar blocos de código.

REFERÊNCIAS BIBLIOGRÁFICAS

ANSWERS. **Plugin**. [S.l.], 2006. Disponível em: <http://www.answers.com/topic/plugin?cat=technology#copyright >. Acesso em: 22 set. 2008.

ARAÚJO, Flavio G. **Desenvolvimento de framework para aplicações .Net.** [S.l.], 2005. Disponível em: <http://www.linhadecodigo.com.br/Artigo.aspx?id=858>. Acesso em: 17 set. 2008.

BEZERRA, Eduardo. **Princípios de análise e projeto de sistemas com UML**. Rio de Janeiro: Campus, 2002.

BURNS, Ed; KITAIN, Roger. **Java Server Faces Specification**. [S.l.], 2004. Disponível em: <https://javaserverfaces-spec-public.dev.java.net/>. Acesso em: 12 set. 2008.

BYSTRONSKI, Márcio. **Um ambiente para integração de ferramentas de tolerância a falhas.** 2004. 35 f. Trabalho de Graduação (Engenharia da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

CARDOSO, Márdel. **Desenvolvimento web para o ensino superior**. Rio de Janeiro: Axcel Books, 2004.

CARLOS, José. **Código nos posts e syntax highlighting.** [S.l.], 2007. Disponível em: <http://logon.com.pt/2007/04/12/cdigo-nos-posts-e-syntax-highlighting/>. Acesso em: 23 set. 2008.

CODEPRESS. **About CodePress**. [S.l.], 2007. Disponível em: <http://codepress.org >. Acesso em: 5 set. 2008.

CRANE, Dave; PASCARELLO, Eric; JAMES, Darre. **AJAX em ação**. Tradução Edson Furmankiewicz e Carlos Schafranski. São Paulo: Pearson Education do Brasil, 2007.

DELAMARO, Márcio E. **Como construir um compilador utilizando ferramentas Java**. São Paulo: Novatec, 2006.

FOWLER, Martin. **UML essencial**: um breve guia para a linguagem-padrão de modelagem de objetos. Tradução João Tortello. Porto Alegre: Bookman, 2005.

GARRET, Jesse J. **A new aproach to web applications**. [S.l.], 2005. Disponível em: <http://www.adaptivepath.com/publications/essays/archives/000385.php>. Acesso em: 25 ago. 2008.

GONÇALVES, Bruno. **Estudo da concepção, definição e projeto de aplicações web enriquecidas com AJAX.** 2006. 70 f. Trabalho de Conclusão de Curso (Licenciatura Plena em Informática) – Instituto de Ciências e Letras do Médio Araguaia, Universidade Federal do Mato Grosso, Pontal do Araguaia.

GONÇALVES, Edson. **Dominando AJAX**. Rio de Janeiro: Ciência Moderna, 2006.

GOODMAN, Danny; MORRISON, Michael. **JavaScript bible**. 5. ed. Indiana: Wiley Publishing, 2004.

GRUNE, Dick et al. **Projeto moderno de compiladores**: implementação e aplicações. Tradução Vandenberg D. de Souza. Rio de Janeiro: Campus, 2001.

HUBBARD, Charlie. **FlexJSON**. [S.l.], 2007. Disponível em: <http://flexjson.sourceforge.net>. Acesso em: 15 out 2008.

JACOBI, Jonas; FALLOWS, John R. **Pro JSF e AJAX**: construindo componentes ricos para internet. Tradução Cleuton Sampaio de Melo Jr. Rio de Janeiro: Ciência Moderna, 2007.

JACONETE, Fernando. **JSON**: JavaScript Object Notation. [S.l.], 2006. Disponível em: <http://www.devmedia.com.br/articles/viewcomp.asp?comp=6992&hl=json>. Acesso em: 20 out. 2008.

JSON. **Introdução ao JSON**. [S.l], 2006. Disponível em: <http://www.json.org>. Acesso em: 02 out. 2008.

KURNIAWAN, Budi. **Java para web com servlets, JSP e EJB**. Tradução Savannah Hartmann. Rio de Janeiro: Ciência Moderna, 2002.

LEYENDECKER, Gustavo Z. **Especificação e compilação de uma linguagem de programação orientada a objetos para a plataforma Microsoft .NET.** 2005. 89 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

LIMEIRA, José L. **Utilização de AJAX no desenvolvimento de sistemas web.** 2006. 44 f. Trabalho de Conclusão de Curso (Especialização em web e Sistemas de Informação) – Curso de Pós-graduação em web e Sistemas de Informação – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

LOPES, Cláudio. **Implementação de um analisador léxico**: a primeira etapa na construção do compilador Marvel. 2007. 70 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Departamento de Ciência da Computação, Universidade Federal de Juiz de Fora, Juiz de Fora.

LOUDEN, Kenneth C. **Compiladores**: princípios e práticas. Tradução Flavio Soares Correia da Silva. São Paulo: Pioneira Thomson Learning, 2004.

MURRAY, Gregory. **Including Ajax functionality in a custom Java Server Faces component**. [S.l.], 2008. Disponível em: <http://java.sun.com/javaee/javaserverfaces/ajax/tutorial.jsp>. Acesso em: 15 out. 2008.

OLIVEIRA, Eric C. **Projeto Eclipse for Java.** [S.l.], 2005. Disponível em: <http://www.linhadecodigo.com.br/Artigo.aspx?id=677>. Acesso em: 17 set. 2008.

PITANGA, Taliga. **Java Server Faces**: a mais nova tecnologia para desenvolvimento web. [S.l.], 2004. Disponível em: <www.guj.com.br/viewtopic.php?t=72288>. Acesso em: 22 jun. 2008.

PROULX, Emmanuel. **Eclipse plugins exposed**. [S.l.], 2005. Disponível em: <http://www.onjava.com/pub/a/onjava/2005/02/09/eclipse.html>. Acesso em: 17 nov. 2008.

RAMALHO, José A. **Iniciando em HTML**. São Paulo: Makron Books, 1996.

SILVA, Osmar J. **JavaScript avançado**: animação, interatividade e desenvolvimento de aplicativos. São Paulo: Érica, 2003.

SOARES, Walace. **AJAX (Asynchronous Java Script and Xml)**: guia prático para Windows. São Paulo: Érica, 2006.

SOUSA, Marcos. **Unindo JavaServer Faces e AJAX:** melhorando o processo de desenvolvimento web. [S.l.], 2006. Disponível em: <http://www.devmedia.com.br/articles/viewcomp.asp?comp=3199>. Acesso em: 23 set. 2008.

TEAGUE, Jason C. **DHTML e CSS para World Wide Web**. 2. ed. Tradução Kátia Roque. Rio de Janeiro: Campus, 2001.