UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
CURSO SUPERIOR DE TECNOLOGIA EM DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO

JONATHAN NASCIMENTO WELZEL

O FRAMEWORK JAVASERVER FACES

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA
2011

JONATHAN NASCIMENTO WELZEL

O FRAMEWORK JAVASERVER FACES

Trabalho de Diplomação apresentado à disciplina de Trabalho de Diplomação, do curso Superior de Tecnologia em Desenvolvimento de Sistemas de Informação - CSTDSI - da Universidade Tecnológica Federal do Paraná - UTFPR - como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. *MSc*. Everton Coimbra de Araújo.

MEDIANEIRA

2011

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Gerência de Ensino
Curso Superior de Tecnologia em Desenvolvimento
de Sistemas de Informação

TERMO DE APROVAÇÃO

O FRAMEWORK JAVASERVER FACES

Por

JONATHAN NASCIMENTO WELZEL

Este Trabalho de Diplomação (TD) foi apresentado às 16:50 h do dia 14 de junho de 2011 como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Desenvolvimento de Sistemas de Informação, da Universidade Tecnológica Federal do Paraná, *Campus* Medianeira. O candidato foi argüido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. Everton Coimbra de Araújo, *MSc*.
UTFPR – *Campus* Medianeira
(Orientador)

Prof. Fernando Schütz
UTFPR – *Campus* Medianeira
(Convidado)

Prof. Juliano Rodrigo Lamb, *M. Eng.*
UTFPR – *Campus* Medianeira
(Responsável pelas atividades de TCC)

Prof. Juliano Rodrigo Lamb, *M. Eng.*
UTFPR – *Campus* Medianeira
(Convidado)

A folha de aprovação assinada encontra-se na Coordenação do Curso.

RESUMO

WELZEL, Jonathan Nascimento**. O *Framework* JavaServer Faces**. 2011, 102f. Trabalho de Conclusão do Curso Superior de Tecnologia em Desenvolvimento de Sistemas de Informação. Universidade Tecnológica Federal do Paraná, Medianeira, 2011.

Para que a crescente demanda por aplicações cada vez mais complexas e interativas possa ser atendida é necessário utilizar as ferramentas corretas, que facilitem ao máximo o desenvolvimento de tais sistemas. Com essa finalidade foi criado o JavaServer Faces (JSF), um *framework* para o desenvolvimento de aplicações *web* que reduz significativamente a complexidade de criar e manter uma aplicação dessa natureza por oferecer uma abordagem simples para a criação de suas interfaces de usuário (UI - *User Interface*) e integração com a lógica de negócios. O presente trabalho explora as principais características do JSF e apresenta o *framework* em suas duas versões, especificando quais mudanças e melhorias foram feitas no processo de evolução do mesmo. Ao final, são apresentados alguns trechos de código que demonstram na prática essas mudanças, ressaltando a facilidade e a rapidez que promovem no desenvolvimento.

Palavras-chave: TI. Aplicação *web*.

ABSTRACT

WELZEL, Jonathan Nascimento**. O *Framework* JavaServer Faces**. 2011, 102f. Trabalho de Conclusão do Curso Superior de Tecnologia em Desenvolvimento de Sistemas de Informação. Universidade Tecnológica Federal do Paraná, Medianeira, 2011.

In order to supply the growing demand for this kind of application the right development tools must be used, and for that purpose the JavaServer Faces (JSF) framework was created. JSF is designed to significantly ease the burden of writing and maintaining applications of this nature by offering a simple approach to user interface (UI) design and integration between business logic. This paper depicts the main features and benefits offered by JSF as well as gives a detailed overview of its present and previous version specifically approaching the differences between them. By the end, these differences will be shown using code snippets that intend to demonstrate how they can help to speed up the development process.

Keywords: IT. Web application.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

API	-	Application Programming Interface
HTML	-	Hypertext Markup Language
HTTP	-	Hypertext Transfer Protocol
IDE	-	Integrated Development Environment
JEE	-	Java Enterprise Edition
JSF	-	JavaServer Faces
JSP	-	JavaServer Pages
JVM	-	Java Virtual Machine
LAN	-	Local Area Network
MVC	-	Model-View-Controller
POJO	-	Plain Old Java Object
REST	-	Representational State Transfer
SGBD	-	Sistema Gerenciador de Banco de Dados
SOAP	-	Simple Object Access Protocol
TCP	-	Transmission Control Protocol
TI	-	Tecnologia da Informação
UI	-	User Interface
URL	-	Uniform Resource Locator
W3C	-	World Wide *Web* Consortium
WWW	-	World Wide *Web*
XML	-	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

No início dos anos 60, a computação era baseada em um modelo centralizado, em que um grande computador central (*mainframe*) possuía um enorme poder de processamento que era compartilhado entre vários computadores com pouco ou nenhum poder de processamento, chamados de “terminais burros”, conectados a ele. Nas décadas seguintes, a computação centralizada foi perdendo espaço com o surgimento e a proliferação da *Internet*. Com a abertura da *Internet* para a exploração comercial, o uso de redes de computadores se popularizou rapidamente como um meio de trocar informações (OYA, 2006).

Com o crescimento da *Internet* e as possibilidades que ela trouxe consigo, surgiu também a demanda por aplicações que pudessem explorar esse novo território distribuído utilizando as tecnologias oferecidas por ela e ao mesmo tempo contornando de maneira satisfatória suas limitações. Foi a partir dessas necessidades que surgiram as aplicações *web*, verdadeiros sistemas que são acessados através de uma rede qualquer, tal como a própria *Internet* ou uma rede local (intranet), geralmente utilizando um *browser* como ambiente de execução, que por sua vez opera sobre o famoso protocolo de comunicação HTTP (*Hypertext Transfer Protocol*) para efetuar a troca de dados.

O problema é que os *browsers* não foram criados para executar aplicações mas sim para exibir documentos ou informações disponíveis na *world wide web* que poderiam ser interligadas umas às outras de maneira estática utilizando o HTML (*Hypertext Markup Language*). Além do mais, o próprio modelo de comunicação requisição-resposta do protocolo HTTP que é utilizado pelos *browsers* não oferece suporte a esse tipo de funcionalidade. *Browsers* e protocolos a parte, o fato é que desenvolver aplicações *web* de qualidade, até mesmo aplicações simples, é algo complexo e isso se torna claro quando são comparadas com aplicações *desktop*. Em uma aplicação *web* é necessário gerenciar vários tipos diferentes de recursos, tais como páginas, arquivos de configuração, imagens, conexões com banco de dados e muitas outras. Os usuários podem estar usando tipos diferentes de *browsers* que se comportam de maneiras diferentes, utilizando larguras de banda das mais variadas, ou até mesmo sistemas operacionais distintos (MANN, 2005).

Para superar essas dificuldades e transformar um ambiente estático em um ambiente dinâmico onde aplicações complexas pudessem ser executadas, várias soluções vieram à tona em forma de APIs (*Application Programming Interface*), inicialmente escritas em C ou Perl (FORD, 2004). Foi somente no final da década de 1990 que a linguagem Java deu seus primeiros passos no mundo do desenvolvimento *web*, quando em 1997 a hoje extinta Sun Microsystems lançou a versão 1.0 da especificação Java Servlets (ORACLE, 2010).

Com o passar do tempo e o avanço da tecnologia novas APIs foram criadas a partir daquelas que já existiam e devido às necessidades, tanto dos desenvolvedores como dos usuários por aplicações cada vez mais complexas, seguras, integradas mas que ao mesmo tempo não exigissem um tempo excessivo de desenvolvimento ou investimentos exorbitantes, essas ferramentas deixaram o patamar de meras bibliotecas de funcionalidades para se tornarem *frameworks*.

Frameworks são extremante comuns nos dias de hoje e por um bom motivo: eles simplificam o desenvolvimento de aplicações *web*. Como a maioria dos *frameworks* para desenvolvimento de aplicações *web* em Java, o JSF reforça uma clara separação entre a camada de apresentação e a camada de lógica de negócios. Porém, ele foca mais na parte de interface com o usuário e pode ser facilmente integrado a outros *frameworks*, como o Struts por exemplo (MANN, 2005).

1.1 OBJETIVO GERAL

O objetivo do presente trabalho é desenvolver um estudo sobre o *framework* JavaServer Faces (JSF), que é parte da plataforma Java Enterprise Edition (JEE), para mostrar as diferenças entre as versões 1 e 2 do mesmo.

1.2 OBJETIVOS ESPECÍFICOS

- Descrever de maneira breve a história das aplicações *web*, como elas surgiram e vieram a ser o que são hoje.

- Fornecer uma visão geral do JSF, explorando os recursos que ele oferece e explicando seu funcionamento interno.
- Apresentar as diferenças entre as versões 1 e 2, quais necessidades levaram esse *framework* à sua segunda versão e como certas coisas são feitas em cada uma delas.
- Utilizando código Java, demonstrar na prática as diferenças entre as versões abordadas e como elas agilizam o processo de desenvolvimento.
- Demonstrar o suporte e os recursos que as principais IDEs, segundo Marty Hall, oferecem à versão 2 do JSF.

1.3 JUSTIFICATIVA

Existe no mercado uma variedade enorme de ferramentas para auxiliar no desenvolvimento de aplicações *web* Java. Porém, poucas são capazes de oferecer uma solução tão completa e, ao mesmo tempo, tão simples como o JavaServer Faces (GEARY, 2009). Completa pois seus recursos trazem benefícios para as diversas camadas da aplicação e simples devido à maneira como ele facilita a criação de páginas complexas através do uso de componentes (LOPES, 2010). O JSF também pode ser facilmente integrado com outras tecnologias populares disponíveis, incluindo Hibernate, Seam, Spring, Jasper Reports para citar algumas.

Além disso o JavaServer Faces ainda conta com o apoio de grandes empresas do mercado e é utilizado em produção por corporações como Credit Suisse, Federal Express, Deutsche Bank, Apple Computer Inc., Garmin, Virgin Online e muitas outras segundo levantamento realizado pela Sun Microsystems em 2009 (JAVA.NET, 2009). O JSF também conta com o apoio de várias empresas de renome na área da tecnologia como IBM, SAP, BEA Systems, Novell e RedHat, o que lhe garante um futuro promissor e muita credibilidade como ferramenta para o desenvolvimento de aplicações *web* (JCP, 2009).

Por esses motivos, ao longo dos anos este *framework* se tornou uma das alternativas mais utilizadas pelos desenvolvedores ao redor do mundo e várias melhorias foram introduzidas para torná-lo ainda mais produtivo e amigável, tanto para usuários como para desenvolvedores de sistemas (NUNES & MAGALHÃES,

2010). Porém, para aproveitar ao máximo os recursos e benefícios oferecidos pela ferramenta é necessário explorar cada um deles bem como ter uma compreensão do funcionamento interno da mesma, além de ter à disposição uma ferramenta que seja capaz de oferecer o melhor suporte a esses recursos (HALL, 2004). Para comprovar os benefícios do JavaServer Faces no desenvolvimento de aplicações *web* e explorar as opções de IDEs, nada melhor do que mostrar na prática o poder deste *framework*.

1.4 ESTRUTURA DO TRABALHO

O presente trabalho é constituído de cinco capítulos. O primeiro apresenta um panorama da história das aplicações *web* Java, como elas surgiram, como funcionavam e como elas eram criadas. Além disso aborda como o JavaServer Faces surgiu nesse meio, faz uma introdução dos objetivos do presente trabalho e uma breve apresentação do tema a ser tratado nos demais capítulos e ao final descreve as características da tecnologia Java Servlet e como ela está ligada ao JSF.

No segundo capítulo, o JSF é introduzido de maneira mais aprofundada, partindo de sua história e depois explicando como ele funciona internamente, quais tecnologias compõem o *framework*, bem como o seu ciclo de vida trabalha. É abordada a versão 2 do JSF e suas novidades. Neste mesmo capítulo ainda é feita uma comparação mostrando na prática as diferenças entre a versão 1 e a versão 2 do JSF, como as melhorias oferecidas pela nova versão são aproveitadas na prática e como algumas das principais IDEs disponibilizam esses recursos.

No terceiro capítulo é desenvolvido um estudo de caso que foi utilizado para constatar os benefícios apresentados. Esse estudo foi feito através do desenvolvimento de uma aplicação *web* com JSF.

No quarto capítulo são apresentadas as conclusões finais referentes a todo o conteúdo apresentado durante o desenvolvimento do trabalho. Também são abordadas ideias para trabalhos futuros relacionados ao desenvolvimento de aplicações *web* Java utilizando *frameworks*.

2 HISTÓRIA DAS APLICAÇÕES *WEB* JAVA

2.1 DEFINIÇÃO DE APLICAÇÃO *WEB* JAVA

Na linguagem Java uma aplicação *web* é definida como sendo um conjunto de *servlets*, páginas HTML, classes e outros recursos que são sistematicamente empacotados e que podem ser executados em vários *containers* de vários fabricantes (ORACLE, 2011). Um servidor que implementa a tecnologia Java servlets e JavaServer Pages converte uma requisição em um objeto *HTTPServletRequest* que é entregue a um componente da aplicação *web* que seja capaz de interpretar e processar os dados da requisição de acordo com as regras de negócio da aplicação. Esse objeto pode ainda interagir com JavaBeans no servidor ou acessar o banco de dados a fim de gerar conteúdo dinâmico. Com os dados resultantes do processamento no servidor, o componente responsável por processar o objeto gerado a partir da requisição pode então gerar um objeto *HTTPServletResponse* que é convertido em uma resposta HTTP pelo servidor e enviado para o cliente. A Figura 1 ilustra esse processo.

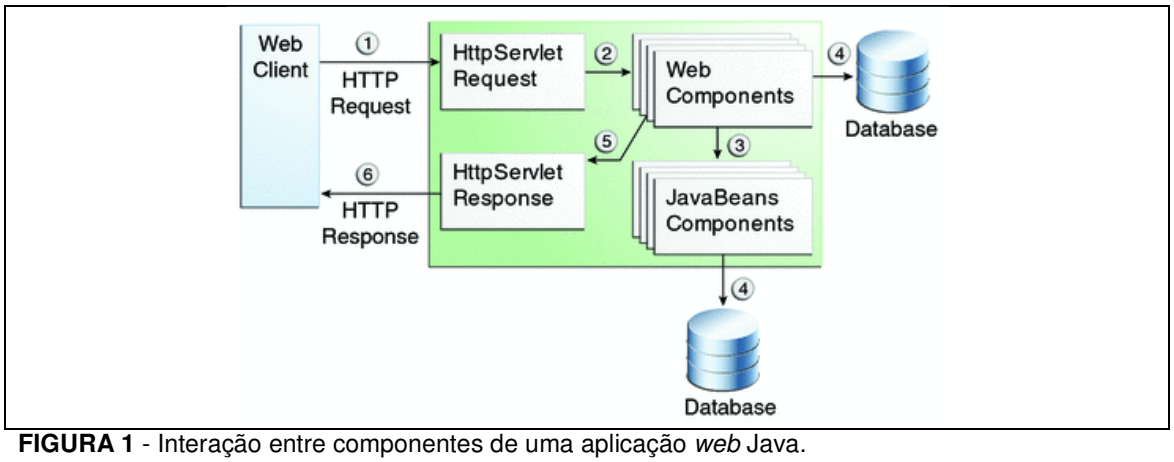

FIGURA 1 - Interação entre componentes de uma aplicação *web* Java.
FONTE: ORACLE (2011).

Esses componentes e recursos são gerenciados por um *web container*, que oferece à aplicação serviços de processamento de requisições HTTP, segurança, concorrência, e gerenciamento de ciclo de vida, além de expor as APIs Java para

esses objetos. Uma aplicação JSF se assemelha a qualquer outra aplicação *web* Java pois utiliza os mesmos artefatos e tecnologias suportadas pelos *web containers* disponíveis. Uma aplicação JSF típica contém os seguintes itens (ORACLE, 2011):

- Um conjunto de páginas *web* onde os componentes gráficos são declarados
- Um conjunto de *tags* que disponibilizam seus componentes para as páginas
- Um conjunto de *backing beans* que são componentes JavaBeans que definem as propriedades e funções dos componentes nas páginas
- Um arquivo denominado *web.xml* que descreve as propriedades de instalação (*deploy*) da aplicação e contém configurações específicas referentes ao JSF, tais como mapeamento do *FacesServlet*, parâmetros globais da aplicação e de *logging*
- Opcionalmente, um ou mais arquivos de configuração de recursos, tais quais o *faces-config.xml*, que pode ser usado para definir as regras de navegação entre as páginas da aplicação, configurar *backing beans* e outros objetos personalizados, como componentes customizados
- Opcionalmente um conjunto de objetos customizados, tais como componentes de interface com o usuário, validadores e *listeners* criados pelo desenvolvedor
- Um conjunto de *tags* customizadas para representar esses objetos nas páginas *web*

A Figura 2 mostra um exemplo de como uma aplicação JSF 2 é estruturada.

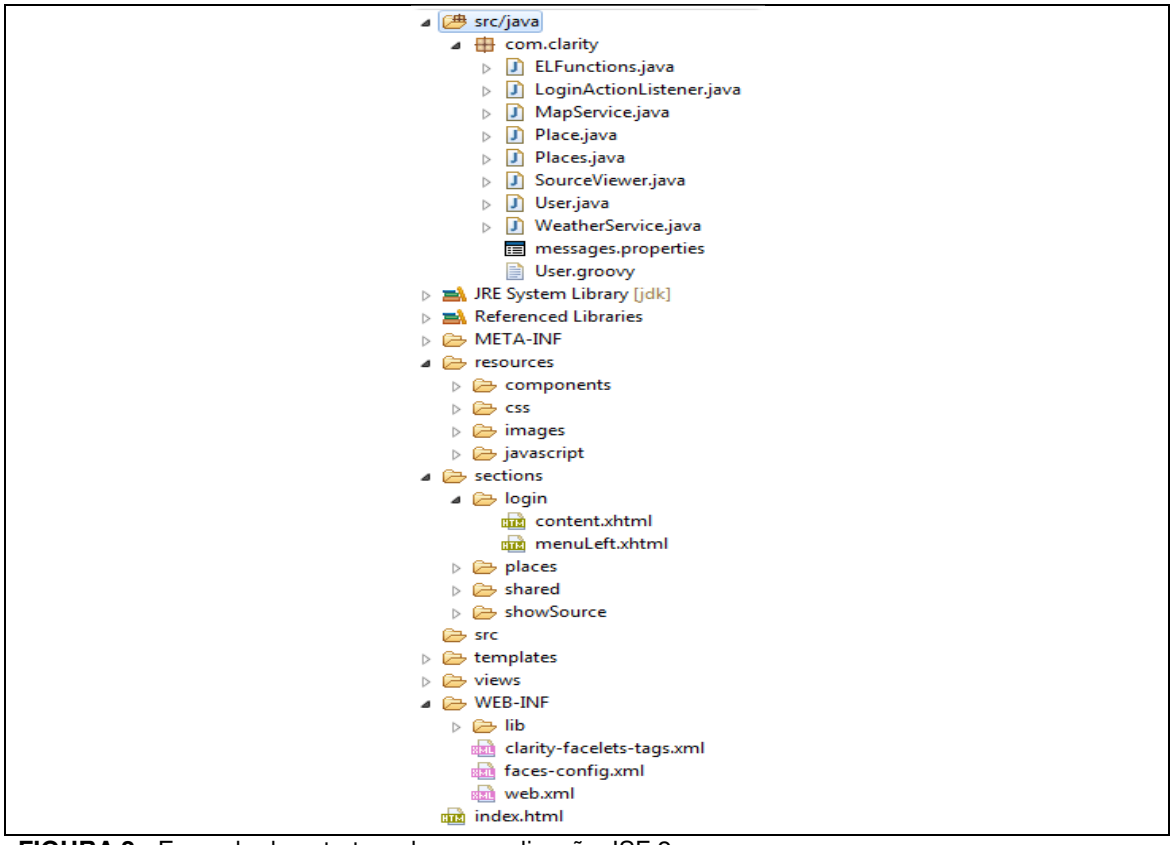

FIGURA 2 - Exemplo de estrutura de uma aplicação JSF 2.

2.2 O INÍCIO DO JAVA NA *WEB*

A *World Wide Web* é um exemplo perfeito de como uma ideia simples (páginas ligadas umas às outras via HTML) pode se transformar em algo rico e extraordinário. Criado com o propósito de exibir páginas estáticas, esse meio logo se expandiu de forma que pudesse oferecer conteúdo dinâmico (FORD, 2004). Esses passos iniciais foram dados com a ajuda de linguagens como *C* e *Perl*, mas como o passar do tempo e o avanço da tecnologia novas APIs vieram a existir, cada uma trazendo novos recursos e aprimorando o que as anteriores já faziam. Novas APIs surgem porque os desenvolvedores encontram limitações e foram essas limitações que colocaram a linguagem Java em uma posição de destaque no que diz respeito ao desenvolvimento de aplicações *web* dinâmicas, primeiro com *servlets* e mais tarde com JSP (FORD, 2004).

Apesar da linguagem Java ter começado como uma linguagem voltada exclusivamente para o desenvolvimento de aplicações *desktop* tradicionais e de *applets*, conforme os desenvolvedores foram se familiarizando com ela e conhecendo seus benefícios, ela rapidamente expandiu suas fronteiras chegando ao território dos sistemas distribuídos e dos sistemas *web* (FORD, 2004).

Os primeiros passos da linguagem Java neste novo território foram dados muito antes da criação do JavaServer Faces através da tecnologia Java Servlet. Isso se deve ao fato de essa tecnologia utilizar classes Java para processar as requisições HTPP de forma que elas possam interagir com a aplicação *web* no servidor e essa por sua vez construir as respostas para o cliente de maneira dinâmica.

No início, os desenvolvedores usavam *servlets* para gerar o conteúdo dinâmico de suas aplicações *web*, mas os gerentes de projeto logo perceberam que os talentos de um desenvolvedor, na maioria dos casos, não eram o suficiente para que eles pudessem criar uma interface com o usuário atraente. Os profissionais mais qualificados para executar esse tipo de tarefa geralmente são aqueles que se especializam na área de *design* gráfico e *layouts*. Para que ambas as partes de um sistema, a lógica e a gráfica, pudessem obter resultados satisfatórios, cada uma dessas tarefas foi delegada a setores diferentes de uma empresa: de uma lado os desenvolvedores criando a lógica e as funcionalidades do programa e do outro os profissionais formados em artes desenhando a interface com o usuário. O problema era que depois, essa interface feita em HTML precisava ser integrada aos *servlets* dos desenvolvedores que geravam o conteúdo dessas páginas.

Porém, um vez que essa tarefa era concluída os problemas ainda estavam longe de chegar ao fim. Se, por exemplo, ao longo do desenvolvimento do projeto fosse necessário fazer alguma mudança na interface da aplicação, o que é muito comum, os *designers* tinham que remodelar todas a parte visual e repassá-la aos desenvolvedores. Estes por sua vez tinham que não somente incorporar a nova interface aos *servlets* que eles já haviam previamente criado, mas também fazer isso de uma maneira sistemática e muito cuidadosa. Tudo isso para não comprometer o funcionamento da aplicação e consequentemente fazer com que o tempo gasto no desenvolvimento até então fosse simplesmente desperdiçado.

O maior problema ao se utilizar *servlets* no desenvolvimento de aplicações *web* com certeza é o fato de que o HTML acaba se misturando muito com o código

Java e isso tem uma série de más consequências para qualquer sistema. Para citar algumas: a legibilidade e compreensão do que está sendo executado e gerado pelos *servlets* se torna complexa demais e os *servlets* acabam possuindo um número exorbitante de linhas por causa do HTML que eles têm que gerar (SERVLETS.COM, 2011). A essência desse modelo de desenvolvimento vai totalmente contra o princípio da separação de interesses inicialmente criado por Edsger W. Dijkstra. Esse princípio, quando aplicado à ciência da computação, determina que o desenvolvedor deve focar-se em cada uma das funções do sistema de maneira independente, de tal modo que ao alterar uma delas, as demais não tenham o seu funcionamento comprometido. Dessa forma o resultado é um sistema altamente extensível e de fácil manutenção (READE, 1989).

Para amenizar os problemas causados pela injeção de HTML nos *servlets* alguns desenvolvedores passaram a utilizar *tags* em suas páginas que informavam para o *servlet* que conteúdo dinâmico deveria ser inserido ali e onde ele deveria ser inserido. Para renderizar uma página o *servlet* tinha que processar todas as *tags* encontradas nela antes de poder enviá-la de volta para o cliente. Uma ótima alternativa, já que separava o conteúdo dinâmico da interface com o usuário, porém essa prática demandava uma grande quantidade de recursos de processamento.

Foi isso que levou ao desenvolvimento da tecnologia *JavaServer Pages*, que contornava o problema do custo de processamento da seguinte maneira: as páginas JSP juntamente com suas *tags* eram processadas e transformadas em *servlets* e em seguida executadas, porém esse trabalho todo era realizado somente uma única vez. A Figura 3 ilustra esse processo.

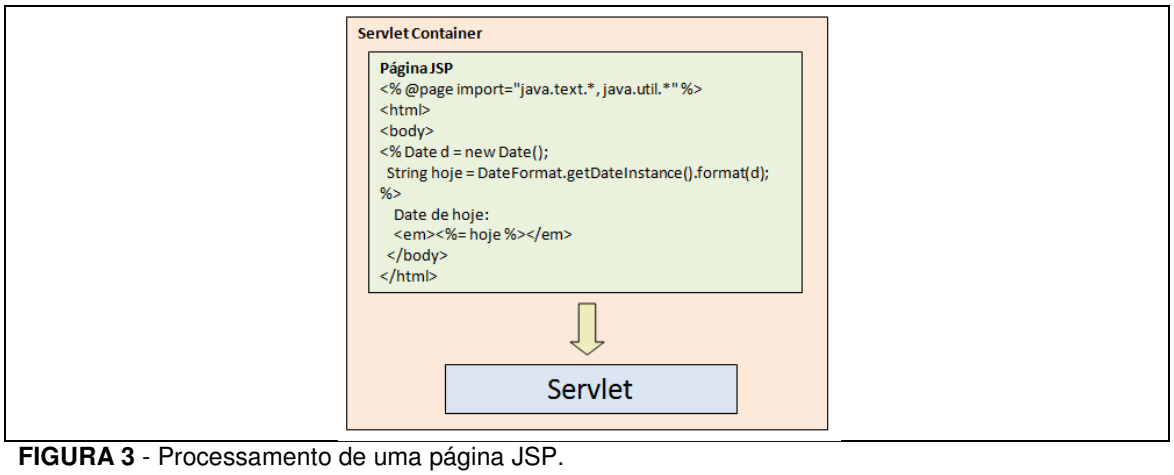

FIGURA 3 - Processamento de uma página JSP.

Através do uso desta tecnologia era possível inserir a lógica da aplicação diretamente nas páginas de uma maneira mais elegante. Infelizmente isso também acabou gerando uma série de problemas pois esse recurso incentivava a utilização do código funcional da aplicação juntamente com a interface com o usuário, o que fazia com que a manutenção dessas páginas se tornasse um verdadeiro pesadelo. Com o passar do tempo porém, os desenvolvedores perceberam que nas páginas JSP não era o lugar do código Java também e deixaram de depender tanto delas para isso. A ideia de que um usuário tivesse que enviar a sua requisição para uma página também começou a parecer estranha já que uma página não serve para processar os dados, mas sim para apresentá-los.

Então, para estruturar melhor uma aplicação e criar uma separação mais clara entre as suas diferentes camadas foi concebido o Modelo 2, uma arquitetura de desenvolvimento para a *web* que utiliza, mediante alguns ajustes, o padrão de projeto *Model-View-Controller* (MVC). No Modelo 2 os *JavaBeans* representam o modelo, a parte dos dados e da lógica da aplicação. As páginas JSP (ou qualquer outra tecnologia que renderize conteúdo HTML de forma dinâmica) são a visão, que é responsável pela interação do usuário com a aplicação e os *servlets* são os controladores, que fazem a integração entre o modelo e a visão. Essa arquitetura garante que a aplicação tenha um acoplamento fraco, o que reduz as dependências entre as suas diferentes camadas (MANN, 2005). Uma ilustração representando os elementos típicos do Modelo 2 é mostrada na Figura 4.

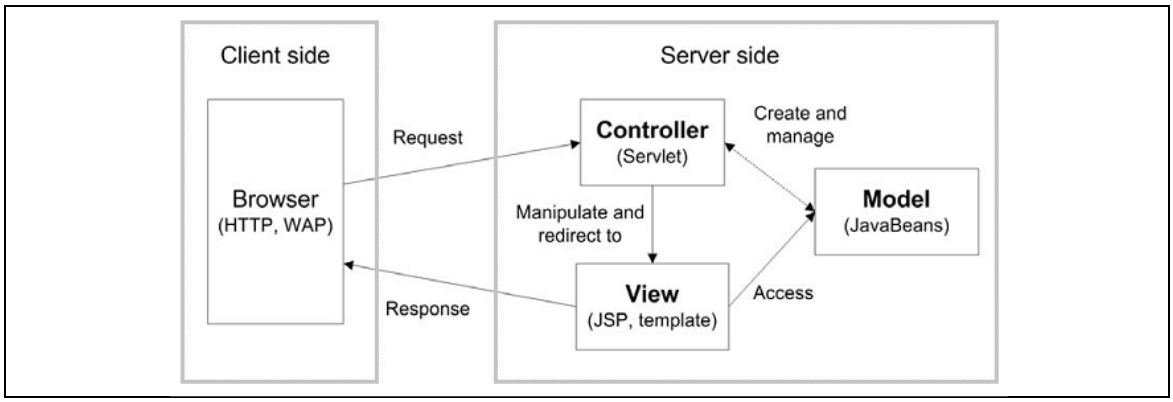

FIGURA 4 - Arquitetura do padrão de projeto conhecido por Modelo 2.
FONTE: MANN (2005).

Um dos mais importantes princípios dessa arquitetura é que a lógica de programação em hipótese alguma deve se misturar à camada de visão. A função da página JSP é unicamente apresentar os dados, dados estes que são representados pelo modelo, os *JavaBeans*, que por sua vez não devem possuir conhecimento algum de que fazem parte de uma aplicação *web*. Para que isso aconteça o controlador é quem interage com as outras duas camadas, fazendo com que cada parte se mantenha especializada e focada na função que deve desempenhar dentro da aplicação. Sendo assim, se a página JSP tiver algum erro, isso não afetará o código da aplicação e nem o modelo. Se houver um erro no modelo, isso não será um problema para a página JSP e nem para a lógica de programação da aplicação. Essa separação também permite que cada parte seja testada de forma separada e que terceiros possam trabalhar nas camadas independentemente. Estes são alguns dos principais benefícios que tornam o padrão MVC e as suas variações tão atraentes aos olhos da maioria dos *frameworks*, e o JavaServer Faces não é uma exceção.

2.2.1 Surgimento dos *Frameworks*

O Modelo 2 é um excelente padrão para desenvolver aplicações *web*. Mas o fato é que com o passar do tempo os desenvolvedores passaram a observar que embora cada uma das aplicações que desenvolviam fosse diferente uma da outra, todas possuíam muitas coisas em comum que poderiam ser reutilizadas em outros projetos. Aprenderam que muitas dessas partes são genéricas e que podem ser combinadas para formar outras partes genéricas ainda maiores e com mais funcionalidades. Por exemplo, os *servlets* controladores das aplicações que utilizam o Modelo 2 têm muitos recursos em comum que se construídos de uma certa maneira promovem a reusabilidade. Os padrões de projeto facilitam a construção desses artefatos reutilizáveis (FORD, 2004) e representam a base de qualquer *framework*, inclusive do JavaServer Faces.

A partir do momento que essas partes genéricas são combinadas e assim passam a formar um conjunto coeso de ferramentas, pode-se dizer então que um *framework* acabou de ser originado. Um *framework* nada mais é do que um conjunto

de classes e de outros elementos de apoio, que de alguma maneira estão relacionados, cuja finalidade é facilitar o desenvolvimento oferecendo recursos pré construídos. A partir desses recursos é que se constrói então algo mais específico para cada aplicação, pois a essência de um *framework* é oferecer uma infra-estrutura básica, porém sólida. É sobre essa estrutura que novas aplicações podem ser desenvolvidas de uma maneira mais organizada e padronizada a fim de diminuir o seu tempo de desenvolvimento e agregar funcionalidade ao produto final, entre vários outros benefícios.

Atualmente, a variedade de *frameworks* disponíveis para desenvolvimento de aplicações *web* em Java é imensa. Vários deles oferecem recursos que facilitam e muito o desenvolvimento dessas aplicações. Mesmo assim, ainda não conseguem abstrair a natureza essencial requisição/resposta do protocolo HTTP e é justamente essa uma das principais funções de um *framework*: abstração, quanto mais alto o nível melhor. Dessa maneira, os desenvolvedores têm mais tempo livre para focarem nas regras de negócio da aplicação e não se preocuparem tanto com a parte mais baixo nível. Quem cuida desse trabalho sujo é o próprio *framework*.

2.3 SERVLETS

O protocolo HTTP é um ótimo meio de distribuição de conteúdo estático e os servidores *web* são muito bons nisso. Mas criar conteúdo dinâmico é algo que exige algumas linhas de código, ou várias dependendo do caso. Mesmo sendo um protocolo simples, o HTTP exige uma certa quantidade de trabalho para que aplicações possam ser executadas através dele. É justamente para isso que a API *Java Servlet* serve: oferecer uma visão simplificada do ambiente *web*, mas de uma perspectiva totalmente orientada a objetos a fim de facilitar o desenvolvimento de aplicações nesse meio. Requisições e respostas HTTP são encapsuladas em objetos, o que acaba tornando possível a comunicação e o acesso, bem como a manipulação, do fluxo de dados na comunicação entre servidor e cliente (ORACLE, 2010).

O poder dessa API vem do fato de que seu uso é baseado na plataforma Java e na sua interação com o *servlet container*. Isso porque Java oferece ao

desenvolvedor uma API robusta, orientada a objetos, que não se atém a uma única plataforma, que é estaticamente tipada, possui um excelente coletor de lixo (*garbage collector*) e todos os recursos de segurança oferecidos pela JVM. Para complementar tudo isso, os s*ervlet containers* oferecem gerenciamento de ciclo de vida, um processo integrado para compartilhar e administrar os recursos da aplicação e também se comunicar com servidores *web*.

Outro benefício muito importante oferecido pela API *Java Servlet* é a sua capacidade de proteger o desenvolvedor das complexidades envolvidas no manuseio da sessão e dos *cookies*. Como já foi mencionado anteriormente, o protocolo HTTP não oferece um contexto de conversação entre o cliente e o servidor que vá alem do tempo de vida de uma única requisição. Para isso foi criado o conceito de sessão de usuário, da qual os *cookies* também fazem parte. Dessa maneira é possível manter um estado de comunicação mais longo entre o cliente e o servidor pois o cliente fica responsável por armazenar pequenas quantidades de dados que o servidor então usa para identificá-lo (MANN, 2005).

A API *Java Servlet* encontra-se atualmente na versão 3.0 que foi lançada em dezembro de 2009 e faz parte da especificação da plataforma *Java EE 6*. Essa última versão trouxe vários recursos e benefícios para os desenvolvedores, tais como plugabilidade, maior facilidade de desenvolvimento, *servlets* assíncronos, segurança e *upload* de arquivos (JAVAWORLD, 2009).

2.3.1 Definição de Servlet

Um *servlet* é uma classe Java que é executada no servidor e está em conformidade com a API *Java Servlet*, o que a torna capaz de interagir com requisições HTTP (ORACLE, 2011). Essa classe oferece os meios para que a aplicação hospedada no servidor se comunique com o cliente, pois um *servlet* é capaz de interagir com a aplicação como qualquer outro objeto que faça parte do seu contexto. *Servlets* podem acessar dados no servidor, objetos, renderizar conteúdo HTML puro com conteúdo dinâmico obtido no servidor como resultado de algum processamento, redirecionar requisições, formatar as respostas HTTP para os

clientes, gerenciar fluxos de dados (enviar/receber arquivos), *cookies*, manipular a sessão do cliente, reforçar a segurança da aplicação, entre muitos outros recursos.

Servlets geralmente usam herança (estendem `HttpServlet`) para sobrescrever os métodos que respondem às requisições HTTP do tipo *get* ou *post*. Ambos os métodos (`doGet` e `doPost`) recebem dois argumentos: um `HttpServletRequest` e um `HttpServletResponse`. O primeiro permite obter todos os dados que estão chegando até o servidor através da requisição, tais como valores de campos de formulários, cabeçalhos da requisição HTTP, o *hostname* do cliente, entre outros. Já o segundo argumento permite especificar o conteúdo da resposta que será enviada pelo servidor de volta ao cliente, tais como códigos de status HTTP (401, 403, 404, etc), cabeçalhos (`Content-Type`, `Set-Cookie`), além de prover acesso ao `PrintWriter` que é usado para gerar conteúdo escrito (NOVOCODE, 2011). O Quadro 1 mostra uma classe *servlet* comum com alguns de seus métodos herdados.

```java
package servlets;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * @author Jon
 *
 */
@SuppressWarnings("serial")
public class HelloWorldServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        // TODO Auto-generated method stub
        super.doGet(req, resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        // TODO Auto-generated method stub
        super.doPost(req, resp);
    }
}
```

QUADRO 1 - Estrutura básica de um *servlet*.

2.3.2 Servlet Container

A boa performance dos *servlets* pode ser atribuída diretamente ao *servlet container*. Um *servlet container*, ou simplesmente *container*, ou ainda JSP *container* é um *software* que gerencia o ciclo de vida dos *servlets*. Este *software* é responsável por interagir com um servidor *web* a fim de manipular as requisições e encaminhá-las para os *servlets* para que eles possam gerar uma resposta.

A definição oficial do que é um *container* é descrita de maneira completa pelas especificações JSP e *Servlet*. Ao contrário da maioria das tecnologias proprietárias, as especificações JSP e *Servlet* só definem um padrão das funcionalidades que devem ser implementadas por um *container* (WESLEY, 2004). Esse fato permite que cada fabricante crie sua própria implementação do *servlet container*, o que coloca várias opções à disposição do desenvolvedor na hora de escolher qual ferramenta utilizar para gerenciar os *servlets* de sua aplicação.

2.3.3 Ciclo de Vida de um Servlet

A chave para se compreender as funcionalidades de baixo nível oferecidas pelos *servlets* é familiarizar-se com seu ciclo de vida. Este ciclo governa o ambiente *multi thread* na qual os *serlvets* são executados e oferece uma visão simplificada dos mecanismos que estão disponíveis para os desenvolvedores para o compartilhamento de recursos do servidor (WESLEY, 2004).

Um *servlet* é gerenciado através de um ciclo de vida bem definido que determina como ele é inicializado, como ele processa as requisições vindas dos clientes e por fim, como ele é destruído. Esse ciclo de vida é representado na API pelos métodos `init`, `service` e `destroy` da interface `javax.servlet.Servlet` que deve ser obrigatoriamente implementada, direta ou indiretamente, através das classes abstratas `GenericServlet` ou `HttpServlet` (MORDANI, 2009). A Figura 6 ilustra esses três métodos que representam o ciclo de vida de um *servlet*.

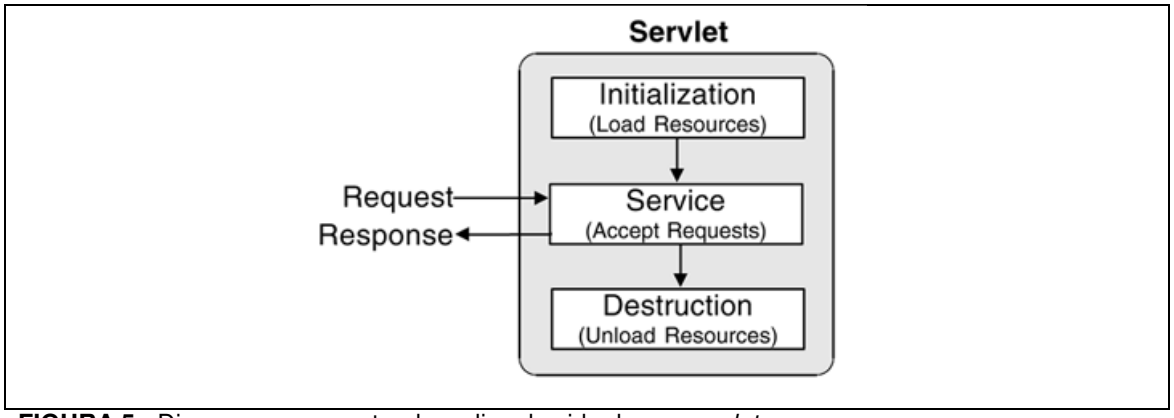

FIGURA 5 - Diagrama representando o clico de vida de um *servlet*.
FONTE: WESLEY (2004).

O *servlet container* é responsável pelo carregamento e inicialização do *servlet*. Isso pode ser feito de forma automática assim que o *container* é inicializado ou sob demanda, onde o *servlet* só é inicializado a partir do momento que ele for solicitado por uma requisição. Quando um *container* é inicializado, ele carrega consigo as classes *servlet* utilizando os métodos convencionais de *class loading* oferecidos pela linguagem Java. Após carregar as classes *servlet*, o *container* então instancia elas para que possam ser usadas.

Após ser instanciado, o objeto *container* deve então inicializar o *servlet* antes que ele possa atender as requisições dos clientes. Essa fase do ciclo de vida representa a criação e o alocamento dos recursos necessários para que o *servlet* possa desempenhar sua função junto à aplicação. Isso acontece para que o *servlet* seja capaz de ler dados de configuração de persistência, executar operações que tenham um custo maior de processamento (tais como conexões JDBC) e outras atividades de execução única. O *container* faz essa inicialização chamando o método init da interface *Servlet*. Durante sua inicialização o *servlet* pode disparar uma `UnavailableException` ou ainda uma `ServletException`. Nesse caso, o *servlet* é impedido de ser ativado e em seguida é descarregado pelo *servlet container*.

Assim que um *servlet* é inicializado com sucesso ele pode ser usado pelo *container* para manipular as requisições do cliente através do método `service` que é utilizado por cada requisição. Começa então a fase de serviço, que representa todas as interações entre o *servlet* e as requisições que ele irá atender até que seja destruído. Essas requisições são representadas por objetos do tipo

`ServletRequest`. As requisições são atendidas com respostas geradas a partir de um objeto do tipo `ServletResponse`. Esses objetos são passados como parâmetros para o método `service` da interface `Servlet`. No caso de se tratar de uma requisição HTTP, os objeto que a representa é do tipo `HttpServletRequest` e no caso de uma resposta é do tipo `HttpServletResponse`. Se caso alguma exceção for disparada durante o processamento de uma requisição, o *container* fica responsável por descartar a mesma e retornar a devida mensagem de status HTTP.

A fase de destruição de um *servlet* representa sua remoção do *container*. O tempo de vida de um *servlet* dentro do *container* pode variar desde milisegundos até meses e anos dependendo do tempo de atividade do ambiente na qual ele está contido. Quando o *container* determina que um *servlet* deve ser removido de serviço, ele chama o método `destroy` da interface `Servlet` para que assim o mesmo possa liberar quaisquer recursos que esteja utilizando no momento. Essa ação pode ser tomada em função de uma limpeza para liberar mais memória no *container* ou até mesmo em decorrência do seu desligamento.

Porém, para que o método `destroy` possa ser chamado pelo *container* ele precisa antes permitir que todas as *threads* que estejam rodando no método `service` completem suas execuções, ou até que o tempo de limite de espera seja atingido. Uma vez que o método destroy é chamado para um *servlet* o *container* não pode mais direcionar requisições para ele. Se por acaso isso for necessário o *servlet* deve ser instanciado novamente através dos passos do ciclo de vida descritos até aqui. Assim que o *servlet* chega ao fim do seu ciclo de vida, sua instância deve ser liberada pelo *container* para que o *garbage collector* possa fazer seu trabalho (MORDANI, 2009).

2.4 SERVLETS E JAVASERVER FACES

A tecnologia *Servlet* oferece vários benefícios e simplifica muito o desenvolvimento de aplicações *web* em Java o que a torna uma base muito sólida sobre a qual elas podem ser construídas. Mas o JavaServer Faces leva essas funcionalidades e benefícios a um nível mais alto, onde os desenvolvedores não têm

que se preocupar em manipular manualmente as requisições e respostas HTTP, que na verdade são propriedades que pertencem ao protocolo HTTP que serve como meio de transmissão de dados. Desenvolvedores não deveriam ter que se preocupar com detalhes de tão baixo nível. A figura abaixo ilustra como a tecnologia *Java Servlet* se encaixa no escopo de uma aplicação JSF.

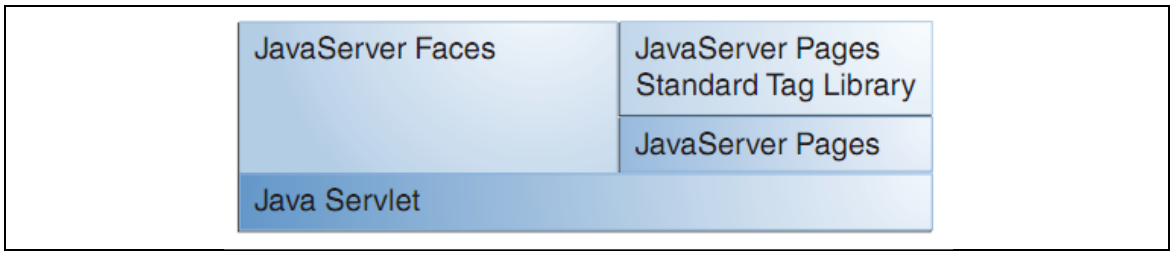

FIGURA 6 - Tecnologias Java para aplicações *web*.
FONTE: BERGSTEN (2004).

As aplicações JSF utilizam um *servlet*, convencionalmente chamado de `FacesServlet`, como seu ponto de acesso principal. Esse *servlet* atua como um *front controller* através da qual as requisições são mapeadas para a aplicação JSF e além disso ele também é responsável por processar o ciclo de vida dessas requisições.

3 JAVASERVER FACES

Ao longo dos últimos anos a plataforma Java se estabeleceu como uma das principais tecnologias para o desenvolvimento de aplicações *web* (JAVA.NET, 2009). Java Servlets e JavaServer Pages ainda hoje são algumas das tecnologias Java mais utilizadas na criação de aplicações *web* escaláveis, robustas e com uma interface gráfica atraente. Mas conforme as aplicações *web* vão se tornando mais complexas, alguns desenvolvedores passam a sentir falta dos dias em que trabalhavam com as tradicionais ferramentas de desenvolvimento gráfico para aplicações *desktop*. Essas ferramentas amenizavam o penoso trabalho de criar uma bela interface com o usuário através da simples tarefa de arrastar os componentes para a tela e em seguida programar suas respectivas ações e propriedades. Servlets e JavaServer Pages ainda têm seu lugar garantido no mercado como alternativas para o desenvolvimento de aplicações *web* em Java. Recentemente a API Java Servlet foi atualizada para a versão 3.0 e JavaServer Pages para a versão 2.2 com o lançamento da plataforma Java EE 6.

Mesmo assim, essas tecnologias ainda não são capazes de oferecer a facilidade e agilidade de desenvolvimento tão prezadas por esses desenvolvedores descontentes pois ainda são forçados a lutar com as dificuldades impostas pelo ambiente distribuído da *web* e suas limitações. Então, a fim de criar uma solução para facilitar a vida dessas pessoas bem como o desenvolvimento de aplicações *web* robustas e sofisticadas, foi lançado em março de 2004 a versão 1.0 do JavaServer Faces (BERGSTEN, 2004).

3.1 LINHA DO TEMPO DO JAVASERVER FACES

Assim como a maioria das outras tecnologias importantes, a criação do JSF foi o resultado de um processo evolutivo combinado com o refinamento das técnicas de programação onde as mais antigas acabaram sendo substituídas por outras mais novas e mais aperfeiçoadas. No caso do JSF, a força que impulsionou essa evolução foi a necessidade de desenvolver interfaces dinâmicas com o usuário para

aplicações *web* de uma maneira mais simples e mais eficaz baseada em uma arquitetura bem projetada que facilitasse a sua manutenção.

O JavaServer Faces foi idealizado através de uma especificação técnica do *Java Community Process* (JCP), com o objetivo de padronizar um *framework* para o desenvolvimento da camada de apresentação em aplicações *web*. Liderada pela Sun Microsystems, esse projeto foi abraçado por várias outras empresas com forte representatividade no setor de tecnologia e de *software*, dentre elas a Oracle, IBM e BEA. Em maio de 2001 o comitê formado por essas empresas votou a favor da criação da especificação da tecnologia que iria padronizar e simplificar o desenvolvimento de interfaces com o usuário para aplicações *web* Java. O resultado disso foi a *Java Specification Request* (JSR) 127 e assim nascia a versão 1.0 do JavaServer Faces (BURNS, SCHALK, 2010). Essa mesma JSR foi responsável tanto pela criação do JSF 1.0 como a do JSF 1.1. Em 2006 a versão 1.2 foi criada através da JSR 252 e em junho de 2009 a versão 2 finalmente foi lançada através da JSR 314.

A versão 1.x trouxe novos conceitos ao desenvolvimento *web*, tais como árvore de componentes, fases do ciclo requisição/resposta, conversores e validadores. Foi também uma versão para amadurecimento. De fato, entre 2004 e 2009, vários projetos *open source* estenderam o JSF trazendo soluções para necessidades ainda não contempladas. Melhorias como validação de campos de formulário sem a necessidade de acessar o servidor, tecnologia, mais simples e com melhor desempenho para descrever páginas JSF, novos conversores, novos validadores, novos escopos para controladores de páginas e principalmente novos componentes interativos (tabelas com ordenação e filtros, caixas de sugestão, menus e gráficos).

A versão 2.0 teve no grupo de especialistas da especificação desenvolvedores da Apache, Oracle, Redhat, IceSoft entre outras empresas bastante envolvidas com JSF. Isso facilitou a introdução das soluções criadas por essas empresas na especificação. Gerenciamento de recursos, Facelets, suporte a Ajax e novos escopos são exemplos de algumas delas (NUNES & MAGALHÃES, 2010).

3.1.1 JSR 127 (JSF 1.0 e 1.1)

A JSR 127 foi a especificação responsável por definir o framework JavaServer Faces em toda a sua amplitude e em 11 de março de 2004 a sua versão final foi lançada (JCP, 2004). Nela foram definidos oito requisitos que descrevem os principais objetivos do JSF e as funcionalidades que fazem parte do *framework*. Esses requisitos foram o foco principal das versões que foram criadas e continuará sendo o foco das versões futuras. São eles (JCP, 2004):

1. Oferecer um *framework* padrão de componentes de interface com o usuário a fim de facilitar a criação de *layouts* de alta qualidade e ao mesmo tempo amenizar as dificuldades de gerenciar a interação entre a camada de apresentação e a camada de negócios da aplicação.
2. Definir um conjunto simples e leve de classes Java para componentes de interface com o usuário, estado de componentes e eventos de entrada. Essas classes são responsáveis por lidar com os aspectos de ciclo de vida, mais especificamente aqueles que estão relacionados ao estado persistente dos componentes dentro de suas respectivas páginas.
3. Oferecer um conjunto padrão de componentes de interface com o usuário, incluindo aqueles que coletam os dados nos formulários HTML. Esses componentes derivam de classes Java e também podem ser usados para criar novos componentes.
4. Prover um modelo JavaBeans para despachar os eventos que o cliente produz através da interação com os componentes da camada de visão para que o servidor possa processá-los.
5. Definir APIs para validação de valores de entrada, incluindo o suporte para validação feita no próprio cliente.
6. Especificar um modelo para internacionalização e recursos de localização para a interface com o usuário.
7. Automatizar a geração de conteúdo que seja apropriado para cada cliente levando em consideração as configurações que cada um possui e o *browser* que utiliza.
8. Gerar conteúdo que esteja de acordo com os padrões de acessibilidade definidos pela *Web Accessibility Initiative* (WAI) de forma automática.

A JSR 127 veio preencher a lacuna existente no leque de tecnologias Java disponíveis para o desenvolvimento *web*. Isso foi necessário porque até então as APIs Java Servlet e JSP, que eram as tecnologias predominantes na época, não ofereciam nenhum recurso avançado relacionado ao tratamento da interface com o usuário (JCP, 2004).

3.1.2 JSR 252 (JSF 1.2)

Essa especificação trouxe várias melhorias que focavam na integração entre JavaServer Faces e JavaServer Pages, porém os recursos e requisitos permaneceram os mesmos tais quais definidos na JSR127. Para atingir seu alvo de alinhar essas duas tecnologias os seguintes temas foram introduzidos como parte da JSR 252 (JCP, 2006):

- A JSR 245 que trouxe a versão 2.1 do JSP.
- A *Unified Expression Language,* que apesar de ser uma especificação separada é usada tanto no JSP como no JSF.
- Correções de problemas referentes à integração entre JSP e *JavaServer Pages Standard Tag Library* (JSTL).
- Possibilidade de utilizar XML para criar os arquivos de configuração ao invés de DTDs.
- Melhorias de segurança para armazenar o estado dos componentes no cliente.
- Correções de *bugs* relacionados a *portlets*.

A versão final da JSR 252 foi lançada no dia 11 de maio de 2006, mesma data em que foi lançada a versão 5 da plataforma Java Enterprise Edition (JCP, 2006), pois a versão 1.2 faz parte da especificação JEE 5.

3.1.3 JSR 314

Por ser uma tecnologia oficial na plataforma Java, o JSF logo alcançou grande sucesso. Além de ser parte da plataforma JEE, seu modelo de desenvolvimento baseado em componentes atraiu bastante atenção por facilitar o desenvolvimento de aplicações complexas baseadas em muitas telas com formulários. Mas ao mesmo tempo, o JSF também era muito criticado devido à quantidade excessiva de configurações, arquivos XML, pela falta de suporte nativo a Ajax e pela complexidade de uso. Alguns *frameworks*, como o JBoss Seam, tentaram resolver esses problemas através de soluções proprietárias de seus fabricantes, sendo que algumas delas foram incorporadas à versão 2 (LOPES, 2010).

Com o lançamento do JSF 2 em julho de 2009 através da JSR 314, muitas novidades foram introduzidas para lidar com os problemas e dificuldades das versões anteriores. Essas inovações foram agrupadas em quatro categorias (JCP, 2009):

1. Facilidade de desenvolvimento.
2. Novos recursos e correções.
3. Performance.
4. Adoção de tecnologias.

Na categoria 1 algumas das melhorias introduzidas foram:

- Agregação de componentes. Permite desenvolver novos componentes JSF com pouco ou nenhum código Java.
- Diminuição da quantidade de arquivos de configuração. Não é mais necessário existir o `faces-config.xml` ou o *`web`*`.xml`. Caso haja necessidade é possível usar anotações para representar os dados de configuração.
- Estágios configuráveis de desenvolvimento da aplicação: *Development* (Desenvolvimento), *SystemTest* (Teste de Sistema), *UnitTest* (Teste Unitário) e *Deployment* (Produção). Cada estágio oferece comportamentos e recursos diferenciados de *logging*, *debugging* e mensagens levando em consideração o estágio atual da aplicação.
- Possibilidade de utilizar anotações para definir uma grande variedade de componentes dentro de uma aplicação, não somente a parte de configuração.

- Eliminar o atraso causado pela necessidade de fazer o *deploy* da aplicação toda vez que ela sofria alguma alteração.
- Remodelagem da estrutura de pastas e diretórios da aplicação para oferecer maior controle e customização na maneira como os seus recursos são acessados pelos usuários.

Dentre algumas das novidades na categoria 2 estão (NUNES & MAGALHÃES, 2010):

- Expansão do ciclo de vida de processamento de requisições para suportar chamadas assíncronas ao servidor (AJAX).
- Possibilidade de criar *bookmarks* de páginas JSF através do suporte à requisições HTTP do tipo GET.
- Validação de dados no cliente a nível de formulário bem como de componente.
- Novas definições de escopo para os *backing beans*.
- Melhorias na parte de navegação entre as páginas.

Relacionadas à categoria 3 algumas melhorias são (LOPES, 2010):

- Mudanças na hora de salvar o estado dos componentes para usar menos espaço, o que acarreta em melhor desempenho. Antes, toda a árvore era salva e serializada. Agora todo componente tem um estado padrão associado e apenas as diferenças do estado em relação a esse padrão são salvas. Isso diminui o tamanho do *view state* e melhora a escalabilidade das aplicações JSF.
- Uma parte maior do processamento de eventos e dos componentes visuais passou a ser feita no cliente.
- Suporte a *SystemEvents* além dos já existentes *PhaseEvents* e *FacesEvents*. Os *SystemEvents* permitem observar a ocorrência de outros tipos de eventos antes não detectáveis pela API do JSF, o que possibilita um maior controle sobre a aplicação.

As novidades da categoria 4 têm como objetivo atrair novos desenvolvedores para utilizarem os produtos que implementam a especificação JSF. Abaixo segue uma lista contendo algumas dessas novidades (JCP, 2009):

- Habilitar os componentes a terem um ciclo de vida próprio que é executado exclusivamente no cliente.

- Facilitar a interoperabilidade entre componentes de diferentes fabricantes.
- Permitir que recursos de uma aplicação JSF sejam acessados via REST.
- Permitir que componentes publiquem eventos em forma de *feeds* RSS/Atom.
- Suporte a REST (JSR 311).
- Permitir que páginas passem parâmetros entre elas.
- Customização visual dos componentes (*skins*, *themes*).

3.2 RECURSOS E FUNCIONAMENTO INTERNO DO JSF

O JavaServer Faces revolucionou a maneira como as aplicações *web* Java são criadas. Desenvolvido com o objetivo de alavancar o processo de criação de interfaces com o usuário para aplicações *web* Java de alta performance, o JSF também simplifica o desenvolvimento das mesmas. O JavaServer Faces oferece uma solução elegante para os principais problemas envolvidos no desenvolvimento de aplicações *web* através de três de seus principais recursos:

1. Arquitetura voltada à componentes.
2. Conjunto padrão de componentes visuais.
3. Infraestrutura de aplicação.

Esses são alguns dos recursos do *framework* que merecem destaque, mas a verdade é que o JSF é muito mais do que isso. Os tópicos seguintes se encarregarão de mostrar um pouco mais do potencial do JSF, do que ele é capaz e como funciona.

3.2.1 Funcionalidades Principais

Um dos recursos mais populares do JSF é a sua arquitetura voltada a componentes que por sua vez são capazes de agilizar muito o processo de

desenvolvimento de interfaces com o usuário de uma aplicação. Isso é feito através de uma API que oferece um conjunto de componentes que podem ser facilmente combinados para criar interfaces ricas com o usuário em questão de pouco tempo. Esses componentes ainda podem ser usados para criar outros componentes com funcionalidades mais especializadas, os chamados componentes compostos.

A fim de facilitar e agilizar mais ainda o desenvolvimento de aplicações *web* o JSF oferece um modelo de desenvolvimento orientado a eventos. Esse mecanismo possibilita que os componentes que são mostrados nas páginas que estão no cliente sejam capazes de interagir com os JavaBeans, ou *backing beans*, que estão no servidor. Cada uma dessas ações praticadas pelo usuário chega até o servidor na forma de uma requisição HTTP, que é então processada pelo ciclo de vida de processamento de requisições do JSF.

O ciclo de vida de processamento de requisições do JSF desempenha um papel fundamental pois é responsável por várias coisas, tais como cuidar da validação dos valores que o cliente submete para o servidor. Isso é feito em uma das seis fases do ciclo de vida conhecida como Processo de Validação, ou no original *Process Validation*. Esse ciclo de vida ainda é responsável por processar parâmetros submetidos pelo cliente, atualizar valores dos *beans* no servidor e gerar a resposta da requisição HTTP.

O JSF ainda oferece recursos para facilitar a internacionalização através de um mecanismo simples de manipulação de pacotes de mensagens de acordo com a localização desejada. Isso pode ser feito de maneira automática ou ainda de forma programática, possibilitando que o próprio usuário escolha o idioma em que todo o texto da aplicação será mostrado.

3.2.2 Ciclo de Vida do JSF

Para compreender melhor as funcionalidades oferecidas pelo JSF bem como o próprio *framemork* em si, um conhecimento mais aprofundado do seu ciclo de vida de processamento de requisições é fundamental. Conhecer apenas a documentação dos componentes não é o suficiente para aproveitar o máximo do JSF, pois é

entendendo o ciclo de vida que se compreende a ordem dos eventos e do processamento dos valores dos componentes (BESSA & PONTE, 2009). Esse ciclo de vida atua como um motor funcionando silenciosamente por baixo do capô possibilitando ao JSF que ele funcione, e que funcione bem.

Um mecanismo como esse se faz necessário pois, historicamente, a maioria do trabalho envolvido no desenvolvimento de uma aplicação *web* tem sido devotado ao processamento das requisições HTTP vindas dos clientes. Conforme a *web* evoluiu de um modelo estático de exibição de documentos acadêmicos para um ambiente dinâmico, a necessidade de processar requisições cada vez mais complexas surgiu e vem aumentando substancialmente com o passar dos anos. Isso fez com que o desenvolvimento de aplicações *web* se tornasse algo extremamente maçante. Como qualquer desenvolvedor experiente sabe, escrever código para processar os parâmetros das requisições enviadas pelo cliente geralmente envolve as seguintes tarefas (MANN, 2005):

- Fornecer um contexto para essa requisição, incluindo qualquer estado que possa ter sido deixado para trás por requisições anteriores.
- Realizar a validação e conversão de dados no servidor bem como disparar as devidas mensagens de notificação caso ocorra algum erro durante esse processo.
- Atualizar os objetos de dados do servidor com novos dados.
- Criar uma resposta para enviá-la de volta ao cliente, levando em consideração as requisições subsequentes que serão feitas pelo *browser* solicitando imagens, *scripts* e folhas de estilo.

Qualquer aplicação com uma interface com o usuário baseada na *web* deve levar em consideração esses aspectos, independente das tecnologias, *frameworks* ou linguagens de programação utilizadas. A necessidade de ter que lidar com essas tarefas é um efeito colateral proveniente do uso de um *browser* para exibir a interface com usuário e ao mesmo tempo estar conectado a um servidor através do protocolo HTTP como único meio de transporte entre os dois (BURNS & SCHALK, 2010). Felizmente, são essas tarefas que o ciclo de vida de processamento de requisições executa de forma automática e consistente sem a necessidade de haver uma intervenção por parte do desenvolvedor. Durante sua execução o ciclo de vida do JSF irá:

1. Determinar se a requisição está solicitando uma página ou um recurso. Se for uma requisição de recurso o servidor irá enviar os *bytes* deste recurso para o cliente. Caso contrário ele irá carregar a página JSP ou Facelets.
2. Criar no servidor uma representação do estado da interface com o usuário.
3. Gerar conteúdo apropriado de forma que o *browser* possa apresentá-lo ao cliente.

O ciclo de vida do JSF identifica automaticamente as porções do estado que sofreram alteração, por isso o estado visual que o cliente enxerga sempre está sincronizado com o estado visual no servidor.

3.2.2.1 Fases do Ciclo de Vida

Uma aplicação *web* típica possui funcionalidades que exigem um alto nível de interação entre o usuário e a aplicação, o que gera um grande número de requisições do cliente para o servidor. Por isso, é preciso gerenciar o estado das telas e a propagação dos dados através da própria aplicação bem como através do *browser*. Além disso, na maioria dos casos, somente texto trafega através do protocolo HTTP, o que aumenta a necessidade de validação dos dados recebidos e conversão dos objetos de negócio.

Para suportar esses e outros requisitos, das quais alguns já foram abordados no tópico anterior, o ciclo de vida de processamento de requisições foi dividido em seis fases: Restaurar Visão (*Restore View*), Aplicar Valores da Requisição (*Apply Request Values*), Processar Validações (*Process Validations*), Atualizar Valores do Modelo (*Update Model Values*), Invocar Aplicação (*Invoke Application*) e Renderizar Resposta (*Render Response*). A Figura 7 apresenta uma visão desse ciclo.

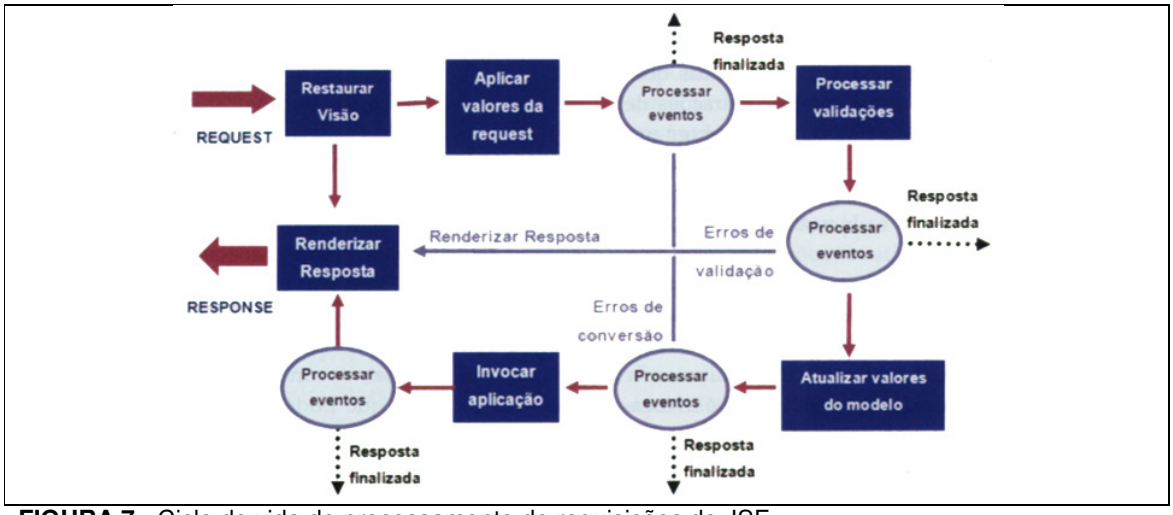

FIGURA 7 - Ciclo de vida de processamento de requisições do JSF.
FONTE: adaptado de NASCIMENTO (2007).

Primeiramente é importante dividir as requisições recebidas em duas categorias: requisições iniciais (a primeira vez que o cliente acessa uma página), também conhecidas como requisições *non-postback* e requisições de retorno (o cliente já acessou a página pelo menos uma vez), também chamadas de requisições *postback* (BESSA & PONTE, 2009). As requisições iniciais são atendidas através da execução da fase **Restaurar Visão** e **Renderizar Resposta**, ao passo em que as requisições de retorno seguem pelo caminho completo.

A fase **Restaurar Visão** é responsável pelo gerenciamento da árvore de componentes (*Faces View*) que representa o estado da página tal como é apresentada para o cliente. A Figura 8 ilustra de maneira gráfica essa árvore de componentes que fica armazenada no servidor.

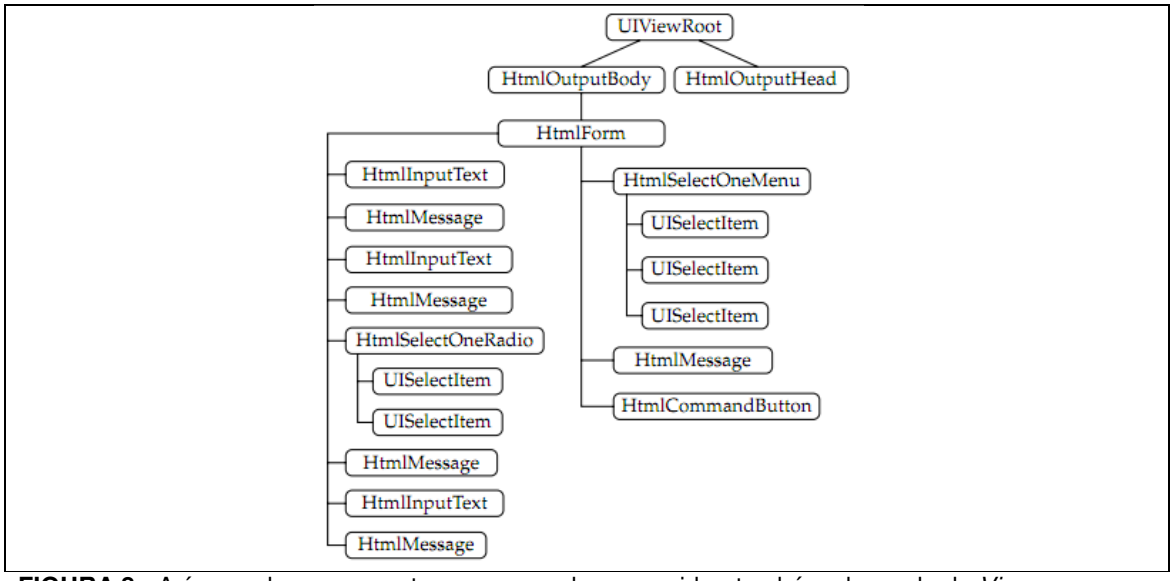

FIGURA 8 - A árvore de componentes armazenada no servidor, também chamada de *View*.
FONTE: BURNS & SCHALK (2010).

A função desta fase consiste em criar o conjunto de componentes JSF utilizados na página caso seja uma requisição inicial, ou recuperá-lo caso seja uma requisição de retorno. Se tratando de uma requisição inicial, a página e a sua representação de componentes é criada e armazenada em um objeto conhecido como *FacesContext*. O *FacesContext* serve como um meio de armazenar todos os dados pertinentes à requisição durante seu processamento ao longo do ciclo de vida. É importante ressaltar que não é preciso se preocupar com a sobreposição acidental de dados de múltiplas requisições de clientes diferentes dentro do *FacesContext*. Isso porque a *Servlet API* garante que as operações realizadas em uma requisição são *thread-safe*, ou seja, todas as operações no *FacesContext* ocorrem em uma única *thread* por requisição de cliente.

Na fase **Aplicar Valores da Requisição**, cada componente extrai seu novo valor dos parâmetros da requisição, armazenando-o em variáveis de estado no servidor. Cada nó da *View* que representa os vários componentes da página agora pode receber seu valor atualizado pelo cliente. Isso é, se o componente for capaz de armazenar valores, pois no JSF, em geral, existem dois tipos de componentes: aqueles que podem conter valores como campos de texto e *check boxes* por exemplo e os que produzem eventos ou ações, como botões e *links*.

Na fase **Processar Validações**, todos os validadores associados aos componentes são executados. Caso seja encontrado qualquer valor inválido, uma

mensagem de erro (*FacesMessage*) é colocada no *FacesContext*, a resposta é renderizada e o fluxo normal da aplicação é interrompido. Caso contrário o fluxo segue para a próxima fase.

Se os dados foram validados e convertidos com sucesso então eles estão prontos para serem transferidos para a camada de modelo. Isso é feito na fase **Atualizar Valores do Modelo** através das expressões de valor (*JSF Expression Language*) dos componentes de entrada que fazem referência aos *backing beans*. É durante essa fase que os valores das propriedades dos objetos da aplicação são atualizados com os valores dos componentes das páginas aos quais eles estão ligados. O Quadro 2 mostra como é feita essa ligação na prática.

```
<h:inputText id="inputNomeInt" value="#{backingInterprete.novoInterprete.nome}" />
```

QUADRO 2 - Ligação de um atributo da página à uma propriedade do *backing bean* através de Expression Language.

A próxima fase é denominada **Invocar Aplicação** e é onde os métodos do *backing bean* referenciados nas expressões de ação dos componentes de comando são executados e as regras de navegação são resolvidas. O Quadro 3 mostra como uma ação de um *backing bean* é associada a um componente.

```
<h:commandButton action="#{backingInterprete.salvarNovo}" value="Salvar" />
```

QUADRO 3 - Método de um *backing bean* associado a um componente de ação.

Em segundo plano é nessa fase que o método `processApplication()` do componente `UIViewRoot` é invocado. Este método, por sua vez, propaga todos os eventos pendentes dessa fase para os seus respectivos objetos `UIComponent`, caso eles implementem a interface `ActionSource`. Isso é realizado através do método `broadcast()` de cada `UIComponent` que essencialmente dispara os eventos fazendo com que os devidos *listeners* sejam alertados e dessa forma realizem o seu processamento. Neste momento é seguro executar a lógica da aplicação porque os valores já passaram por conversão e validação. A partir do método que processa esse evento (o `salvarNovo()` no exemplo da Figura 11), um *backing bean* pode sugerir qual será a próxima página a ser renderizada retornando um resultado, que é uma *String* que será usada para localizar uma regra de navegação ou uma página com aquele nome.

O ciclo de vida de processamento de requisições chega ao seu fim na fase **Renderizar Resposta**. Para gerar o conteúdo que é enviado de volta para o cliente, mais uma vez uma sequência de métodos é invocada a partir de cada componente da árvore. É através desses métodos que cada componente renderiza sua representação para o cliente. Essa representação pode ser gerada em uma variedade de linguagens e formatos, tais como HTML, WML, XML, JSON e muitos outros. Além disso, a fase Renderizar Resposta também salva o estado atual da *View* a fim de torná-la acessível e reutilizável para as requisições posteriores. A persistência do estado da árvore pode ser no cliente ou no servidor dependendo da escolha da aplicação. Além disso, caso haja mensagens de erro armazenadas no *FacesContext*, elas são apresentadas para o cliente (NASCIMENTO, 2007).

3.3 JAVASERVER FACES VERSÃO 2

Existe um debate acalorado sobre qual seria o cenário mais apropriado para a criação de um *framework* para desenvolvimento de aplicações *web*: uma sala de reunião, onde os especialistas discutem suas ideias, ou o mundo real, onde os *frameworks* são forjados a partir de necessidades recorrentes do dia-a-dia. A resposta pode parecer um tanto óbvia e intuitiva, mas o fato é que a versão 1 do JSF foi projetada longe da realidade diária da maioria dos desenvolvedores. Sendo assim, logo no início o JavaServer Faces foi recebido com algumas críticas por parte da comunidade dos desenvolvedores. Porém, uma coisa que os especialistas fizeram certo ao criar a especificação técnica do JSF foi dar a ele a capacidade de ser estendido e melhorado pelos protagonistas do mundo real, criando assim um *framework* altamente personalizável. Ao longo de seus muitos anos de existência pôde-se ver uma imensidão de bibliotecas e APIs sendo criadas para integrá-lo e melhorá-lo, frutos de vários projetos de código aberto, tais como *Richfaces*, *Seam*, Facelets, *Woodstock* e *JSFTemplating* para citar alguns.

Mas mesmo sendo um *framework* flexível, capaz de auxiliar grandemente no desenvolvimento de aplicações *web* e de ter introduzido novos conceitos como árvore de componentes, fases do ciclo requisição e resposta, conversores e validadores, o JSF ainda carecia maturidade. O excesso de configurações, a grande

quantidade de arquivos XML necessária, a falta de suporte nativo à AJAX e a complexidade de uso foram algumas das maiores queixas dos desenvolvedores em relação à versão 1.x do JSF. Determinado a não cometer o mesmo erro duas vezes, o grupo de especialistas responsável pela especificação da versão 2 do JSF decidiu aproveitar as melhorias introduzidas por aqueles projetos e torná-las parte da nova versão. Apesar de ter sido projetada por um grupo de especialistas, a versão 2 do JSF não deixou de ser influenciada pelas inovações do mundo real e foi capaz de combinar o melhor que cada cenário tem a oferecer (GEARY, 2009).

Com o lançamento da versão 2.0 do JSF em meados de 2009, muitas novidades foram incorporadas com o objetivo de simplificar o desenvolvimento e aumentar a produtividade do desenvolvedor, benefícios esses tão almejados pela comunidade. Dentre as várias novidades destacam-se as seguintes:

- Suporte nativo à AJAX e integração das requisições assíncronas ao ciclo de vida de processamento do JSF.
- Adoção do Facelets como tecnologia padrão para a camada de visão.
- Navegação implícita e condicional de forma simplificada.
- Validação melhorada e mais simples com a JSR 303 - *Bean Validation*.
- Linguagem de expressão mais poderosa.
- Suporte a GET possibilitando que páginas JSF sejam salvas no favoritos do navegador.
- Mensagens mais detalhadas durante o desenvolvimento através dos estágios de projeto.
- *Tags* específicas para gerenciamento de recursos da aplicação (imagens, scripts, folhas de estilos, etc).
- Configurações sem XML através do uso de anotações.

Essas são somente algumas das melhorias introduzidas na versão 2 do JSF. Todas buscam tornar o desenvolvimento de soluções interativas para *web* mais simples e a experiência dos usuários destas aplicações melhor.

3.4 O QUE MELHOROU NA VERSÃO 2

Para demonstrar de maneira mais eficaz as várias novidades do JSF elas foram classificadas em quatro categorias: Páginas, Componentes, Controladores de Página e Navegação. Essas categorias representam artefatos do desenvolvimento *web* e a no Quadro 4 é possível identificar à qual delas cada novidade pertence.

Novidade/Artefato	Páginas	Componentes	Controladores	Navegação
XHTML	X	X		
Templates	X			
Recursos	X	X	X	
Ajax	X	X		
Tratamento de Exceções			X	
Estágio de projeto	X			
Novos escopos	X		X	
Eventos	X		X	
Parâmetros de visão e favoritos	X			
Melhorias na EL	X			
Validação	X			
Anotações		X	X	
Genéricos			X	
Navegação implícita				X
Navegação por caminho físico				X
Navegação condicional				X

QUADRO 4 - Novidades do JSF classificadas por artefato.
FONTE: adaptado de NUNES & MAGALHÃES, 2010.

É possível notar que boa parte das melhorias diz respeito ao desenvolvimento das páginas e isto faz bastante sentido, pois a camada de visão, dentre as demais do padrão de projeto arquitetural MVC, é a que mais necessita de facilidades. Portanto essa será a primeira categoria cujas novidades serão abordadas e além de uma breve explicação sobre cada uma delas será também apresentado um comparativo entre a versão 1 do JSF.

3.4.1 Páginas JSF

Uma página *web* é um documento formatado para a *World Wide Web* que pode ser acessado por um navegador e apresentado em algum dispositivo de saída como a tela de um computador, celular, TV e afins. Em geral, é declarada usando HTML ou XHTML. Pode ser estática ou gerada dinamicamente no servidor *web*. É composta por componentes visuais tais como caixas de texto e botões de comando. Em JSF declara-se uma página *web* dinâmica usando Facelets ou JSP. Esta declaração é representada em memória como um conjunto de componentes visuais e não visuais chamado árvore de componentes, cuja maior responsabilidade é renderizar a página *web* (ORACLE, 2011).

O processamento da árvore de componentes é feito pelo ciclo de vida de processamento de requisições e é iniciado quando um evento de um componente de ação é enviado ao servidor (como por exemplo o clique de um botão). Um recurso notável das páginas JSF é que o desenvolvedor pode criá-las ainda sem controladores associados e já ter uma ideia aproximada do resultado final. Ou seja, não é preciso criar as páginas em HTML primeiro e depois converter tudo para o XHTML do JSF. As páginas podem ser desenhadas com os componentes que serão usados na aplicação, mas ainda sem as classes Java associadas (GEARY, 2009).

3.4.2 XHTML no Lugar de JSP

A primeira versão do JSF definiu que essas páginas deveriam ser declaradas usando JSP com bibliotecas de *tags* especiais do JSF. Mas sempre houve muitas críticas em relação ao uso do JSP como *view* do JSF. Era difícil integrar as bibliotecas de *tags* que não fossem de JSF, o ciclo de vida se confundia com o processamento do JSP, o desempenho não era adequado, as páginas eram pouco reaproveitáveis e páginas JSP nunca foram uma maneira muito adequada para se declarar uma árvore de componentes (NUNES & MAGALHÃES, 2010). Além disso, vários conceitos de JSF não são suportados por JSP, e conceitos do JSP se

tornaram desnecessários para JSF. Um exemplo: a necessidade de tradução e compilação dos arquivos JSP. Faz sentido, pois os arquivos JSP devem se tornar servlets no servidor, mas por outro lado isso não interessa aos propósitos do JSF que apensa pretende representar uma árvore de componentes em memória. Logo começaram a surgir tecnologias alternativas para descrever páginas dinâmicas em JSF, sendo Facelets a mais popular dentre elas. Essa tecnologia foi desenvolvida pela comunidade e utiliza arquivos XHTML para descrever as páginas JSF. Como foi criada para JSF e procurou oferecer recursos que o JSP não podia, ela se tornou a tecnologia mais utilizada na versão 1.x, apesar de não ser oficial. Alguns dos benefícios do Facelets são (LOPES, 2010):

- Menor tempo de compilação necessário.
- Melhor desempenho de renderização.
- Mecanismo simples e robusto para criar composições, *templates* e novos componentes.

Na versão 2 do JSF a maturidade do Facelets foi comprovada pelo grupo de especialistas da especificação, adotando a tecnologia e substituindo o JSP como recomendação oficial. O uso do JSP, embora ainda suportado, passa a ser desencorajado e Facelets agora é a *view* padrão para JSF. O Quadro 5 mostra uma página dinâmica descrita usando JSP puro.

```
<%@ page import="..." %>
<html>
  <head>
  </head>
  <body>
    <span>Hello ${usuario.nome}</span>
  </body>
</html>
```

QUADRO 5 - Página dinâmica descrita em JSP puro.

Já o Quadro 6 mostra uma página dinâmica descrita usando JSP com tags JSF 1.

```
<%@ taglib uri=... prefix=h %>
<%@ taglib uri=... prefix=f %>
<f:view>
  <html>
  <head>
  </head>
  <body>
    Hello <h:outputText value="${controlador.usuario.nome}" />
  </body>
  </html>
</f:view>
```

QUADRO 6 - Página dinâmica descrita em JSP usando *tags* JSF 1.

E o Quadro 7 mostra uma página dinâmica descrita usando XHTML com *tags* JSF 2. Comparando os três quadros anteriores é possível perceber claramente como o uso do Facelets simplifica o trabalho do desenvolvedor das páginas na versão 2.

```
<? xml version="1.0" ?>
<html xmlns:h=... xmlns:f=...>
  <h:head>
  </h:head>
  <h:body>
    Hello #{controlador.usuario.nome}
  </h:body>
</html>
```

QUADRO 7 - Página dinâmica descrita em XHTML usando *tags* JSF 2

O XHTML tem muitas vantagens interessantes: por ser um documento XML, é muito mais fácil de interpretá-lo e a validação da sintaxe também se torna mais simples já que não existem blocos de código Java misturados com a declaração da página. Até mesmo a declaração das bibliotecas de *tag* fica mais fácil com o uso do XML *namespace* ao invés de diretivas JSP. Com XHTML também é possível usar a *Expression Language* (EL) em qualquer parte do documento, não apenas dentro das *tags* JSF. O fato do Facelets ter sido adotado como tecnologia oficial na versão 2 do

JSF faz com que não seja mais necessário configurar no *web*.xml e no faces-config.xml sua integração ao JSF como era na verão 1.x.

3.4.3 Recursos Gerenciados

Em uma aplicação JSF é muito importante saber como referenciar um recurso em uma página e como publicá-lo a fim de torná-lo um recurso que possa ser utilizado pela aplicação. Um recurso no contexto do JSF é qualquer arquivo estático responsável pela aparência ou pelo comportamento de um componente de interface de usuário em uma página *web*. Alguns exemplos típicos de recursos são: imagens, arquivos de estilo (CSS) e arquivos *Javascript*.

Para referenciar esses recursos, várias *tags* JSF foram criadas. As *tags* <h:head> e <h:body> correspondem às *tags* <head> e <body> do HTML e existem para que seja possível dizer à API em que parte do HTML o recurso deve ser declarado. As *tags* <h:outputStyleSheet> e <h:outputScript> permitem referenciar arquivos CSS e arquivos *Javascript*, respectivamente. A *tag* <h:graphicImage> ganhou novos atributos, library e name, para referenciar imagens, e com o atributo value passa a ser possível referenciar imagens usando o objeto implícito resource na EL. Sendo assim, para publicar um recurso JSF basta colocar o arquivo dentro da pasta resources na raíz da aplicação *web* ou na pasta META-INF/resources de um arquivo JAR. O JSF é capaz de determinar o caminho completo do recurso automaticamente.

A classe *FacesServlet*, não só trata requisições de páginas, mas passa a tratar também requisições de recursos. Isso até a versão 1.x do JSF só era possível de ser feito através de filtros ou *servlets* adicionais, o que por sua vez acaba exigindo mais configurações. Um benefício do suporte a recursos para páginas está na facilidade de referenciá-los através das novas *tags* e da criação de novos atributos em *tags* já disponíveis na versão 1.x. Outro benefício ainda é a possibilidade de publicar os recursos não só na seção pública do arquivo WAR, mas também em arquivos JAR, facilitando ainda mais a criação de bibliotecas de componentes.

3.4.4 Templates e *Composite Components*

Outra novidade para as páginas JSF é o suporte nativo à criação de *templates*, recurso esse oferecido pelo Facelets. Isso já era possível desde a versão 1.2 do JSF, mas só depois de configurar o Facelets no `web.xml` e no `faces-config.xml`. Com o uso de *templates* é possível definir uma página com uma estrutura padrão e reaproveitável e depois definir páginas específicas que preencham esse *template* com informações próprias. O suporte a *templates* é simples assim: um arquivo XHTML é tratado como modelo e para isso deve delimitar áreas substituíveis através da *tag* `<ui:insert>`. Cada arquivo que utiliza o modelo deve referenciá-lo utilizando a `<ui:composition>` e substituir o que for preciso através do uso da *tag* `<ui:define>`. O Quadro 8 mostra como é feita essa declaração do modelo.

template.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      mxlns:h="http://java.sun.com/jsf/html">
    <h:head>
      <title>
        <ui:insert name="titulo">Titulo da app</ui:insert>
      </title>
    </h:head>
    <h:body bgcolor="green">
      <ui:insert name="corpo">Corpo da app</ui:insert>
    </h:body>
</html>
```

QUADRO 8 - Modelo com as definições de *layout* através das *tags* do *namespace* "ui".

O Quadro 9 mostra como o modelo é utilizado em outra página.

```
pagina.xhtml
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    mxlns:h="http://java.sun.com/jsf/html">
  <ui:composition template="/template.xhtml">
    <ui:define name="corpo">
      <h:form id="formTeste">
        <h:panelGrid columns="2">
          Nome: <h:inputText />
          Idade: <h:inputText />
          <h:commandButton value="Cadastrar" />
        </h:panelGrid>
      </h:form>
    </ui:define>
  <ui:composition>
</html>
```

QUADRO 9 - Página implementando e definindo as diretivas do modelo.

Outro recurso importante do Facelets é a facilidade na criação de componentes reutilizáveis, denominados *Composite Components*. No JSF 1.x, criar um novo componente exigia um trabalho enorme que envolvia a implementação de várias classes e configurações, uma tarefa de grande complexidade. Para ilustrar isso de maneira simples, tome-se de exemplo uma listagem de itens que precisa ser reutilizada em vários pontos diferentes da aplicação, mas que precisa receber os objetos a serem listados em cada situação. Uma alternativa seria criar um componente que recebe uma lista e os renderiza para cada cenário. Com Facelets esse novo componente pode ser definido através das *tags* `<composite:interface>` e `<composite:implementation>`. Esse componente é definido diretamente em um arquivo XHTML dentro de uma pasta pasta do diretório `/resources` com o nome do namespace desejado. Por exemplo, para criar uma *tag* `<jon:tabela>`, é necessário criar o arquivo `tabela.xhtml` na pasta `/resources/jon/`. No arquivo tabela.xhtml é preciso usar a *tag* `<composite:interface>` para definir o que os usuários do componente precisam saber (atributos, *facets*, eventos) e a `<composite:implementation>` para implementar o código do componente.

3.4.5 Suporte à AJAX

O suporte à AJAX do JSF é com certeza uma das novidades mais importantes da versão 2. Apesar de já ser possível na versão 1.x, usar AJAX em uma aplicação JSF significava usar alguma biblioteca não padronizada com extensões proprietárias tal como Richfaces, ADF Faces, IceFaces e Dynamic Faces para citar algumas. O problema é que cada uma dessas bibliotecas implementou esta funcionalidade da sua maneira. Isso acabou fazendo com que desenvolvedores JSF 1.x precisassem escolher bibliotecas de componentes com suporte a AJAX que fossem compatíveis. Ou seja, os desenvolvedores tiveram que testar e descobrir que não era viável usar Richfaces com IceFaces e quando era nem todas as funcionalidades operavam corretamente.

Para dar um fim a este problema a especificação da versão 2 padronizou uma biblioteca chamada *ajax.js*, escrita em Javascript, que define funções AJAX que deverão ser utilizadas tanto por desenvolvedores de aplicação como por desenvolvedores de componentes. Além de ser possível usar diretamente as funções Javascript desta biblioteca, a *tag* `<f:ajax>` permite dar capacidade AJAX (ajaxificar) para componentes que não o utilizam nativamente. O Quadro 10 mostra como essa chamada pode ser feita na prática utilizando tanto a API Javascript como a *tag* `<f:ajax>`.

```
<!-- Tabela que será renderizada ao clicar no botão -->
<h:dataTable id="tabela">
...
</h:dataTable>

<!-- Usando a API Javascript -->
<h:comandButton id="botaoSalvar" value="Salvar"
  actionListener="#{backingBean.salvar}"
  onClick="jsf.ajax.request(this, event, {execute:'botaoSalvar', render:'tabela'});" />

<!-- Usando a tag de AJAX -->
<h:comandButton id="botaoSalvar" value="Salvar"
  actionListener="#{backingBean.salvar}">
  <f:ajax render="tabela" />
 </h:commandButton>
```

QUADRO 10 - Maneiras de usar AJAX em uma aplicação JSF 2.

Esta proposta é uma solução semelhante ao uso da *tag* `<a4j:support>` oferecida pelo Richfaces. A *tag* `<f:ajax>`, no entanto, permite não só o conceito de *nesting*, que é ser declarada dentro de um componente como um botão de ação ou um link de ação, mas também o conceito de *wrapping*, quando pode agrupar um conjunto de componentes que devem ser enviados na requisição AJAX. Outra ideia que foi emprestada do Richfaces, mais especificamente da *tag* `<a4j:region>`.

É importante destacar que o suporte a AJAX é fruto da criação do conceito de *Behavior* (Comportamento) na especificação. A classe que representa o componente da *tag* `<f:ajax>` é *AjaxBehavior* e é derivada da classe *ClientBehavior*. Na prática a especificação não só permitiu vincular funções AJAX a componentes visuais JSF, mas também criou uma API que oferece ao desenvolvedor de componentes a capacidade de vincular a esses componentes qualquer função Javascript. Isso possibilita componentizar comportamentos cliente tradicionais de interação, como abrir um diálogo de confirmação ao clicar em um botão de "Excluir", mostrar um *tooltip* personalizado ao passar o mouse por cima de um componente, entre outros.

Outro fato notável é que a biblioteca *jsf.js* é a primeira biblioteca Javascript da história da plataforma *Java Enterprise Edition*, da qual o JSF faz parte. Além disso, na especificação também foram criadas as classes *PartialViewContext* e *PartialResponseWriter*, que estão relacionadas ao suporte a requisições AJAX e que estão acessíveis nas classes Java para possíveis personalizações. Elas oferecem o benefício da renderização parcial da página, o que evita carregamentos desnecessários de componentes que não precisam ser atualizados.

3.4.6 Estágios de Projeto

Uma dificuldade para os desenvolvedores JSF 1.x era identificar o que estava acontecendo de errado na aplicação. Muitas vezes a resposta de uma requisição JSF não era a esperada e a causa não era esclarecida pois o *framework* não oferecia mensagens amigáveis (BESSA & PONTE, 2009). Isso melhorou muito com a introdução do Facelets no JSF a partir da versão 1.2 e o console passou a

oferecer mais informações para o desenvolvedor sobre o que havia acontecido de errado.

Na versão 2 do JSF foi introduzido o conceito de estágio de projeto. Assim, o *framework* passa a oferecer mensagens mais detalhadas sobre o que pode estar acontecendo nas requisições JSF. Para isso, basta configurar o parâmetro de contexto `javax.faces.PROJECT_STAGE` como `Development` no arquivo *`web`*`.xml`. O Quadro 11 mostra como essa configuração é feita na prática.

```
web.xml
<context-param>
  <param-name>javax.faces.PROJECT_STAGE</param-name>
  <param-value>Development</param-value>
</context-param>
```

QUADRO 11 - Configurando o estágio de projeto da aplicação.

Na prática isso significa que, por exemplo, se o desenvolvedor criar um formulário mas esquecer de incluir um botão de comando ou um link de comando dentro dele, uma mensagem será exibida na página quando ela for renderizada. Se o erro ocorrer no lado servidor, a página padrão de erros do Facelets será mostrada exibindo a pilha de execução e a linha do erro bem como os atributos armazenados nos diferentes escopos. Se um *outcome* não puder ser resolvido nas *tags* <h:button> e `<h:link>` uma mensagem também será mostrada na página renderizada. Outras opções de configuração para o estágio de projeto são Production, SystemTest e UnitTest.

Outra melhoria ainda em relação à versão 1.x do JSF é que na versão 2 foi criado o conceito de ExceptionHandler que, por padrão, continua capturando e mostrando as exceções. Contudo, passa a relançar a exceção para que ela possa ser tratada pelo desenvolvedor: quer seja por redirecionamento para uma página *web* configurada na aplicação quer seja por uma classe Java que decide o que deve ser feito. Isso é um grande avanço, já que nas versões anteriores do JSF as exceções ocorridas durante o ciclo de vida da requisição eram capturadas pelo *framework*, apresentadas no console, mas não eram relançadas. Isto impedia que o desenvolvedor pudesse interceptá-las para tratamento.

3.4.7 Novos Eventos de Sistema e Aplicação

A versão 2 do JSF passa a suportar uma maior variedade de eventos do que aqueles presentes na versão 1.x que eram PhaseEvent, ValueChangeEvent e ActionEvent. Agora com o conceito de eventos de sistema e evento de aplicação incorporados, que já estavam previstos para a versão 1.x, é possível tomar decisões mais simples de acordo com os eventos que estiverem acontecendo no *framework*.

Sendo assim, nas páginas JSF agora está disponível a *tag* `<f:event>`, que permite capturar eventos de interesse e associá-los a um método no *backing bean* que então pode reagir de acordo. Um uso interessante para isso é assinar o evento preRenderView e assim permitir que um método seja executado logo antes que a página seja renderizada. Para fazer isso na versão 1.x era necessário assinar o PhaseEvent e executar um trecho de código Java antes da fase seis do ciclo de vida.

Existem vários outros eventos que podem ser assinados, tais como postValidate e preValidate. A lista passa de 20 eventos divididos em eventos do *framework* e eventos dos componentes, como por exemplo addToView, postRestoreState e preRenderComponent.

3.4.8 Suporte à Instrução GET

Uma crítica constante ao JSF até a versão 1.x era o fraco suporte a GET e sua característica de *postback* que trata as requisições quase sempre como POST. Não suportar adequadamente o uso de GET gera dificuldades no uso de *bookmarks* para páginas da aplicação e prejudica a *Search Engine Optimization* (SEO).

A partir da versão 2 do JSF as páginas da aplicação podem ser referenciadas através da instrução HTTP GET. Esse recurso possibilita que parâmetros possam ser adicionados ao endereço de páginas JSF e que sejam devidamente tratados, assim como acontece nos componentes de entrada dos formulários. Em outras palavras, é possível passar parâmetros através da URL que podem ser convertidos,

validados e vinculados a propriedades de *backing beans*. Isso já era possível na versão 1.x, mas somente através do uso de APIs externas como PrettyFaces.

A versão 2 também oferece a possibilidade de gerar URLs com parâmetros usando as *tags* `<h:link>` ou `<h:button>` combinadas à *tag* `<f:param>`. A página destino, que vai ser acionada pelas *tags* `<h:button>` e `<h:link>` é configurada com o atributo *outcome*. Por fim, o atributo includeViewParams permite que os parâmetros da página destino sejam determinados a partir dos endereços recebidos na própria página origem.

3.4.9 Expression Language Melhorada

O XHTML ganhou várias funcionalidades novas no JSF 2 graças ao aprimoramento da Expression Language (GEARY, 2009). Agora é possível invocar métodos arbitrários nas páginas JSF e um conjunto de novos objetos implícitos também foi criado. Na prática isso significa que o objeto implícito *resources* permite acessar o endereço de um recurso em qualquer ponto do documento XML e a invocação oferecida pelo JEE 6 EL permite que parâmetros sejam passados em métodos declarados na EL, por exemplo, em métodos vinculados ao atributo *action* de componentes de comando.

Este recurso faz com que a *tag* `<f:setPropertyActionListener>` seja necessária somente em servidores de aplicação JEE 5 ou em *containers web* como o Tomcat.

3.4.10 Validação Simplificada

O JSF dispõe de mecanismos de validação desde sua primeira versão. A validação está diretamente ligada ao ciclo de vida de processamento de requisições do JSF, sendo o foco principal da fase três. Por esse motivo, o JSF 1.x oferece um conjunto de validadores que permitem verificar se um valor numérico está dentro de

uma faixa específica e se um texto tem uma quantidade mínima ou máxima de caracteres. Também é possível criar um validador personalizado através de um método do *backing bean* ou como uma classe separada. Porém, é comum que essas validações sejam feitas diretamente na camada de negócio, fazendo com que essa validação JSF seja redundante.

Com a introdução da JSR 303, conhecida como Bean Validation, na plataforma EE 6 e a sua integração com o JSF se tornou possível usar anotações para validar atributos de um JavaBean indicando a validação que deve ser feita para cada atributo. Sendo assim, os validadores nos componentes de entrada JSF não são mais necessários como eram na versão 1.x.

Por ser parte da especificação Java EE 6, a Bean Validation é usada também pela Java Persistence API automaticamente, o que garante a integridade no banco de dados. Se o JSF identificar uma implementação da Bean Validation, como o Hibernate Validator por exemplo, basta anotar as classes de modelo e tudo será automaticamente validado na fase três do JSF, inclusive com a exibição das mensagens de erro nos devidos componentes.

Para indicar que um determinado JavaBean deve ser validado conforme as anotações da JSR 303 foi criada a *tag* `<f:validateBean>`. Em servidores de aplicação com suporte completo a JEE 6 a validação é feita automaticamente e o uso da *tag* é totalmente opcional. As *tags* `<f:validateRequired>` e `<f:validateRegexp>` também são novidades do JSF 2. A primeira diz se um item é obrigatório ou não e a segunda permite validar conteúdo textual com expressões regulares.

3.5 COMPONENTES

O conceito de componente no JSF é muito abrangente: classes Java definem comportamentos e atributos de componentes que podem ser renderizados não só em HTML, como também em outras linguagens de marcação. Essa infraestrutura de componentes do JSF ainda possibilitou que o conjunto básico de componentes visuais oferecido por ele fosse significativamente ampliado por bibliotecas de componentes de terceiros. Para o desenvolvedor isso significa que funcionalidades

avançadas e ricas estejam à sua disposição sem a necessidade de dominar HTML, Javascript e CSS.

Entretanto, por mais simples que seja uma aplicação *web* nos dias hoje é comum existir a necessidade de criar componentes compostos: combinar vários componentes diferentes em uma área para reuso. Uma seção de login, uma barra de ferramentas, um menu com busca integrada ou uma caixa de texto com as suas mensagens de erro são alguns exemplos. Em JSF 1.x isso pode ser feito de duas maneiras: criar uma subclasse de UIComponent ou criar *templates* de componente, usando os recursos do Facelets. A primeira alternativa é bastante trabalhosa e foi alvo de muita crítica no JSF pois envolve codificar HTML, CSS e Javascript em classes Java. Processo similar ao utilizado no desenvolvimento de páginas *web* antes da criação do JSP. A segunda alternativa é mais prática, mas ainda não oferece recursos para identificar precisamente os atributos de entrada e suportar a passagem de atributos e métodos vinculados a um *backing bean*.

No JSF 2 o conceito de componentes compostos foi criado a partir das ideias de *template* de Facelets, dos recursos existentes na solução Tag Files do JSP e dos próprios conceitos de componentes já disponíveis na especificação. Isso significa que na versão 2, para criar um componente composto JSF basta criar um arquivo XHTML e publicá-lo na pasta de recursos.

3.5.1 Componentes Compostos com XHTML

Um arquivo XHTML na pasta *resources/components* é o que um desenvolvedor precisa escrever no JSF 2.0 para ter um componente composto. Nada mais de configurações no `faces-config.xml` e criação de classes para componentes, renderizadores e tratadores de *tag* (`<TagHandler>`). Isto pode ser traduzido em: agora é mais fácil e mais rápido criar componentes compostos, que promovem o uso de soluções interativas nas páginas do que era na versão 1.x. Este arquivo XHTML faz uso das bibliotecas *composite*, *ui*, *h* e *f*. Apenas a primeira é nova, pois as demais já existiam no JSF 1.x. O arquivo descrito no XHTML herda a classe base UIComponent e pode ter suas capacidades interativas enriquecidas com

recursos como imagens, CSS e Javascript através da EL. As *tags* `<composite:intertace>` e `<composite:implementation>` delimitam respectivamente a interface pública do componente e sua implementação.

Qualquer arquivo XHTML que esteja dentro desta pasta é tratado como um componente, já a pasta é tratada como biblioteca. Caso exista interesse em criar várias bibliotecas basta criar pastas dentro da pasta *components*, por exemplo: uma pasta *login* poderia ser criada dentro de *components* com arquivos como `loginArea.xhtml`, `publicidadeArea.xhtml`. Para quem ainda pretende criar componentes a partir de classes também há novidades: a classe *StateHelper* evita escrever os métodos saveState() e restoreState(), a anotação @FacesComponent dispensa a declaração no `faces-config.xml`, e o arquivo `taglib.xml` simplifica a declaração de bibliotecas, antes responsabilidade do *Tag Library Descriptor* (TLD).

3.5.2 Recursos para Componentes

Além de facilitar o desenvolvimento de páginas JSF os recursos também facilitam o desenvolvimento de componentes. Em um arquivo XHTML além de *tags* também é possível usar EL com os objetos implícitos já conhecidos no JSF 1.x: requestScope, sessionScope, request, response, params entre outros. Além desses no JSF 2 foi introduzido o objeto implícito resources, já apresentado. Somente para componentes compostos, na EL é possível acessar o objeto implícito *cc*, sigla de *composite component*, através do qual pode-se referenciar atributos do componente, o identificador de cliente e a lista de mensagens associada ao componente. Com *#{cc.clientId}* é possível utilizar o identificador do cliente em funções Javascript dentre outras utilizações e com *#{cc.messageList}* é possível renderizar as mensagens associadas a este componente.

Nas classes Java que herdam UIComponent também é possível referenciar recursos. Isto pode ser feito através das anotações @ResourceDependency e @ResourceDependencies. É importante destacar que o suporte a recursos é padrão para imagens, CSS e Javascript. Em outras palavras, a implementação de referência sabe declarar os recursos citados em uma página HTML, porém não está apta para

declarar qualquer arquivo contido na pasta *resources*. O Quadro 12 mostra como fica a configuração de uma classe de componente usando anotação no código.

```
@ResourceDependency(name="jsf.js", library="javax.faces", target="head")
public class MeuComponente extends UIOutput {
...
}
```

QUADRO 12 - Classe de componente JSF declarando uma dependência através de anotação.

3.5.3 Componentes e AJAX

No JSF 2 componentes compostos podem referenciar recursos como CSS, Javascript e imagens. Um recurso padrão disponível na implementação de referência é o *jsf.js* da biblioteca `javax.faces`. Além de ser possível declarar o recurso de AJAX do JSF em uma página JSF, também é possível declarar em um componente composto usando a *tag* `<h:outputScript>` e em um componente escrito com classe Java através da anotação @ResourceDependency.

Além de tudo isso ainda é possível vincular uma instância da classe AjaxBehavior a um componente e torná-lo capaz de suportar AJAX sem qualquer necessidade de configuração por parte do usuário do componente. Dessa forma a funcionalidade AJAX estaria relacionada a um evento do componente. Também é possível escrever *behaviors* com a possibilidade de associar qualquer código Javascript a um componente, não somente código AJAX. Isto permite componentizar comportamentos em uma aplicação, como por exemplo executar um efeito de transição quando um evento é disparado no navegador. Até a versão 1.x isso só era possível utilizando bibliotecas de componentes de terceiros.

3.5.4 Anotações no Lugar de XML

Com as anotações @FacesComponent, @FacesConverter e @FacesValidator não é mais necessário declarar estes componentes no arquivo

`faces-config.xml`. Mais uma novidade que facilita a criação de bibliotecas de componente além de evitar erros de configuração decorrentes de refatoração do nome do pacote ou da classe. O Quadro 13 mostra como é feita a declaração de um validador no JSF 1.x. No JSF 2 essa mesma configuração pode ser feita diretamente na classe do validador aplicando a ela a anotação @FacesValidator, eliminando assim a necessidade de existir um arquivo específico para isso.

```
<validator>
 <validator-id>validaEdNomeGenero</validator-id>
 <validator-class>br.edu.utfpr.blasterMusic.validators.ValidaEdNomeGenero</validator-class>
</validator>
```

QUADRO 13 - Configuração de um validador feita no `faces-config.xml`.

3.6 CONTROLADORES

Controladores de página têm como responsabilidade disponibilizar dados e métodos que possam ser referenciados por uma página *web* para entrada de dados, apresentação de dados ou processamento de eventos. Em JSF os controladores são chamados de *managed beans* ou ainda de *backing beans*.

Em JSF existem dois tipos de controladores: aqueles que não referenciam as classes de componente no código Java e os que referenciam. Os controladores que referenciam são chamados *backing beans* e a referência à classe de componente é chamada vínculo a componente ou *Component Binding*. Os controladores de página que não referenciam componentes são chamados *Plain Old Java* Objects (POJOs) e apenas utilizam vínculo a dados (*Data Binding*) e vínculo a métodos (*Method Binding*).

As novidades do JSF também englobam os controladores de página, começando pela configuração do *backing bean* que não precisa mais ser feita no arquivo `faces-config.xmI`. Só é preciso usar a anotação @ManagedBean e alguma das anotações de escopo.

3.6.1 Acesso Programático de Recursos

Recursos também são acessíveis no *backing bean*. Através do objeto ResourceHandler em `FacesContext.getApplication()` é possível buscar a referência de uma imagem, de um arquivo Javascript ou de um arquivo CSS. Isto permite, por exemplo, a criação dinâmica de componentes em páginas que dependem de recursos ou a alteração das imagens de algum componente que já esteja na árvore JSF.

3.6.2 Tratamento de Exceções

Em JSF 1.x, para que exceções não verificadas, filhas de RuntimeException, não fossem escondidas pelo *framework*, era necessário utilizar *try/catch* em todos os métodos de tratamento de eventos, o que deixava o código repetitivo e pouco legível. No JSF 2 o *Backing bean* não precisa mais se preocupar com isto, pois a classe ExceptionHandler relança exceções para que elas possam ser tratadas pelo servlet configurado no `web.xml` ou por qualquer outra classe relacionada a ExceptionHandler.

No JSF 2 agora também existem parâmetros de contexto no `web.xml` que configuram o comportamento do *framework* para que seja igual ao da versão 1.x do JSF.

3.6.3 Escopos Novos

Baseado no Richfaces e no Icefaces e inspirado no Rails, dois novos escopos chamados view e flash foram criados, além dos tradicionais request, session e application que já existiam na versão 1.x. O escopo de view é iniciado quando a árvore de componentes de uma página JSF é criada e só encerra quando acontece

navegação para outra página JSF. É um escopo bastante prático, pois quando se tem uma única pagina em uma funcionalidade, basta definir este escopo no *backing bean*. Não é necessário gerenciar este controlador em um escopo maior, o que normalmente causa desconforto ao desenvolvedor, ou no caso de não ser gerenciado, controladores ficam instanciados na aplicação por mais tempo que o necessário.

Existe também o escopo flash. Ele resolve o problema clássico de passagem de parâmetros para a resposta via redirecionamento e não encaminhamento. Por fim, ainda é possível criar escopos personalizados como o escopo de conversação, disponível no JBoss Seam e na JSR 299.

3.6.4 Novos Eventos

Além das páginas com a *tag* `<f:event>`, agora também é possível assinar eventos de sistema nos controladores. Outra possibilidade é assinar e publicar eventos personalizados. Por exemplo, métodos que realizam publicação e subscrição de eventos estão disponíveis na classe AbstractManager.

3.6.5 Simplificação De Configuração

Como já foi citado, a configuração de um *backing bean* com anotações deixou tudo muito simples. Através da própria anotação *@ManagedBean* é possível definir por qual nome o controlador será referenciado nas expressões EL em páginas JSF. Caso o atributo *name* não seja preenchido, o controlador, por convenção, é referenciado pelo nome da classe com a primeira letra em minúsculo.

Outra novidade é que também que é possível injetar outros *backing beans* via anotação com @ManagedProperty, algo que tinha que ser configurado no `faces-config.xml` até a versão 1.x do JSF. Assim a configuração via XML passa a ser uma opção disponível e preferencial, tal como acontece com EJB 3: se o *backing*

bean for declarado com anotações e no XML, prevalece o segundo. O Quadro 14 mostra como é feita a declaração de um *backing bean* no JSF 1.x.

```
faces-config.xml
<managed-bean>
 <managed-bean-name>managedUsuario</managed-bean-name>
 <managed-bean-class>br.edu.utfpr.blasterMusic.managedBeans.ManagedUsuario</managed-bean-class>
 <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

QUADRO 14 - Configuração de um *backing bean* no JSF 1.x.

O Quadro 15 mostra como a declaração do mesmo *backing bean* é feita no JSF 2.

```
@ManagedBean(name="managedUsuario")
@SessionScoped
public class ManagedUsuario {
...
}
```

QUADRO 15 - Configuração de *backing bean* simplificada no JSF 2 através do uso de anotações.

Comparando os quadros 14 e 15 é possível notar claramente a melhoria da versão 2 em relação a 1.x no que diz respeito a configuração dos *backing beans* uma vez que não é mais necessário configurar um arquivo XML pois tudo isso pode ser feito diretamente na classe.

3.6.6 Listas Fáceis

Duas das melhorias da versão 2 agradam muito os desenvolvedores acostumados a trabalhar com a versão 1.x: a possibilidade de usar *generics* no DataModel e também de popular um componente de seleção sem a necessidade de criar uma lista de SelectItem no *backing bean*. Isso facilita bastante as interações da página JSF com a lista de itens do *backing bean* e também assegura que instâncias de outras classes não sejam inseridas nessas listas.

Nos componentes de seleção populados a partir do *backing bean*, agora basta apenas passar a lista de JavaBeans diretamente. As *tags* dos componentes

de seleção, como `<h:selectOneMenu>`, passam a suportar a *tag* `<f:seIectItems>` com as propriedades *var*, *itemLabel*, *itemValue* e *noSelectionValue*. Dessa maneira, a definição dos itens de seleção fica bastante parecida com a iteração sobre a lista feita nos tempos do JSTL ou a solução do Struts 1 para o mesmo problema.

3.7 NAVEGAÇÃO

No JSF as regras de navegação são configuradas no arquivo faces-config.xml indicando a página de origem, a página de destino e quando o evento ocasiona a transição. O identificador do evento é chamado *outcome* e deve ser devolvido como retorno em métodos de ação ou diretamente informado no atributo *action* de comandos de ação. Essa é a chamada configuração de navegação explícita presente na versão 1.x do JSF. O Quadro 16 mostra como é configurado esse tipo de navegação.

```
faces-config.xml
<navigation-rule>
 <from-view-id>/pages/telaPrincipal.xhtml</from-view-id>
 <navigation-case>
  <from-outcome>logoutOk</from-outcome>
  <to-view-id>/pages/sucessoLogout.xhtml</to-view-id>
  <redirect/>
 </navigation-case>
</navigation-rule>
```

QUADRO 16 - Configuração de navegação explícita oferecida pelo JSF 1.x.

Nos esforços de diminuir a quantidade de configuração necessária, o JSF 2 trouxe duas novas alternativas para configurar navegação entre páginas: a navegação implícita e a navegação condicional.

3.7.1 Navegação Implícita

A navegação implícita parte do princípio de que se o *outcome* corresponder ao nome da pasta e do arquivo destino sem o seu sufixo, não é necessário fazer nenhuma configuração no arquivo `faces-config.xml`. Ou seja, se ao executar uma *action* e nenhuma regra de navegação bater com o *outcome* devolvido, o JSF buscará uma página com o nome desse *outcome* e a exibirá.

Por exemplo, se o botão de comando apresentado no Quadro 17 devolver o *outcome* "login" e nenhuma regra de navegação existir, o JSF irá procurar uma página `login.xhtml` para exibir.

```
minhaPagina.xhtml
<!-- Botão que redireciona para a página login.xhtml -->
<h:commandButton value="Login" action="login" />
```

QUADRO 17 - Navegação implícita no JSF 2.

Usar navegação implícita simplifica muito a configuração do fluxo das páginas e também deixa o arquivo `faces-config.xml` mais enxuto. Porém, para os desenvolvedores que gostam do diagrama de fluxo entre as páginas gerado pelo plugin JBoss Tools do Eclipse e pelo NetBeans, este recurso pode não ser tão interessante.

3.7.2 Navegação Condicional

A navegação condicional do JSF 2 permite que uma condição seja usada para definir se a navegação acontecerá da página de origem para a página de destino. Para isso, a *tag* `<if>` foi adicionada como filha de `<navigation-case>`, que é mostrada no Quadro 18, e fez com que a *tag* `<from-outcome>` passasse a ser opcional. Dentro da *tag* `<if>` podem ser adicionadas expressões EL. Ao configurar uma <navigation-rule> no XML é possível apontar um resultado booleano que indica se aquela regra deve ser executada ou não. O Quadro 18 ilustra um

exemplo do uso da *tag* invocando um método de um *backing bean* que retorna um valor lógico.

```
faces-config.xml
<navigation-rule>
  <from-view-id>/minhaPagina.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>login</from-outcome>
    <to-view-id>/login.xhtml</to-view-id>
    <if>#{managedUsuario.naoLogado}</if>
  </navigation-case>
</navigation-rule>
```

QUADRO 18 - Exemplo de configuração da navegação condicional no JSF 2.

3.8 FERRAMENTAS DE DESENVOLVIMENTO

Tão importante quanto ter um *framework* para o desenvolvimento de aplicações *web* é ter também uma ferramenta que ajude o desenvolvedor a extrair o máximo dos recursos e benefícios oferecidos por ele (HALL, 2004). Para isso, estão disponíveis no mercado vários Ambientes de Desenvolvimento Integrado (IDE - *Integrated Development Environment*) de excelente qualidade que suportam a versão 2 do JSF e os vários benefícios que ela trouxe.

Cada IDE oferece comodidades diferentes e fica a critério do desenvolvedor escolher aquela que mais lhe agrada. Porém, todas são capazes de promover uma aceleração considerável no processo de desenvolvimento e colocam à disposição do desenvolvedor uma série de ferramentas para lhe ajudar ao longo do projeto.

3.8.1 Eclipse 3.6

O Eclipse é um dos principais IDEs e é desenvolvido pela Eclipse Foundation, um consórcio de grandes empresas e membros da comunidade que financiam os projetos da fundação através de uma contribuição anual e que também ajudam a definir seu futuro, além de hospedar e incubar os vários projetos da ferramenta (ECLIPSE, 2011).

Infelizmente, o suporte nativo a JSF 2 que o Eclipse oferece é muito limitado e por esse motivo o mais recomendado para quem deseja desenvolver utilizando esse

framework é instalar um *plugin* chamado JBoss Tools para suprir as deficiências do Eclipse nessa área. O JBoss Tools é gratuito e o seu download pode ser feito através do site dos seus desenvolvedores ou a partir do próprio Eclipse através do Eclipse Market. É importante ressaltar que também é necessário instalar um servidor que ofereça suporte a JEE 6 ou a JSF, tal como Glassfish 3 ou Tomcat 7 respectivamente, e configurá-lo dentro do Eclipse.

Um dos recursos mais vantajosos do JBoss Tools com certeza é o seu editor visual de páginas JSF que pode ser visto em ação na Figura 9 e suporta as várias *tags* novas da versão 2.

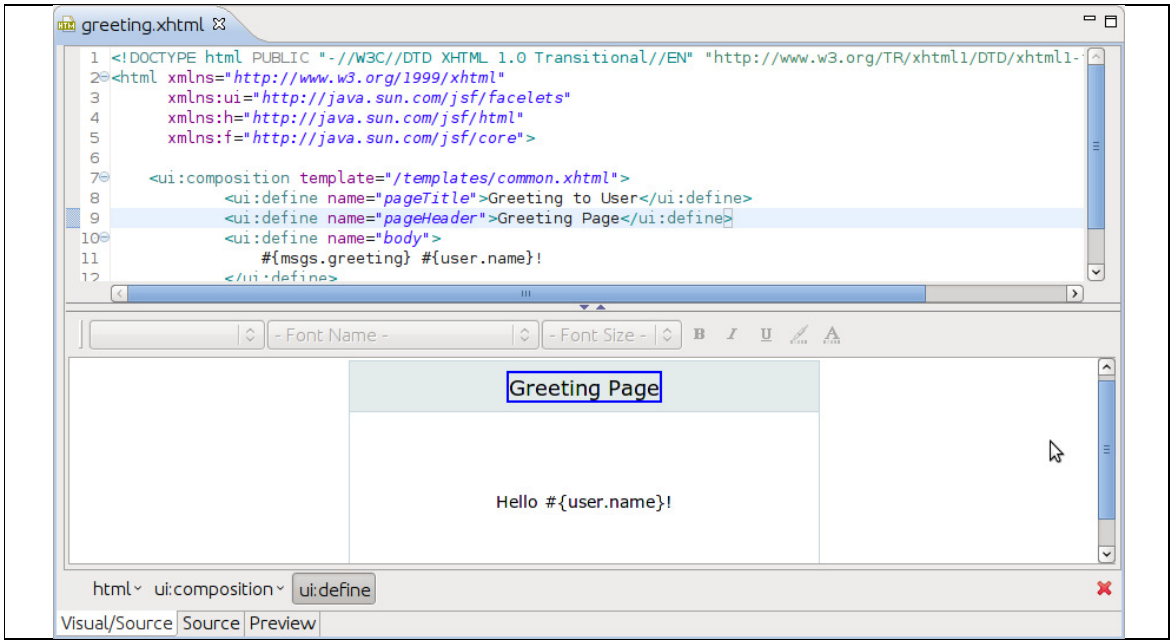

FIGURA 9 - Editor visual de páginas JSF do JBoss Tools.

Através do editor visual é possível visualizar em tempo real as alterações que são feitas no código da página. É possível também gerar o código automaticamente arrastando componentes visuais da paleta diretamente para o *canvas* do editor. A paleta de componentes é mostrada na Figura 10 e oferece também suporte à biblioteca de componentes Richfaces, entre outras.

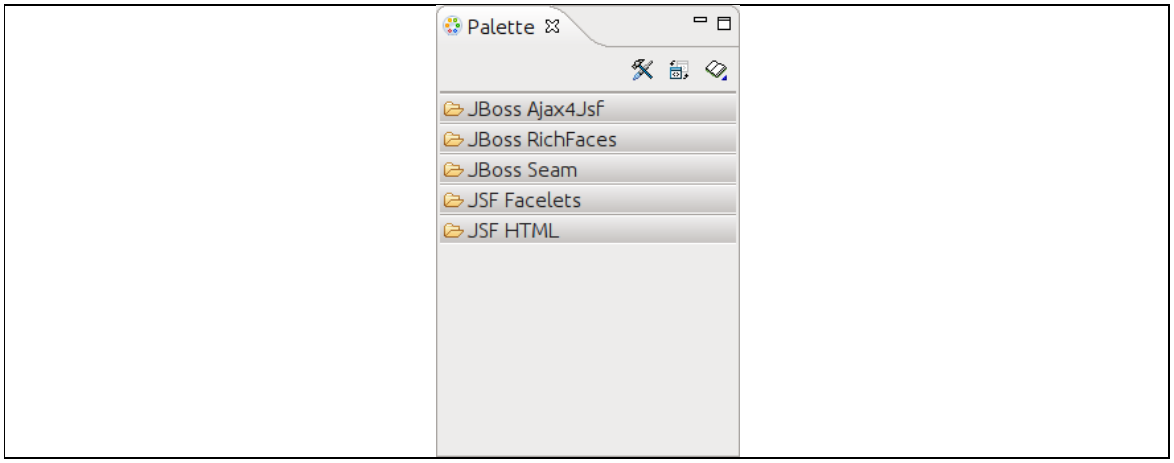

FIGURA 10 - Paleta que contém os componentes JSF que podem ser arrastados para a tela do editor visual.

Já o editor de código oferece suporte às novas *tags* introduzidas pelo JSF 2 bem como à utilização e criação de *composite components* e seus atributos como pode ser visto na Figura 11.

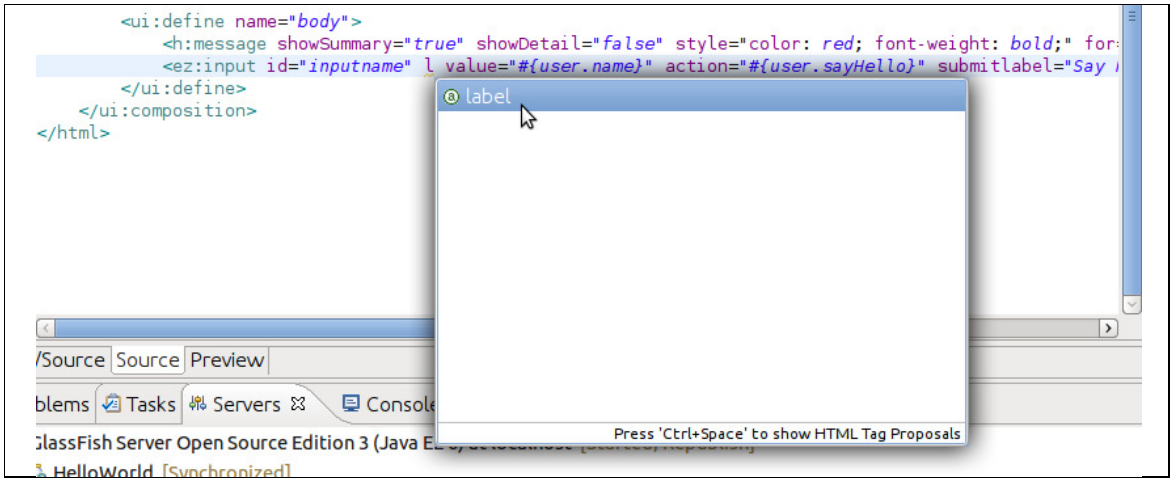

FIGURA 11 - Utilização de *composite component* em uma página JSF é suportada.

Além de tudo, o editor ainda oferece suporte nativo à Facelets, e possibilita a utilização de todas as suas *tags*.

Outra facilidade oferecida é na hora de utilizar o novo recurso de navegação condicional. O editor onde é feita a configuração de navegação oferece suporte à nova *tag* `<if>`, como mostra a Figura 12.

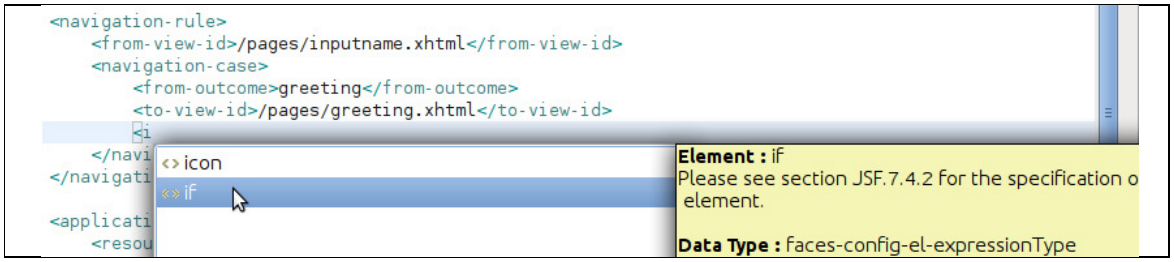

FIGURA 12 - Suporte à navegação condicional.

O editor também oferece a opção de configurar visualmente as regras de navegação empregadas na aplicação, além de mostrar as ligações entre as páginas que já possuem alguma regra. A Figura 13 mostra duas páginas ligadas por uma regra de navegação, bem como a paleta de opções no canto esquerdo. As páginas envolvidas na regra podem ser arrastadas diretamente da árvore de navegação dos arquivos do projeto ou adicionadas através de um dos botões da paleta.

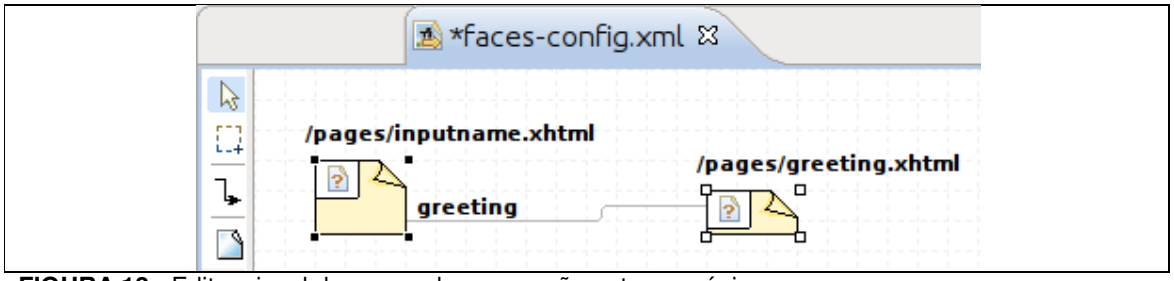

FIGURA 13 - Editor visual de regras de navegação entre as páginas.

E por fim, o editor de código Java disponibiliza todas as anotações introduzidas na versão 2, facilitando a configuração de *backing beans*, validadores e convertedores. A Figura 14 mostra algumas anotações sendo usadas na classe de um *backing bean*. O JBoss Tools também possui um assistente para criação de novos projetos que pode ser utilizado para gerar o esqueleto de uma aplicação simples, porém bem estruturada, com páginas, *templates*, configurações e um *backing bean*.

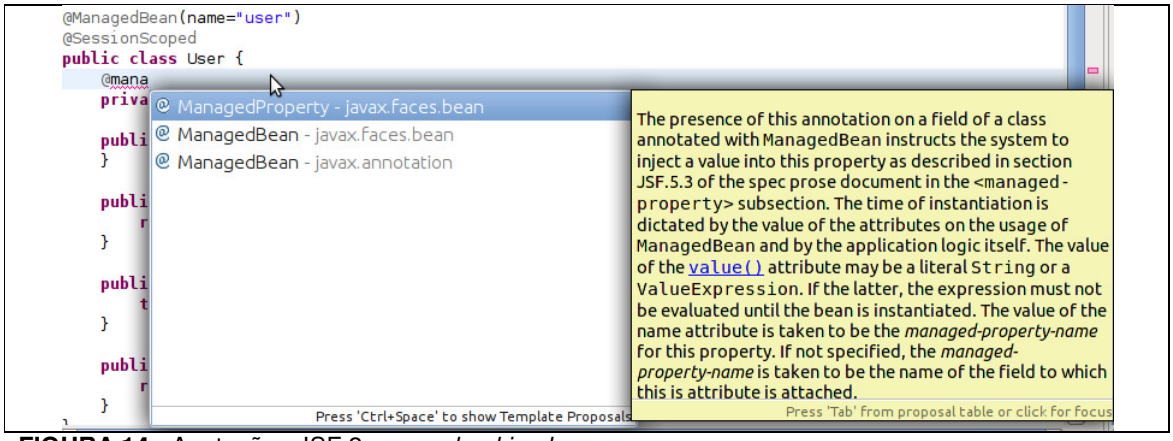

FIGURA 14 - Anotações JSF 2 em um *backing bean*.

O suporte oferecido pelo Eclipse através do *plugin* JBoss Tools é bem completo e abrange vários dos novos recursos introduzidos pelo JSF 2. O editor visual de páginas JSF com certeza é a sua maior vantagem pois além de mostrar a página em tempo real ele também oferece a possibilidade de gerar o código para o desenvolvedor. Além disso, até o presente momento o JBoss Tools é o único que oferece este recurso. A desvantagem é que o *plugin* utiliza uma quantidade considerável de memória para isso o que as vezes pode ocasionar travamentos.

3.8.2 IntelliJ IDEA 10.5

O IntelliJ IDEA é mantido pela JetBrains, uma empresa da República Checa criada no ano de 2000 e que disponibiliza uma vasta suíte de programas que oferecem soluções inteligentes para os mais variados problemas dos desenvolvedores. O IntelliJ está disponível em duas versões, Community e Ultimate, sendo que esta custa cerca de 250 dólares. A versão gratuita infelizmente não oferece suporte a JSF 2, então o desenvolvedor que deseja utilizar esta tecnologia no IntelliJ só poderá fazê-lo mediante pagamento (JETBRAINS, 2011).

Começando pelo editor de páginas, o IntelliJ oferece suporte à todas as novas *tags* do JSF 2. A Figura 15 mostra a *tag* `<h:outputStylesheet>` sendo usada e como é possível preencher o valor dos atributos automaticamente de acordo com o conteúdo disponível dentro da pasta *resources*.

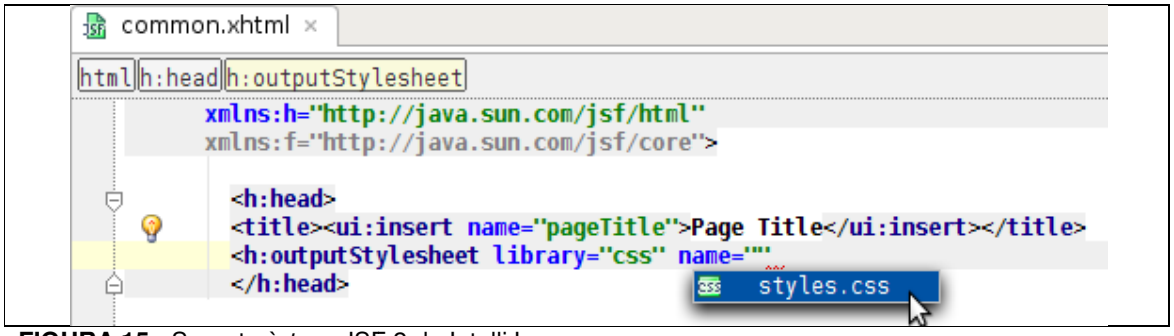

FIGURA 15 - Suporte à *tags* JSF 2 do IntelliJ.

Apesar de não oferecer um editor visual integrado, o IntelliJ disponibiliza atalhos para os navegadores instalados na máquina diretamente no seu editor de páginas como pode ser visto na Figura 16.

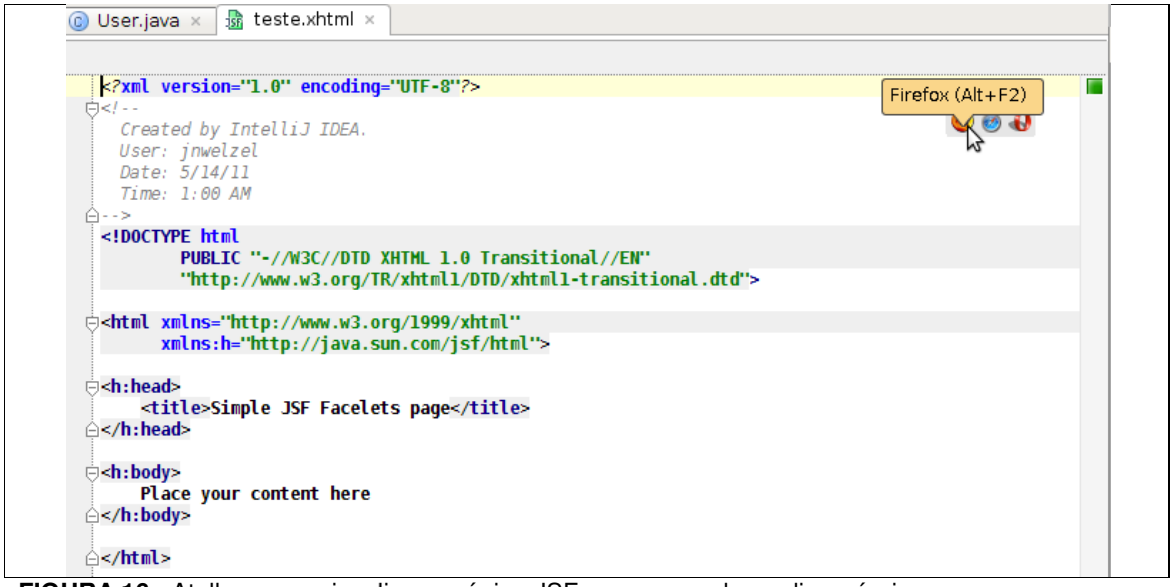

FIGURA 16 - Atalhos para visualizar a página JSF nos navegadores disponíveis.

O editor de páginas também oferece suporte à criação e uso de *composite components* como pode ser visto na Figura 17. Os atributos dos componentes são mostrados dentro de suas *tags* conforme foram configurados na interface no componente. Os valores desses atributos também podem ser preenchidos automaticamente.

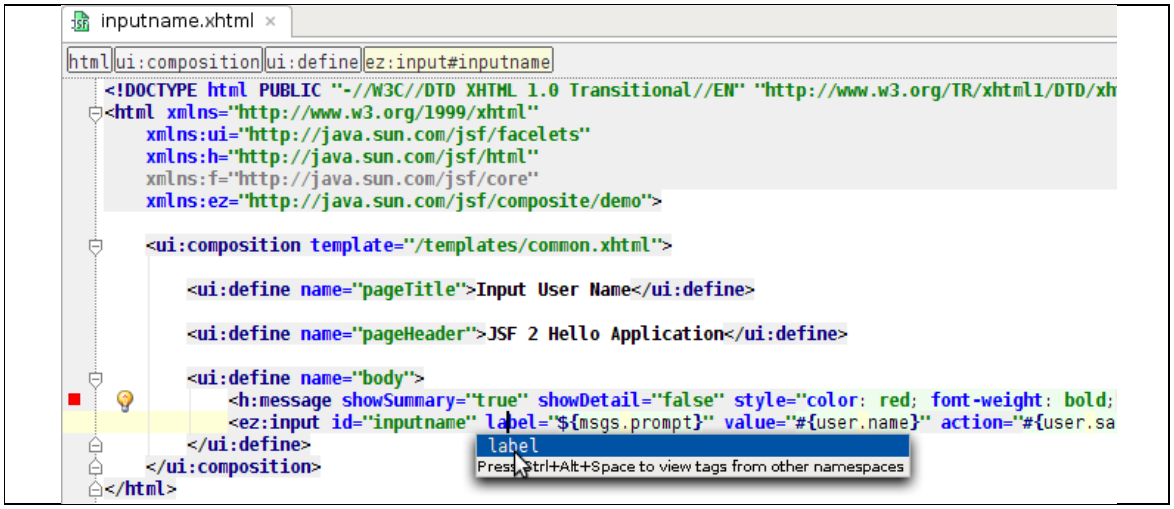

FIGURA 17 - Uso de *composite component* em uma página JSF no IntelliJ.

Além disso o editor ainda permite manipular na página o objeto implícito flash que representa um dos novos escopos introduzidos no JSF. A Figura 18 mostra como isso é feito.

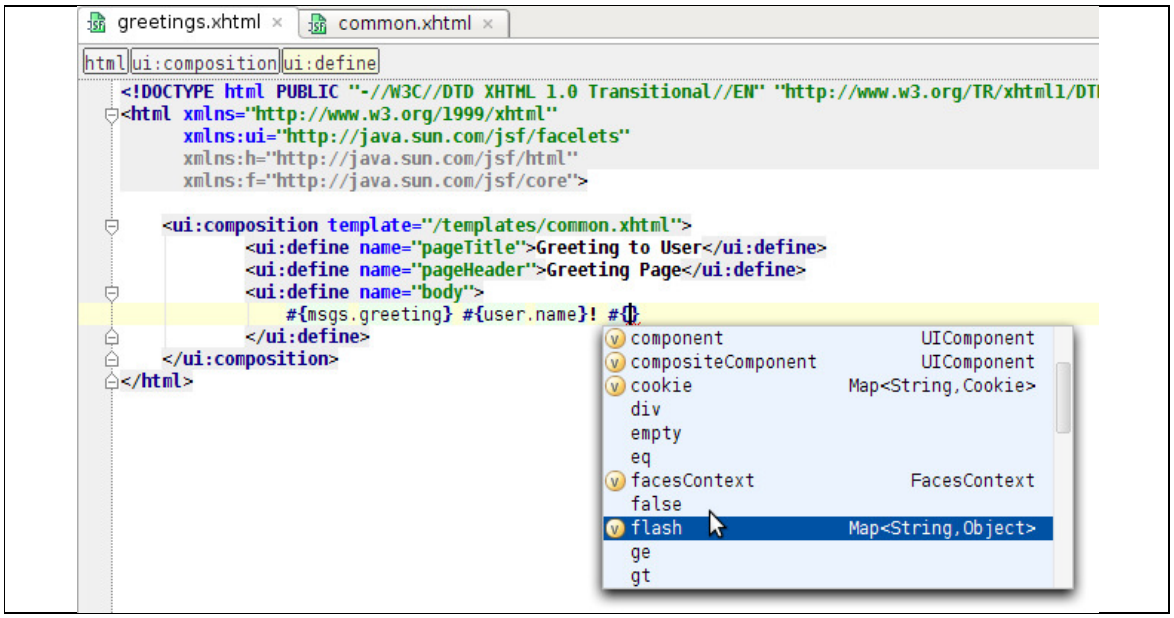

FIGURA 18 - Uso do objeto implícito flash em uma página JSF.

O IntelliJ oferece ainda recursos para configuração de regras de navegação em geral e para navegação condicional através da *tag* `<if>`. As configurações de navegação podem ser feitas através do editor de texto ou através do editor visual, assim como é feito no Eclipse. A Figura 19 mostra essa funcionalidade.

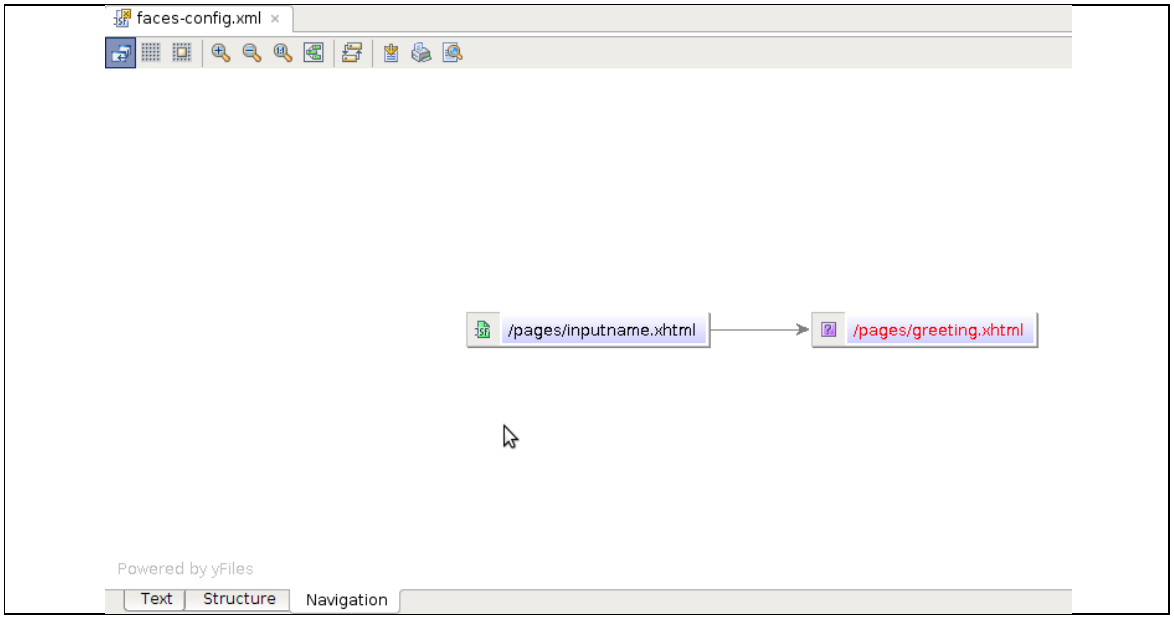

FIGURA 19 - Editor visual de navegação entre as páginas JSF no IntelliJ.

Da mesma maneira como o editor de código Java do Eclipse suporta as novas anotações do JSF, o do IntelliJ não é diferente. Oferece suporte às anotações de *backing beans*, validadores e conversores. A Figura 20 mostra um *backing bean* com suas respectivas anotações.

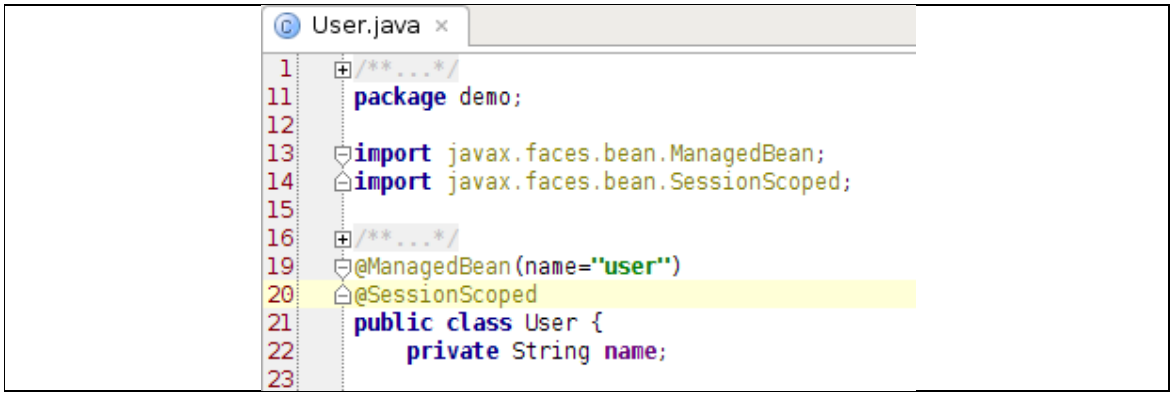

FIGURA 20 - Suporte do IntelliJ às novas anotações do JSF 2.

Um recurso muito útil que o IntelliJ oferece é uma janela dedicada que mostra de forma categorizada todo o conteúdo do projeto. As categorias abrangem arquivos de configuração, *backing beans*, *composite componentes* e mostram todos os arquivos que pertencem àquela categoria. Dessa maneira é possível ter uma visão completa e organizada dos artefatos que compõem a aplicação. A Figura 21 mostra como funciona essa janela.

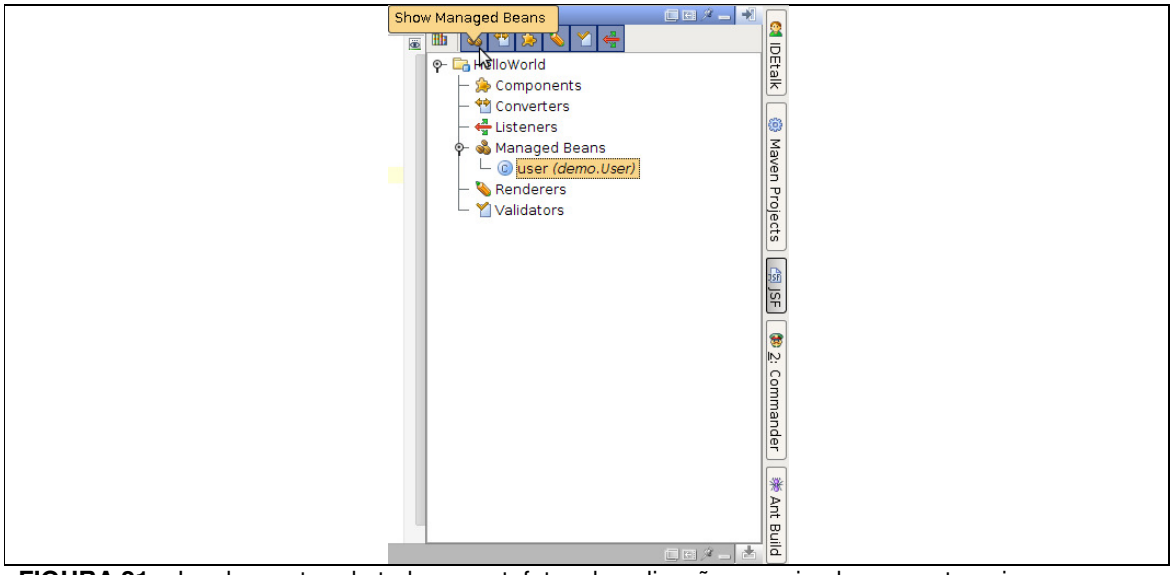

FIGURA 21 - Janela mostrando todos os artefatos da aplicação organizados por categorias.

Outro recurso muito interessante oferecido pelo IntelliJ é que antes de excluir ou alterar qualquer conteúdo, seja relacionado à JSF ou não, ele mostra uma lista dos locais onde esse arquivo é usado e já sugere um refatoramento levando em consideração o arquivo que está sendo excluído. Isso ajuda o desenvolvedor a ter uma ideia mais precisa das repercussões que as alterações em um determinado trecho de código podem causar, prevenindo assim complicações futuras.

3.8.3 NetBeans 6.9.1

O NetBeans foi criado pela, hoje extinta, Sun Microsystems e atualmente é mantido pela Oracle como um projeto de código aberto para oferecer à comunidade uma IDE gratuita e de alta qualidade. Por ser desenvolvida pela companhia responsável pela criação do JSF, o NetBeans possui uma integração muito grande não somente com o *framework* JavaServer Faces, mas também com uma série de outras tecnologias e ferramentas ligadas à linguagem Java e a outras linguagens também (NETBEANS, 2011).

Primeiramente, o assistente para criação de novos projetos é muito completo e com poucos cliques é possível obter uma aplicação completamente configurada,

pronta para ser desenvolvida. O assistente ainda conta com uma vasta lista de projetos prontos para serem usados como referência para os desenvolvedores que assim desejarem. Uma vez que o projeto está criado, também é possível criar páginas JSF a partir de entidades do banco de dados, um recurso muito útil na hora montar as telas CRUD da aplicação e agilizar o desenvolvimento. Essa opção também gera os arquivos de apoio relacionados à entidade selecionada, tais como conversores, *backing bean*, folha de estilo, arquivo Javascript e arquivos de mapeamento objeto-relacional. A Figura 22 mostra essa opção.

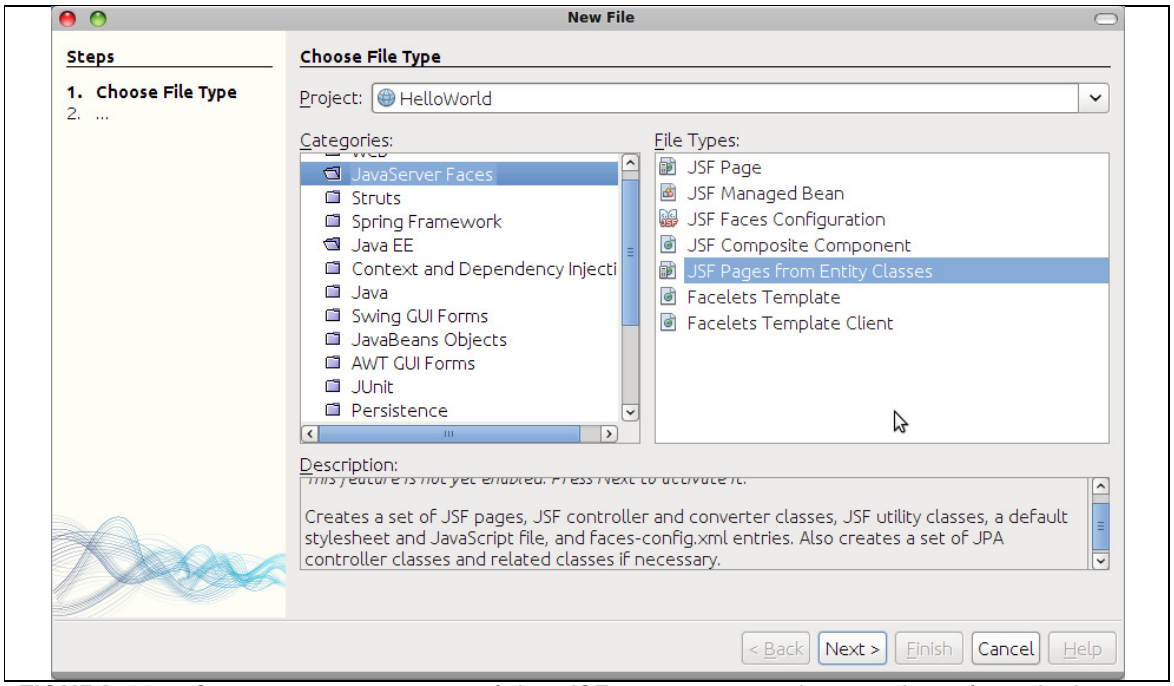

FIGURA 22 - Opção para gerar uma página JSF e seus respectivos artefatos à partir de uma entidade do banco de dados.

O NetBeans também oferece suporte às novas *tags* do JSF 2 bem como às do Facelets como a Figura 23 mostra.

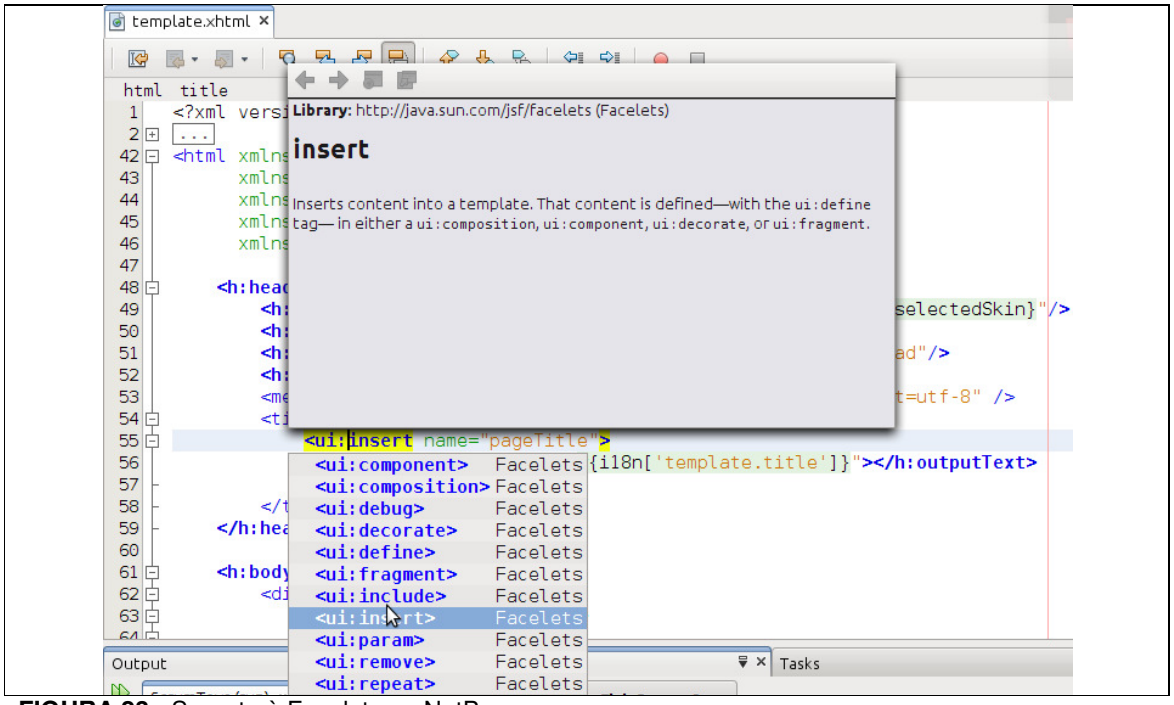

FIGURA 23 - Suporte à Facelets no NetBeans.

O NetBeans oferece suporte também à criação e uso de *composite components* com sugestões de código baseadas na estrutura do componente para facilitar seu uso no código das páginas JSF. A Figura 24 mostra esse recurso sendo usado no editor.

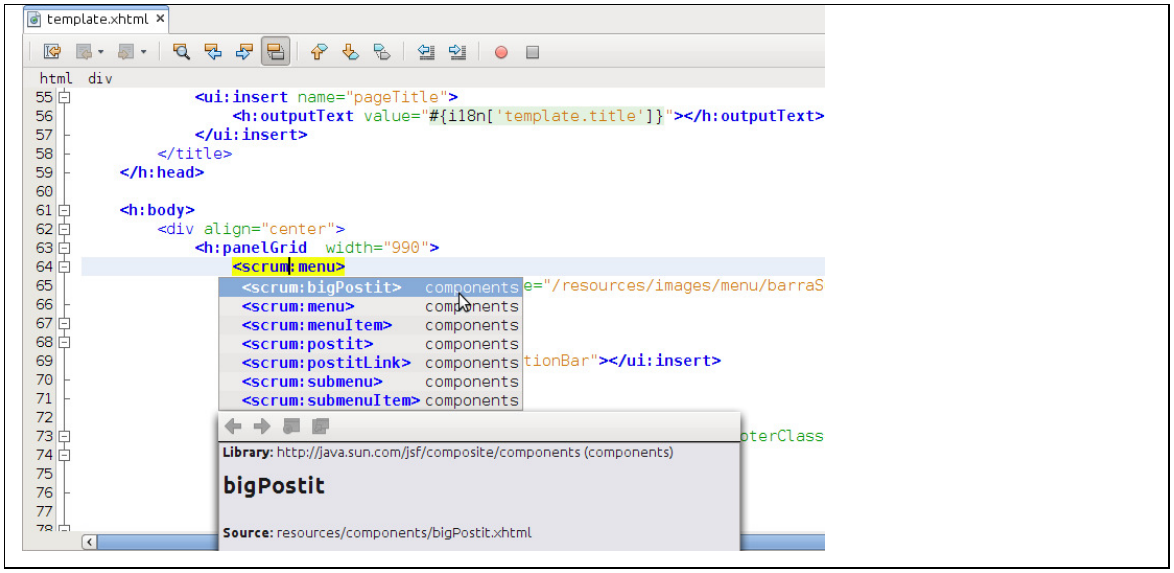

FIGURA 24 - Os *composite components* disponíveis na aplicação de referência *ScrumToys* do NetBeans.

Outro recurso interessante na hora de criar os *templates* do Facelets é a possibilidade de selecionar um dos vários layouts pré-configurados do NetBeans. Essa opção gera automaticamente a página JSF com o layout escolhido juntamente com o arquivo de estilos na pasta *resources* caso tenha sido selecionada a opção *CSS*. A Figura 25 mostra com mais detalhes essa opção.

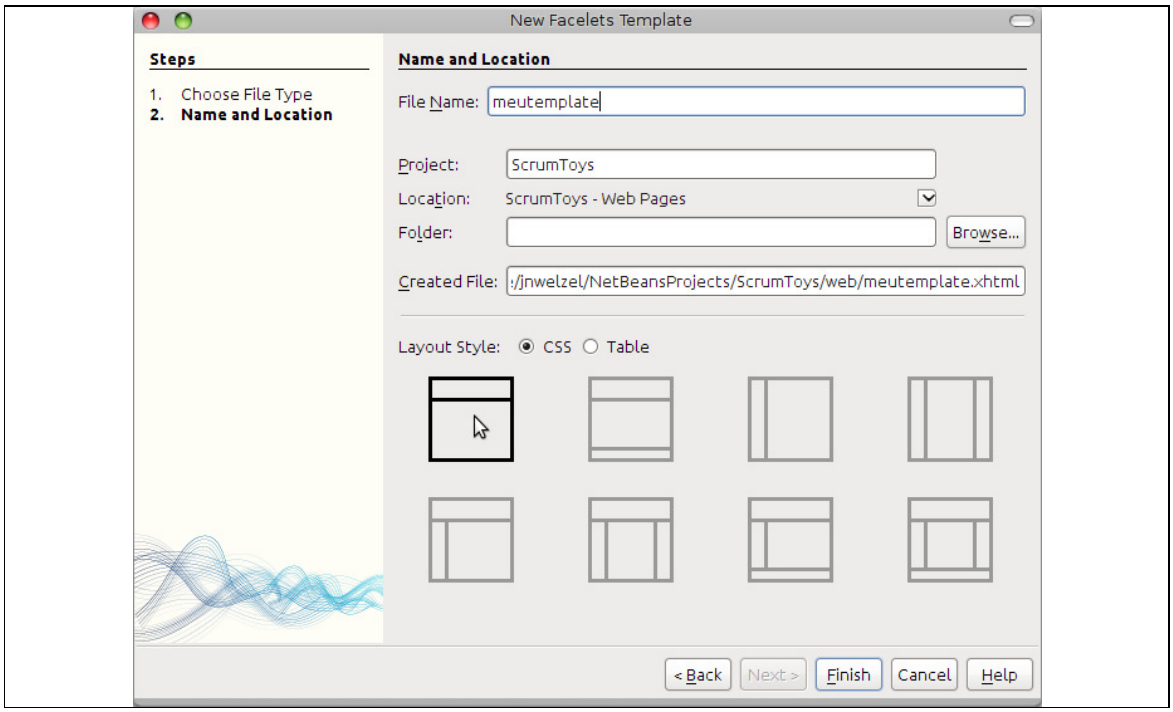

FIGURA 25 - Recurso que auxilia na criação de layouts para *templates* do Facelets.

Assim como as demais IDEs, o NetBeans também suporta o uso do novo objeto implícito do escopo *flash* nas páginas JSF. O NetBeans também conta com um editor de regras de navegação do `faces-config.xml` que suporta a nova *tag* `<if>` do recurso de navegação condicional bem como um modo gráfico de edição das regras. O NetBeans suporta ainda as novas anotações do JSF 2 referentes a *backing beans*, validadores, conversores, etc.

Das IDEs analisadas o NetBeans oferece as soluções mais completas e abrangentes e além disso possui uma forte integração com servidores de aplicação e serviços de banco de dados, componentes vitais em uma aplicação *web*. E o melhor de tudo é que assim como o Eclipse ele também é totalmente gratuito.

4 ESTUDO DE CASO

Nesse capítulo é desenvolvida uma aplicação JSF para gerenciar as auditorias de uma empresa utilizando a versão 1.2 do JavaServer Faces, em conjunto com a biblioteca de componentes Richfaces 3.2. Através de diagramas e de trechos de código extraídos da aplicação são mostrados os seus principais casos de uso e como o sistema funciona internamente para oferecer os devidos recursos a seus usuários.

Dessa maneira, é possível demonstrar na prática vários dos recursos oferecidos pelo JavaServer Faces bem como os benefícios que ele traz para o processo de desenvolvimento de uma aplicação *web*. Além disso, comprova os benefícios de estender as funcionalidades do *framework* através de bibliotecas de componentes e como é fácil integrar o JSF com outros *frameworks*, o que o torna uma ferramenta ainda mais útil.

4.1 TECNOLOGIAS UTILIZADAS

Para o desenvolvimento da aplicação de estudo de caso foram utilizadas as seguintes tecnologias:

- *Framework* JavaServer Faces 1.2 como plataforma de desenvolvimento, fornecendo as APIs necessárias para a construção da interface com o usuário bem como das regras de negócio.
- Facelets para criação de páginas JSF e *layouts*.
- Biblioteca de componentes Richfaces 3.2 para oferecer integração entre JSF e AJAX bem como uma série de componentes visuais avançados.
- Linguagem Java para implementação do código da aplicação.
- Eclipse IDE 3.5 como ambiente de desenvolvimento integrado em conjunto com o *plugin* JBoss Tools 3.1.

- *Framework* Hibernate 3.2.1 para persistência de dados, validação e mapeamento objeto-relacional em conjunto com o sistema gerenciador de banco de dados (SGBD) PostgreSQL versão 8.2.
- Apache Tomcat 6 como *servlet container* para executar a aplicação.

4.2 GERENCIADOR DE AUDITORIAS

A ideia de desenvolver uma aplicação para gerenciar auditorias surgiu da necessidade que algumas empresas da região de Medianeira tinham de controlar o ciclo de vida de suas auditorias e de agilizar o processo de elaboração das mesmas. Foi proposta então a criação de um sistema que oferecesse essas funcionalidades visando auxiliar o processo de planejamento, execução e avaliação de resultados das auditorias.

A auditoria oferece uma avaliação sistemática, detalhada e independente das atividades desenvolvidas em um setor ou área da empresa e o seu objetivo é verificar se os mesmos estão de acordo com os itens estabelecidos pela norma de avaliação. Caso elas não estejam, o sistema possibilita a criação de não-conformidades a partir da auditoria. As funcionalidades principais do sistema são demonstradas no diagrama de Caso de Uso da Figura 26.

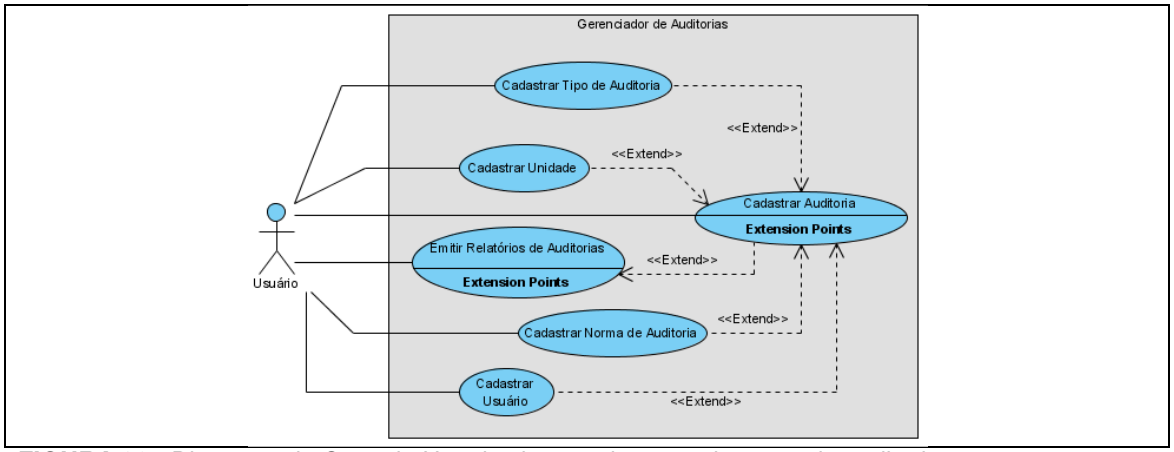

FIGURA 26 - Diagrama de Caso de Uso do sistema de gerenciamento de auditorias.

Cada caso de uso representa uma funcionalidade do Gerenciador de Auditorias que pode ser acessada pelo usuário.

Os casos de uso **Cadastrar Tipo de Auditoria**, **Cadastrar Norma de Auditoria**, **Cadastrar Unidade** e **Cadastrar Usuário** são acessíveis de forma direta pelo ator Usuário, que representa o usuário que está logado no sistema e possui as permissões necessárias. Já o caso de uso **Cadastrar Auditoria** só pode ser acessado a partir do momento que os quatro casos de uso citados anteriormente forem executados. Ou seja, existe uma dependência sobre eles de tal maneira que para cadastrar uma auditoria é necessário antes executar os quatro casos anteriores. A Figura 27 mostra o menu de auditoria que é responsável por disponibilizar alguns dos casos de uso citados acima.

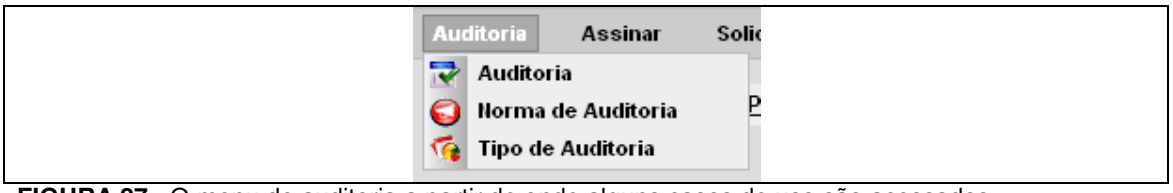

FIGURA 27 - O menu de auditoria a partir de onde alguns casos de uso são acessados.

Já o caso de uso **Emitir Relatórios de Auditorias** só pode ser executado com sucesso a partir do momento que houver ao menos uma auditoria cadastrada. Ele pode ser acessado, mas não trará nenhum resultado. A Figura 28 mostra como esses relatórios são acessados dentro do sistema.

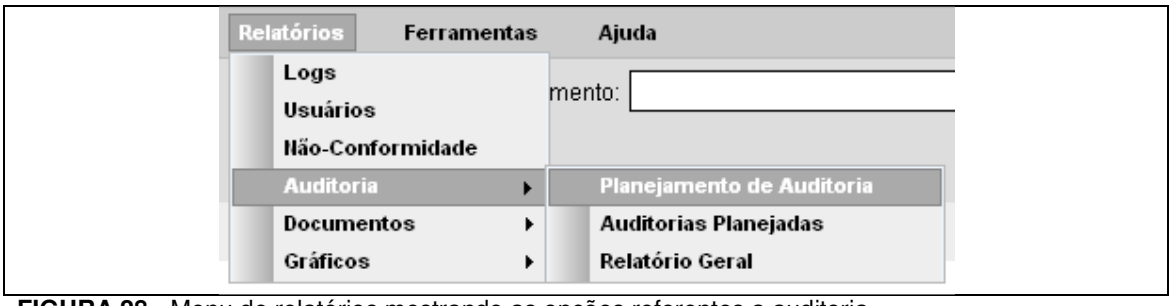

FIGURA 28 - Menu de relatórios mostrando as opções referentes a auditoria.

O caso de uso **Cadastrar Tipo de Auditoria** é responsável por registrar tipos de auditoria que possam ser executadas usando o sistema. Este caso de uso é

fundamental para o funcionamento do sistema pois o tipo de auditoria é usado para classificar a natureza da mesma (*i.e.* interna, externa, de sistemas, de processos).

O caso de uso **Cadastrar Norma de Auditoria** também é muito importante pois é responsável por coletar as informações que compõem uma norma. A sua finalidade é estabelecer os critérios ou a forma geral das táticas que o auditor tem que seguir. Estas táticas representam as regras de inspeção que o auditor implementa na busca de informações probatórias para alcançar um resultado. Nesse caso o usuário deve fornecer um nome para a norma e deve também informar pelo menos um item que faça parte dela.

Outro caso de uso fundamental para o funcionamento do sistema é o **Cadastrar Unidade**. Sua principal função é manter a estrutura de unidades e/ou setores da empresa. Esses dados por sua vez são usados para definir a abrangência da auditoria dentro da empresa. Essa estrutura está organizada em forma de árvore para facilitar a visualização da hierarquia existente entre as unidades e setores, como mostra a Figura 29.

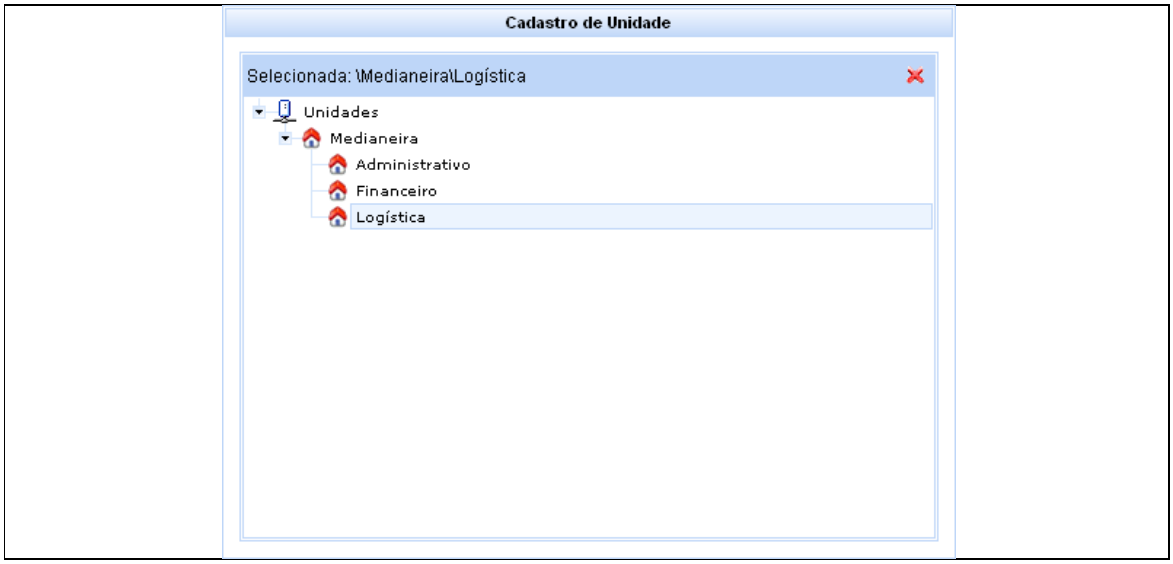

FIGURA 29 - Tela de cadastro de unidades/setores.

O último pré-requisito para efetuar o cadastro de uma auditoria é o caso de uso **Cadastrar Usuário**. Esse caso de uso é responsável por manter os usuários do sistema que poderão fazer parte de uma auditoria, ou seja, que irão compor a equipe auditora desta auditoria. A partir da lista de membros da equipe auditora

também são definidos os papéis de cada um dentro do escopo da auditoria pois é necessário que haja um líder e, se houver necessidade, um ou mais observadores.

O caso de uso **Cadastrar Auditoria** representa o ponto central da aplicação. Ele depende dos três casos de uso anteriores já que o tipo de auditoria define o perfil da mesma, a norma especifica todos os itens que serão avaliados nessa auditoria e as unidades e/ou setores indicam as áreas da empresa que serão auditadas. Esse caso de uso é responsável por manter todas as informações necessárias para o planejamento, execução e avaliação de resultados e por isso tem um papel tão fundamental.

O caso de uso **Emitir Relatórios de Auditoria** permite ao usuário visualizar e imprimir três tipos distintos de relatórios de auditoria. O primeiro tipo diz respeito as auditorias que estão em planejamento, ou seja, que ainda necessitam de aprovação. O segundo tipo de relatório é emitido a partir das auditorias que já foram planejadas e estão aguardando sua execução. Já o terceiro e último tipo traz um relatório de todas as auditorias que estão cadastradas no sistema. Apesar de abordarem tipos diferentes de auditorias, as três opções de relatórios oferecem telas de parâmetros para refinar a busca que é feita para gerar os resultados dos relatórios.

É importante notar que para acessar o Gerenciador de Auditorias é necessário, antes de tudo, autorização e validação de acesso ao Digitaldoc, e em segundo lugar o perfil de acesso do usuário deve possuir a permissão referente a parte de auditoria. Somente mediante essa autenticação e autorização no perfil ele poderá acessar os casos de uso discutidos anteriormente.

Na Figura 30, é possível observar como se dá a criação de uma auditoria e como os dados que foram coletados através dos casos de uso são usados nesse processo.

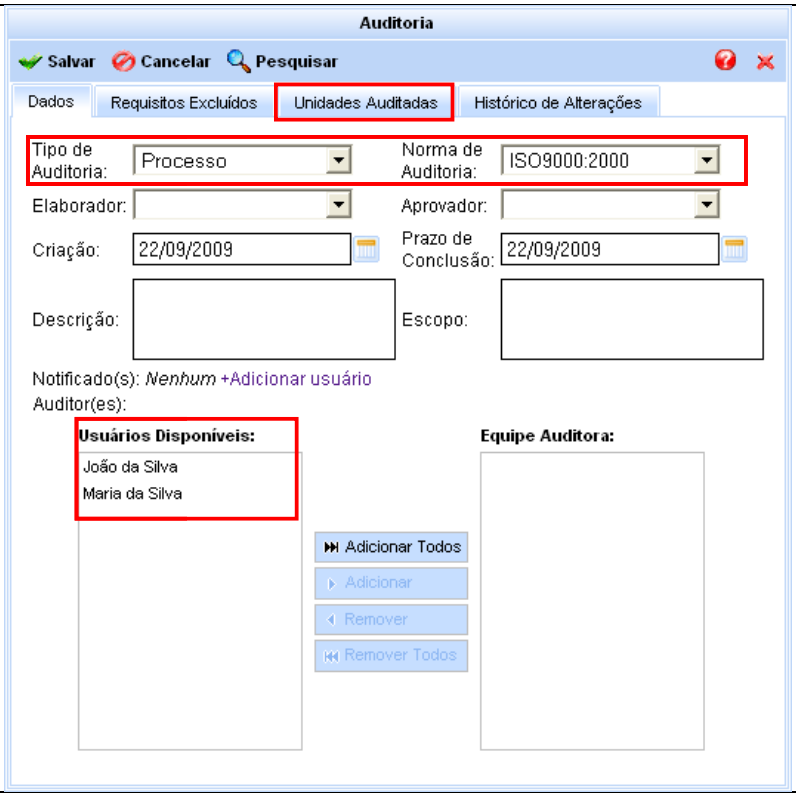

FIGURA 30 - Tela de cadastro de auditoria.

Para acesso a dados foi utilizado o *framework* de persistência e mapeamento objeto-relacional Hibernate em conjunto com o banco de dados PostgreSQL versão 8.2. A Figura 31 mostra as tabelas do sistema de gerenciamento de auditorias e seus relacionamentos.

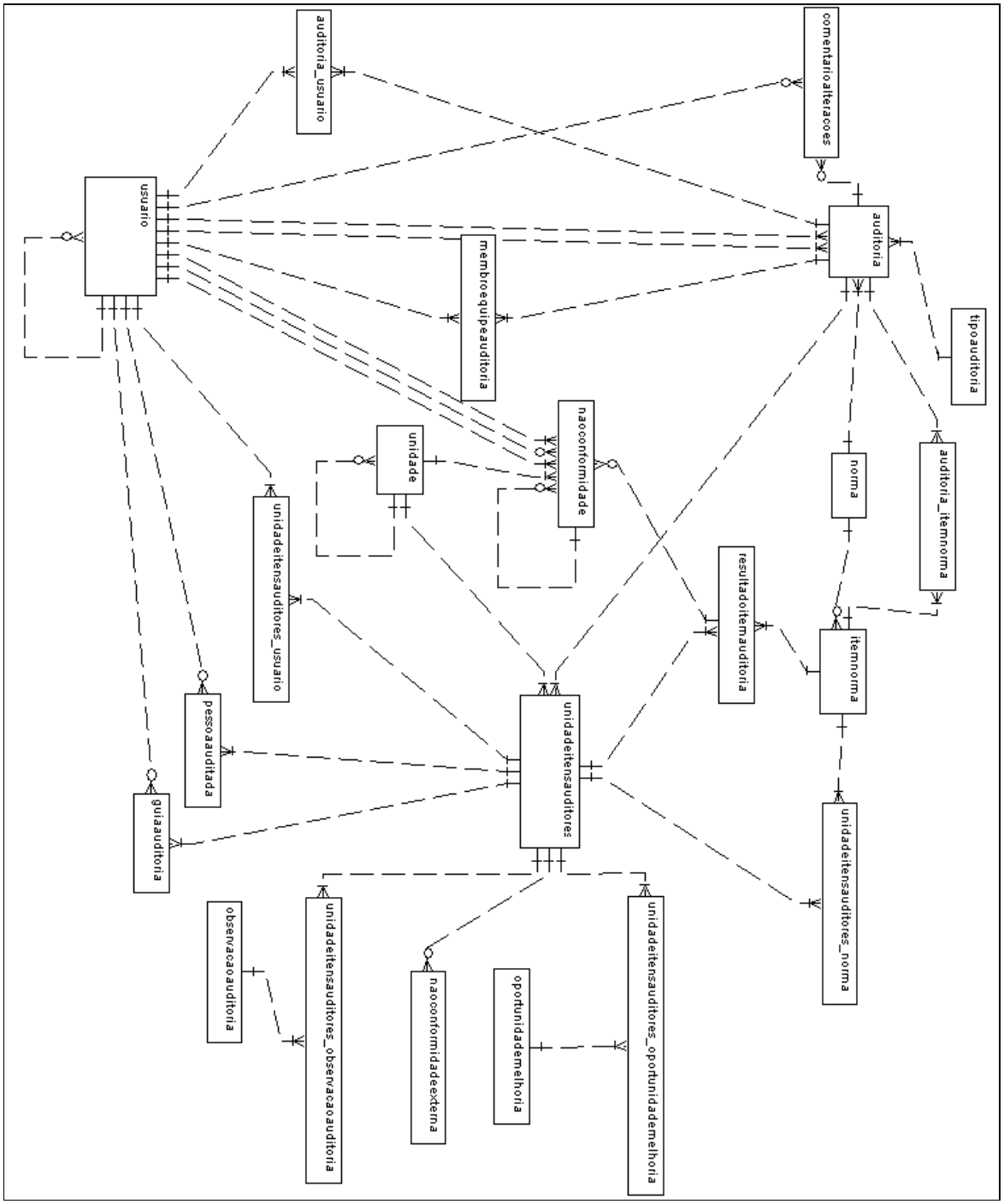

FIGURA 31 - Tabelas e relacionamentos das entidades do Gerenciador de Auditorias (DER).

O diagrama de classes mostrado na Figura 32 descreve a estrutura do Gerenciador de Auditorias mostrando suas classes (ou *beans*), atributos, métodos e relacionamentos.

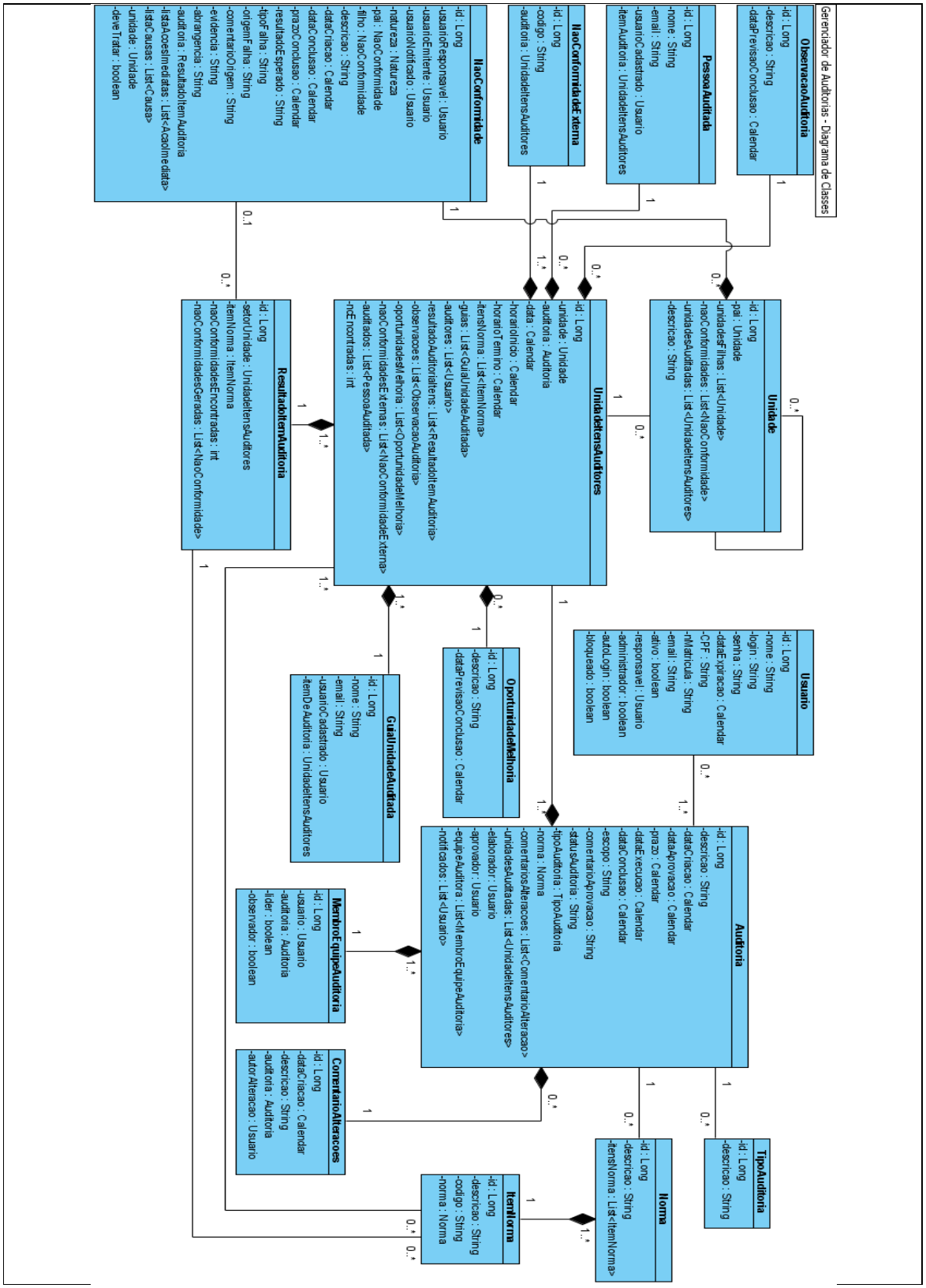

FIGURA 32 - Diagrama de classes do Gerenciador de Auditorias.

4.2.1 Ciclo de Vida

Em linhas gerais, o ciclo de vida de uma auditoria é tradicionalmente dividido em quatro etapas: planejamento, execução, avaliação da auditoria e elaboração de ações corretivas para as próximas auditorias.

No caso do Gerenciador de Auditoria ele pode ser ilustrado através de um diagrama abaixo da Figura 33 onde cada uma das fases do ciclo de vida é destacada juntamente com as ações que ocorrem nelas.

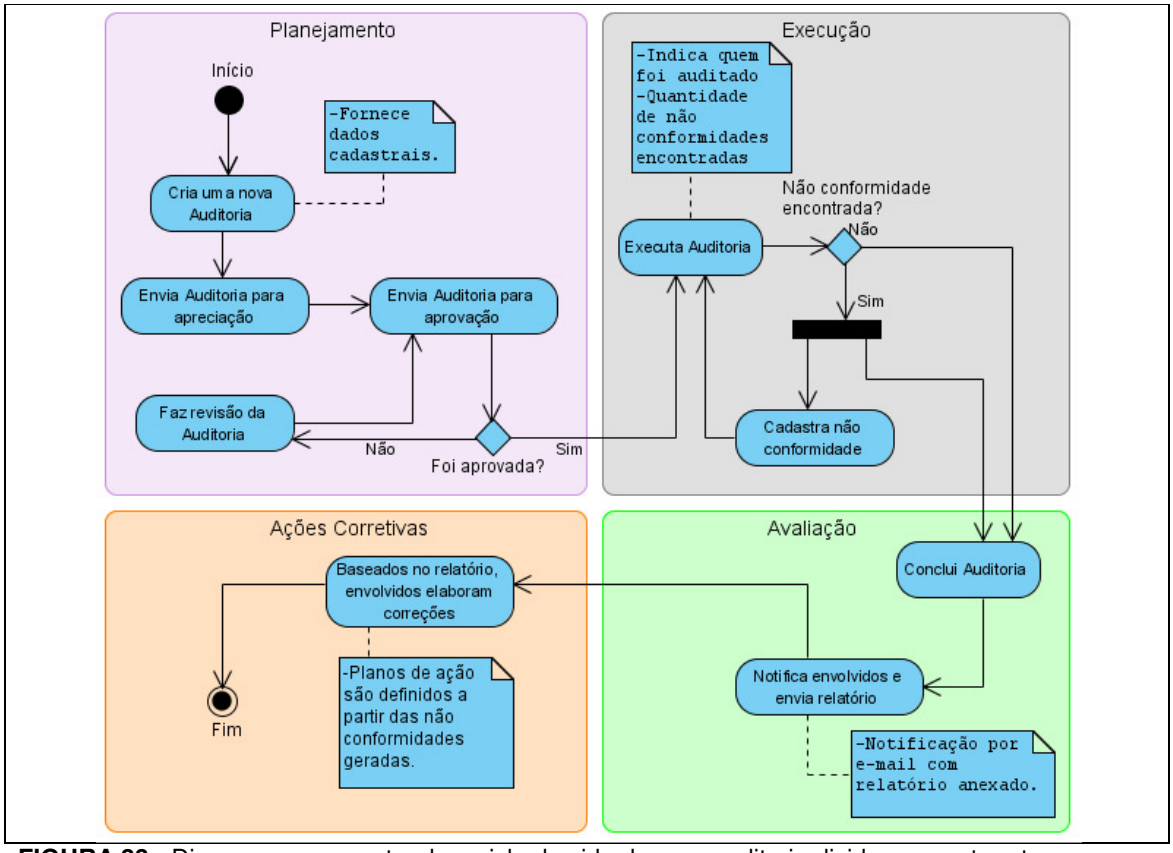

FIGURA 33 - Diagrama representando o ciclo de vida de uma auditoria divido em quatro etapas.

Na etapa de planejamento o elaborador da auditoria estabelece a estratégia geral de avaliação a ser executada nas áreas onde a auditoria será feita. No caso do Gerenciador de Auditorias é onde ele define que norma será utilizada na avaliação, qual será o seu tipo, quem é responsável por aprová-la, o seu prazo de conclusão, quem fará parte da equipe auditora, qual desses membros será o líder, quais setores serão avaliados, por quem e quando serão avaliados. Essas configurações são feitas a partir das abas da tela de Auditoria, ilustradas na Figura 13 e uma das

características implementadas no sistema que auxilia esse processo é a verificação de incompatibilidade de horários dos auditores de cada setor ou área que será avaliada. Essa necessidade de validação surgiu devido a dificuldade que existe na hora de configurar os horários de avaliação e os membros da equipe responsáveis por cada setor pois é muito comum haver conflitos de horários de auditores entre diferentes setores. A validação é feita sobre o setor e os integrantes da equipe responsáveis por ele antes que ele seja incluído na auditoria. Os métodos que fazem as verificações são invocados na ordem representada pelo diagrama da Figura 34.

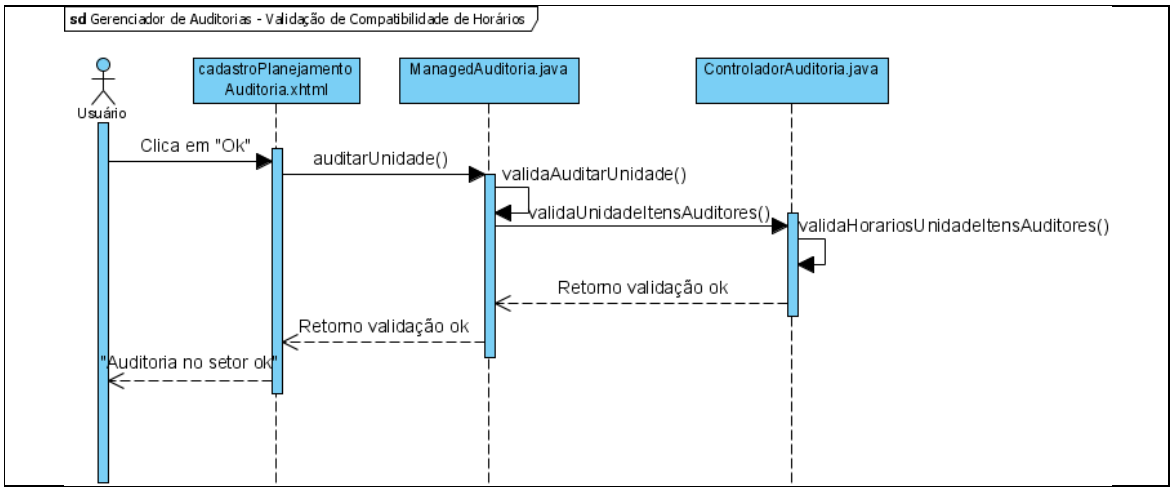

FIGURA 34 - Diagrama de sequência representando o processo de validação de um setor que está prestes a ser incluído na auditoria.

O processo de validação é iniciado a partir do momento que o usuário fornece os dados referentes a auditoria do setor em questão e os submete através do botão "Ok" que se encontra na janela, como é ilustrado na Figura 35.

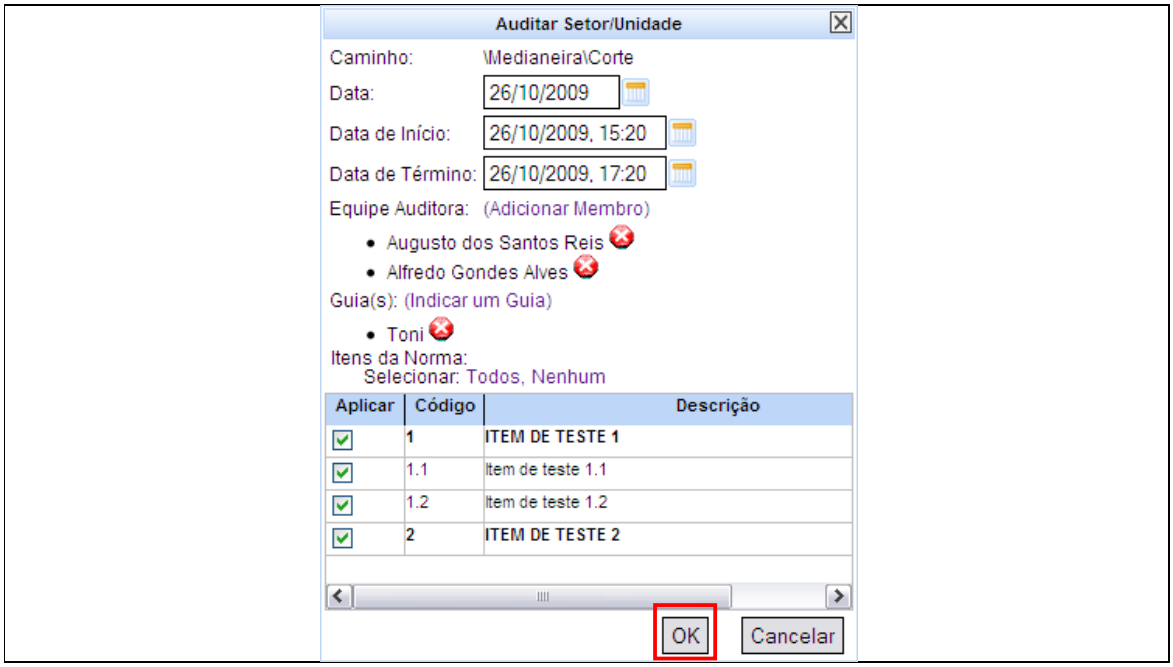

FIGURA 35 - Tela de cadastro dos dados do setor que será incluído na auditoria.

Essa interação do usuário com a página é responsável por chamar o método auditarUnidade do *backing bean* ManagedAuditoria, conforme mostra o Quadro 19.

*cadastroPlanejamentoAuditoria.xhtml

```
<a4j:commandButton value="#{msg.ok}"
    action="#{managedAuditoria.auditarUnidade}"
```

QUADRO 19 - Componente do Richfaces que faz a chamada ao método do ManagedAuditoria que inicia o processo de validação.

Como é o *backing bean* quem gerencia os objetos que são exibidos e manipulados pelo usuário através dos componentes das páginas, a instância do objeto que acabou de ser configurada pelo usuário está pronta para ser validada por ele. O método auditarUnidade depende do retorno lógico do método validaAuditarUnidade para dar continuidade ao processamento de sua lógica de negócios. Por isso, antes de mais nada, ele chama esse método, que além da validação de horários verifica se pelo menos um auditor foi designado para aquele setor e se pelo menos um item da norma foi escolhido para ser avaliado nele. Tanto a chamada ao método validaAuditarUnidade como parte do processamento que depende do seu retorno são ilustrados no Quadro 20

*ManagedAuditoria.java

```
/**
 * Faz a validação de horários entre as unidades que serão auditadas e caso haja algum
 * problema os erros são exibidos. Caso contrário a unidade é adicionada a lista de unidades
 * que serão auditadas.
 * @return String
 */
public String auditarUnidade() {
    podeFecharModal = false;
    try {
      if(validaAuditarUnidade()) {
    //Gera uma senha temporária aleatória se esse objeto ainda não foi salvo
    if(unidadeItensAuditores.getId() == null)
        unidadeItensAuditores.setIdTemp(new Date().getTime());
    //Se o item que foi criado/editado já existe ele é removido
    //e substituído por este que está sendo criado/editado nesse momento
     for (int i = 0; i < auditoria.getUnidadesAuditadas().size(); i++) {
         if(auditoria.getUnidadesAuditadas().get(i).getId() != null) {
```

QUADRO 20 - Método de validação do ManagedAuditoria chamado pela página.

O retorno do método validaAuditarUnidade é determinado pela validação que é feita pelo método validaUnidadeItensAuditores que se encontra na classe ControladorAuditoria. Esse método, além de fazer a validação dos horários dos auditores dos diferentes setores da auditoria chamando o método validaHorariosUnidadeItensAuditores, verifica se todos os atributos daquela classe anotados pelo Hibernate Validator estão em conformidade com as regras de validação que foram estabelecidas para aquele tipo de objeto. O Quadro 21 mostra como isso é feito através do código.

```
public RetornoValidacao
    validaUnidadeItensAuditores(UnidadeItensAuditores unidadeItensAuditores) {
    UnidadeItensAuditores uia = new UnidadeItensAuditores();
    uia = unidadeItensAuditores;
    List<String> listaErros = validaHorariosUnidadeItensAuditores(unidadeItensAuditores);

    ClassValidator<UnidadeItensAuditores> cv = new
            ClassValidator<UnidadeItensAuditores>(UnidadeItensAuditores.class,
            ResourceBundle.getBundle(ResourcesDigitalDocV2.HIBERNATE_MESSAGES, linguagem));
    InvalidValue[] values = cv.getInvalidValues(uia);
    InvalidValue[] erroHorarios = new InvalidValue[listaErros.size()];
```

QUADRO 21 - Método que faz validação dos horários e do objeto que representa um setor que será auditado.

O último método do processo de validação de uma auditoria sobre um setor é o validaHorariosUnidadeItensAuditores. Esse método varre a lista de membros da equipe auditora, verifica em cada setor que faz parte da auditoria em questão se algum dos membros da auditoria que está sendo validada faz parte de alguma outra auditoria de outro setor. Se algum membro da equipe auditora é responsável por outro setor dentro da mesma auditoria o método então verifica se as datas e os

horários coincidem e caso isso aconteça ele retorna uma mensagem. O Quadro 22 mostra o código que faz a validação descrita acima.

```
//Usuários da equipe dessa unidade
for (Usuario u : unidadeItensAuditores.getAuditores()) {
    //Unidades que já estão cadastradas para serem auditadas
    for (UnidadeItensAuditores uia : a.getUnidadesAuditadas()) {
        //Se alguma das unidades contem um dos usuários
        //verificar se os horarios batem
        if(uia.getAuditores().contains(u)) {
            if((unidadeItensAuditores.getHorarioInicio().
            after(uia.getHorarioInicio())
            && unidadeItensAuditores.getHorarioInicio().
            before(uia.getHorarioTermino()))
            || (unidadeItensAuditores.getHorarioTermino().
            after(uia.getHorarioInicio())
            && unidadeItensAuditores.getHorarioTermino().
            before(uia.getHorarioTermino()))) {
                listaDeMensagens.add("O membro " +
                u.getNomeCompleto() + " possui uma" +
                        auditoria agendada no setor "
                        + uia.getUnidade().getDescricao() +
                        " no dia " + uia.getData().get(Calendar.DAY_OF_MONTH) +
                        " das " + DataUtils.
                        parseToStringHoraMinuto(uia.
                        getHorarioInicio().getTime()) + " às " +
                        DataUtils.parseToStringHoraMinuto(uia.
                        getHorarioTermino().getTime()) + ".");
                return listaDeMensagens;
            }
        }
    }
}
return listaDeMensagens;
```

QUADRO 22 - Método que faz a validação dos horários dos membros da equipe que irão auditar o setor.

Esse método marca o final do processo de validação de um setor que está prestes a ser incluído na auditoria. Se tudo correr bem, ele é associado a auditoria e passa a fazer parte do plano de auditoria que está sendo criado. Os dados da auditoria sobre aquele setor específico que acabou de ser validade e incluído podem ser visualizados a partir da árvore de unidades/setores da auditoria em forma de *tooltip* como mostra a Figura 36.

FIGURA 36 - Visualização dos dados do setor que será auditado.

Se, depois de feitas todas as alterações e planejamento necessário sobre a auditoria, ela for aprovada, segue então para a próxima etapa.

Na etapa de execução as áreas onde a auditoria será feita são avaliadas de acordo com a norma especificada na sua configuração. Nesse momento, são especificadas as pessoas do setor que foram envolvidas na avaliação (auditados), quantas não conformidades foram encontradas a partir de cada item da norma utilizada, se existe alguma não conformidade que esteja fora do sistema mas associada a essa auditoria (não conformidade externa), observações que foram feitas durante a avaliação e oportunidades de melhoria dentro da área avaliada. Esses dados são manipulados através da janela de execução de auditoria, como mostra a Figura 37.

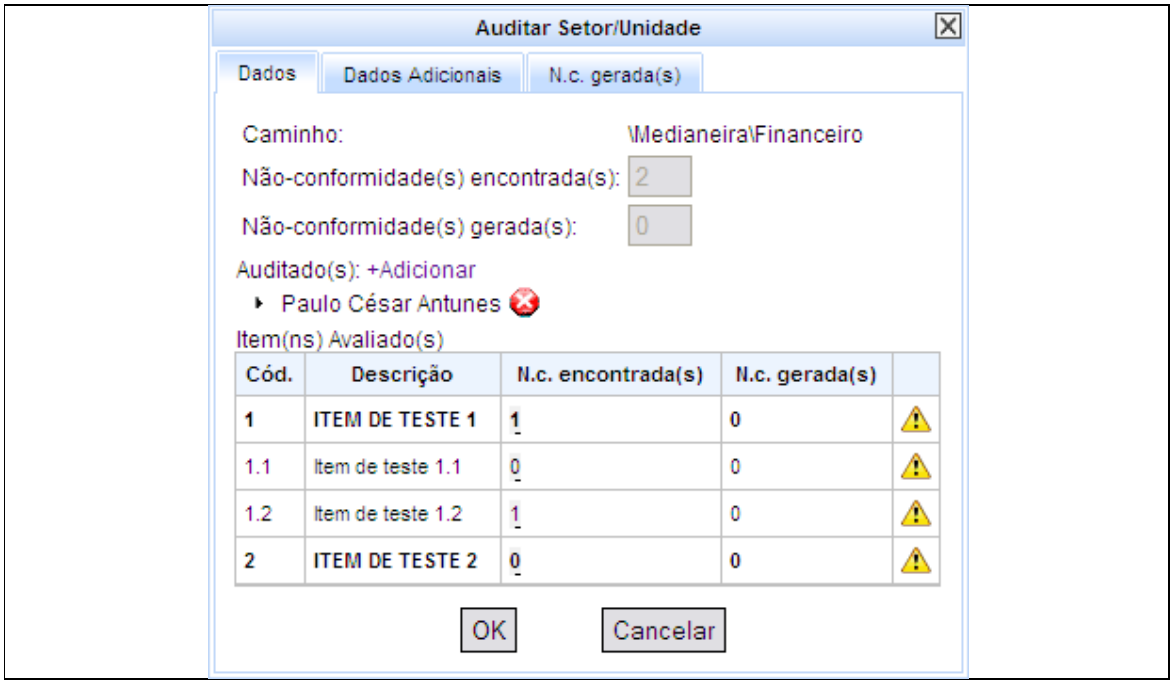

FIGURA 37 - Avaliação de um dos setores que fazem parte da auditoria.

É possível também cadastrar não conformidades a partir dos itens da norma onde foram encontradas as falhas. Nesse caso as não conformidades são associadas àquela auditoria e muitos de seus dados são preenchidos automaticamente a partir das informações que essa auditoria traz. Essa opção é acionada através do botão indicado na Figura 58. Se durante essa etapa algum dado referente ao planejamento for alterado, a auditoria é reenviada para aprovação e os dados da execução que foram registrados até então são apagados. Esse tipo de recurso só pode ser acionado pelo elaborador ou pelo administrador do sistema e visa dar maior flexibilidade a gerência do planejamento da auditoria ao longo do seu ciclo de vida já que imprevistos podem ocorrer com frequência quando lida-se com pessoas.

A etapa de avaliação é iniciada a partir do momento que a execução da auditoria é concluída, ou seja, a partir do momento que todas as áreas cadastradas na auditoria foram avaliadas de acordo com a norma. Nesse momento um relatório detalhado da auditoria e das informações que foram coletadas ao longo de sua execução é enviado para todos os envolvidos para que fiquem cientes dos resultados e comecem a elaborar o plano de ação para melhoria dos processos avaliados.

O fim do ciclo de vida da auditoria é marcado pela elaboração propriamente dita das medidas corretivas. Isso é feito a partir das não conformidades que foram geradas a partir da auditoria e estão cadastradas no sistema. As auditorias é concluída, mas as não conformidades dão continuidade ao seu próprio ciclo de vida e são tratadas de forma independente do Gerenciador de Auditorias para promover a melhoria do funcionamento da empresa.

Como pode ser observado, a aplicação faz uso extenso dos componentes JSF do Richfaces pois oferecem recursos mais avançados que possibilitam uma maior interatividade entre a aplicação e o usuário. Essa é uma grande vantagem de utilizar o JSF para o desenvolvimento de aplicações *web* pois existem várias bibliotecas disponíveis que oferecem uma enorme variedade de componentes. A Figura 38 mostra a interface com o usuário do sistema de gerenciamento de auditorias.

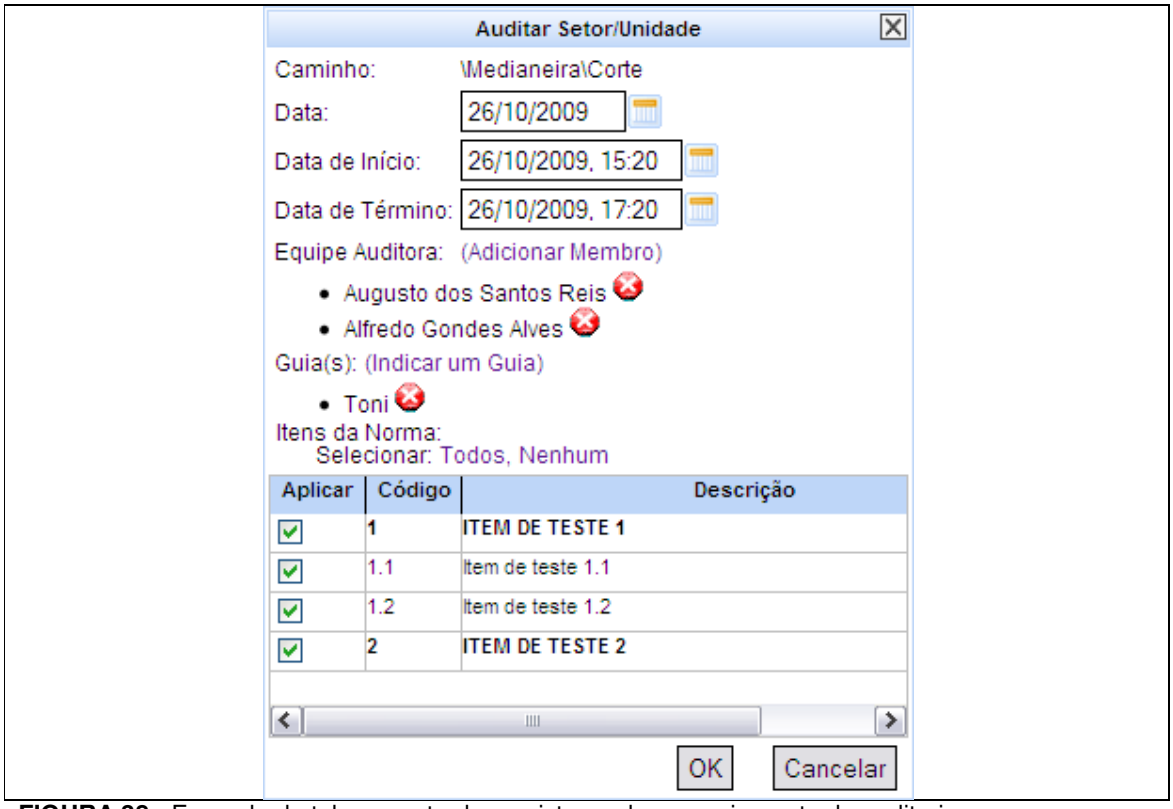

FIGURA 38 - Exemplo de tela encontrada no sistema de gerenciamento de auditorias.

Além de fazer uso de uma biblioteca de componentes a aplicação também emprega vários dos recursos oferecidos pelo JavaServer Faces. Um deles é o uso de Facelets para criar as páginas JSF e *layouts*. O uso dessa tecnologia não só

facilitou a criação das páginas em si, mas também promoveu uma grande reutilização de código já que uma composição pode ser utilizada em qualquer página JSF. O Quadro 23 mostra um trecho de código que representa uma aba do sistema que foi reutilizada várias vezes dentro de outras páginas JSF.

abaUnidadesAuditadas.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:a4j="http://richfaces.org/a4j"
      xmlns:rich="http://richfaces.org/rich">

<ui:composition>
    <rich:panel id="panelUnidadeSelecionada"
        styleClass="panelCaminho">
        <h:outputFormat id="unidadeSelecionada"
            value="#{msg.unidadeSelecionada}">
            <f:param value="#{managedUnidade.caminhoUnidadeSelecionada}" />
        </h:outputFormat>
    </rich:panel>

    <rich:panel style="height:200px; overflow:auto;"
        id="panelTreeUnidadesAuditadas">
        <ui:include src="/paginas/trees/treeUnidades.xhtml">
            <ui:param name="exibirMenuAdicionar" value="false" />
            <ui:param name="exibirMenuAuditoria" value="true" />
        </ui:include>
    </rich:panel>
</ui:composition>

</html>
```

QUADRO 23 - Exemplo de uma composição do Facelets que foi reaproveitada em vários pontos diferentes da aplicação.

Outro recurso do JSF muito utilizado na aplicação foi a integração facilitada entre a camada de apresentação e a camada de controladores, ou *Backing beans*. Este recurso facilitou a organização dos métodos dentro dos controladores pois cada um representava uma ação específica que poderia ser acionada pelo usuário a partir do seu navegador. Além disso, os retornos desses métodos ainda poderiam ser usados para redirecionar as páginas de acordo com as regras de navegação pré-estabelecidas. Dessa maneira ficou muito mais prático gerenciar a interação entre o usuário e os diversos recursos da aplicação. O Quadro 24 mostra um dos métodos do ManagedAuditoria, controlador responsável por boa parte das interações relacionadas a auditoria.

*ManagedAuditoria.java

```
/**
 * Faz a validação de horários entre as unidades que serão auditadas e caso haja algum
 * problema os erros são exibidos. Caso contrário a unidade é adicionada a lista de unidades
 * que serão auditadas.
 * @return String
 */
public String auditarUnidade() {
    podeFecharModal = false;
    try {
        if(validaAuditarUnidade()) {
            //Gera uma senha temporária aleatória se esse objeto ainda não foi salvo
            if(unidadeItensAuditores.getId() == null)
                unidadeItensAuditores.setIdTemp(new Date().getTime());
            //Se o item que foi criado/editado já existe ele é removido
            //e substituído por este que está sendo criado/editado nesse momento
            for (int i = 0; i < auditoria.getUnidadesAuditadas().size(); i++) {
                if(auditoria.getUnidadesAuditadas().get(i).getId() != null) {
                    if(auditoria.getUnidadesAuditadas().get(i).getId().equals(unidadeItensAuditores.getId())) {
                        auditoria.getUnidadesAuditadas().remove(i);
                        break;
                    }
                }
                if(auditoria.getUnidadesAuditadas().get(i).getIdTemp() != null) {
                    if(auditoria.getUnidadesAuditadas().get(i).getIdTemp().equals(unidadeItensAuditores.getIdTemp())) {
                        auditoria.getUnidadesAuditadas().remove(i);
                        break;
                    }
                }
            }
            auditoria.getUnidadesAuditadas().add(unidadeItensAuditores);
            ((ManagedUnidade) getManaged(Managed.UNIDADE)).adicionarUnidadeItensAuditores(unidadeItensAuditores);
            novaUnidadeItensAuditores();
            podeFecharModal = true;
        }
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}
```

QUADRO 24 - Método do ManagedAuditoria responsável por adicionar um elemento na lista de unidades/setores auditados baseado na escolha que o usuário fez na página.

Na aplicação também foram utilizados vários dos escopos de controladores oferecidos pelo JavaServer Faces, especialmente o session e request. O escopo session tornou o gerenciamento dos dados da sessão do usuário muito simples e eficaz. Através dessa funcionalidade era possível recuperar os objetos armazenados na sessão de um determinado usuário e reaproveitá-los em vários pontos diferentes da aplicação. Dessa maneira também se tornou mais fácil o gerenciamento de permissões dos usuários bem como a autenticação de cada um junto ao sistema. Esses recursos do JSF promoveram uma produtividade maior já que o *framework* é capaz de gerenciar boa parte dos detalhes mais técnicos envolvidos nessas tarefas relacionadas à sessão por conta própria sem a necessidade do desenvolvedor intervir.

5 CONSIDERAÇÕES FINAIS

5.1 CONCLUSÃO

Os benefícios oferecidos pelo JavaServer Faces para o desenvolvimento de aplicações *web* são muitos, especialmente quando usado em conjunto com uma IDE como foi comprovado através do estudo de caso, e ao longo dos seus anos de existência a evolução foi grande. Graças a esses fatores houve uma ótima aderência da comunidade à essa tecnologia e na última versão uma grande participação dela no processo de criação da especificação. É claro que, assim como qualquer *framework* no mercado, o JSF não é perfeito e não é a bala de prata para os problemas dos desenvolvedores, citando a ideia do artigo de Fred Brooks - *No Silver Bullet: Essence and Accidents of Software Engineering* - publicado em 1986. Como existem várias opções disponíveis hoje, o mais recomendado é escolher uma tecnologia que supra as necessidades de cada projeto em particular levando em consideração os seus requisitos e o custo-benefício oferecido por cada ferramenta.

As especificações Java são muito criticadas hoje em dia pelo fato de ainda serem controladas principalmente pelos grandes líderes da indústria. O foco desses grupos está, na maioria dos casos, em manter o JSF uma tecnologia regressivamente compatível e não tanto em torná-la uma solução inovadora. Embora a participação da comunidade no processo de especificação tenha aumentado na última versão do JSF, muitos acreditam que isso tenha acontecido tarde demais e que o JSF está destinado a se tornar uma tecnologia usada somente pelas corporações que a criaram. Outra crítica feita a esse processo fechado de criação é que dessa maneira a evolução do *framework* é muito lenta, ao ponto de que quando uma nova versão é finalmente lançada ela já é, em vários aspectos, obsoleta. O grande desafio para que o JavaServer Faces continue sendo um *framework* de sucesso e amplamente utilizado nos projetos de sistemas é acelerar o passo da sua evolução e abrir o processo de especificação para uma maior participação daqueles que o utilizam.

Porém, sem sombra de dúvida, os objetivos dos especialistas das especificações em tornar o uso das soluções corporativas, da qual o JavaServer

Faces faz parte, mais simples está sendo atingido. Talvez não no ritmo desejado, mas está. Isso pode ser visto claramente na evolução do JSF 1.x para JSF 2 através das várias melhorias que foram citadas neste trabalho. Cada vez mais as aplicações corporativas ficam mais fáceis de desenvolver sem perder o poder que já lhe são características e quando esse processo é auxiliado por uma IDE de qualidade como Eclipse, IntelliJ IDEA ou NetBeans os benefícios são maiores ainda. É importante que isto continue acontecendo para que Java continue sendo uma ótima e cada vez mais produtiva opção de linguagem de programação.

5.2 TRABALHOS FUTUROS

Novos trabalhos poderão ser realizados abordando o uso do *framework* JavaServer Faces em conjunto com outras linguagens de programação além do Java, tais como Groovy, Scala ou até mesmo JRuby. Por serem linguagens cujo código fonte é compilado para arquivos binários que são executadas pela *Java Virtual Machine* (JVM), é possível utilizá-las para desenvolver qualquer coisa que poderia ser desenvolvida com a linguagem Java sem problemas.

Quando abordado temas relacionados a *frameworks* para desenvolvimento de aplicações *web* Java, existem outras alternativas que podem ser estudadas, como é o caso do Grails, Struts, Vaadin, Apache Wicket e VRaptor que poderão ser estudados, pois visam agilizar o desenvolvimento de aplicações *web* utilizando diferentes abordagens do que o JavaServer Faces.

Como o JSF é uma especificação que faz parte da plataforma JEE, poderá também ser realizado um trabalho abordando o desenvolvimento de uma aplicação que abranja as outras especificações para agregar funcionalidades mais avançadas ao sistema. Uma ideia seria explorar a JSR 299 - *Context and Dependency Injection* - para gerenciar os contextos dos *beans* e dos serviços da aplicação e utilizá-la em conjunto com a JSR 224 - *XML-Based Web Services* - para expor os recursos da aplicação através do estilo arquitetural *Representational State Transfer* (REST), ressaltando as vantagens e diferenças quando comparado com a utilização do protocolo *Simple Object Access Protocol* (SOAP).

REFERÊNCIAS

BERGSTEN, Hans. JavaServer Faces. O'Reilly, 2004.

BESSA, Tarso; PONTE, Rafael. Os 10 Maus Hábitos dos Desenvolvedores JSF. **Mundo Java**, Campinas, v. 8, n. 38, p. 38-50, nov./dez. 2009.

BOND, M. et. al. **Aprenda J2EE em 21 dias**. São Paulo: Pearson Education do Brasil, 2005.

BURNS, Ed; KITAIN, Roger. JavaServer Faces **Specification**. Sun Microsystems, 2009.

BURNS, Ed; SCHALK, Chris. JavaServer Faces **2.0: The Complete Reference**. McGraw-Hill, 2010.

ECLIPSE. **About the Eclipse Foundation**, 2011. Disponível em <http://www.eclipse.org/org/>. Acesso em 14/05/2011.

FORD, Neal. **Art of Java *Web* Development**. Manning Publications, 2004.

GEARY, David. **JSF 2 Fu, Part 1: Streamline *Web* Application Development**, 2009. Disponível em <http://www.ibm.com/developerworks/java/library/j-jsf2fu1/index.html>. Acesso em 03/05/2011.

HALL, Marty. **Java Integrated Development Environments and Editors**, 2004. Disponível em <http://apl.jhu.edu/~hall/java/IDEs.html>. Acesso em 14/06/2011.

JAVA.NET. **JSF DataSheet**, 2009. Disponível em <www.*javaserverfaces.java.net/presentations/20090520-**jsf**2-datasheet.pdf*>. Acesso em 14/06/2011.

JAVAWORLD. **Asynchronous processing support in Servlet 3.0**, 2009. Disponível em <http://www.javaworld.com/javaworld/jw-02-2009/jw-02-servlet3.html>. Acesso em 01/02/2011.

JETBRAINS. **About The Company**, 2011. Disponível em <http://www.jetbrains.com/company/>. Acesso em 14/05/2011.

JCP. **Java Specification Requests - Detail JSR 127**, 2004. Disponível em <http://www.jcp.org/en/jsr/detail?id=127>. Acesso em 17/03/2011.

JCP. **Java Specification Requests - Detail JSR 252**, 2006. Disponível em <http://www.jcp.org/en/jsr/detail?id=252>. Acesso em 22/03/2011.

JCP. **Java Specification Requests - Detail JSR 314**, 2009. Disponível em <http://www.jcp.org/en/jsr/detail?id=314>. Acesso em 28/03/2011.

LOPES, Sérgio. As Novidades do JSF 2.0. **Mundo Java**, n. 40, p. 46-50, março/abril 2010.

MANN, Kito D. JavaServer Faces **in Action**. Manning, 2005.

MORDANI, Rajiv. **Java Servlet Specification**. Sun Microsystems, 2009.

NASCIMENTO, Rogério. Uma Aplicação Java EE Completa - Parte 2. **Java Magazine**, n. 45, p. 38, 2007.

NETBEANS. **An Introduction To NetBeans**, 2011. Disponível em <http://netbeans.org/about/index.html>. Acesso em 14/05/2011.

NOVOCODE. **Servlet Essentials - Chapter 1**, 2011. Disponível em <http://www.novocode.com/doc/servlet-essentials/chapter1.html>. Acesso em 27/01/2011.

NUNES, Vinny; MAGALHÃES, Eder. JSF 2.0 - Aprendendo JSF 2.0 com ScrumToys. **Java Magazine**, n. 78, p. 22-36, 2010.

ORACLE. **Java Servlet Technology**, 2010. Disponível em <http://www.oracle.com/technetwork/java/index-jsp-135475.html>. Acesso em 30/08/2010.

ORACLE. ***Web* Applications**, 2011. Disponível em <http://download.oracle.com/javaee/6/tutorial/doc/geysj.html>. Acesso em 17/01/2011.

READE, Chris. **Elements of Functional Programming**. Addison-Wesley Longman Publishing Co, 1989.

SERVLETS.COM. **The Problems With JSP**, 2011. Disponível em <http://www.servlets.com/soapbox/problems-jsp.html>. Acesso em 19/01/2011.

WESLEY, Addison. **Servlets and JavaServer Pages: the J2EE Techonology *Web* Tier**. Pearson Education Inc., 2004.