UNIVERSIDADE ESTADUAL DE MONTES CLAROS
Centro de Ciências Exatas e Tecnológicas
Curso de Bacharelado em Sistemas de Informação

Marcos Paulo Maia Rodrigues

DESENVOLVIMENTO DE SISTEMA PARA GERENCIAMENTO COMERCIAL INTEGRADO COM OS SERVIÇOS DE SUPORTE DA VBC AUTOMAÇÃO COMERCIAL LTDA

Montes Claros/MG
Novembro/2015

Marcos Paulo Maia Rodrigues

DESENVOLVIMENTO DE SISTEMA PARA GERENCIAMENTO COMERCIAL INTEGRADO COM OS SERVIÇOS DE SUPORTE DA VBC AUTOMAÇÃO COMERCIAL LTDA

Monografia apresentada ao Curso de Bacharelado em Sistemas de Informação da Universidade Estadual de Montes Claros, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Orientadora: **Prof.ª. CHRISTINE MARTINS DE MATOS, MESTRA**

Montes Claros - MG

Novembro/2015

Marcos Paulo Maia Rodrigues

DESENVOLVIMENTO DE SISTEMA PARA GERENCIAMENTO COMERCIAL INTEGRADO COM OS SERVIÇOS DE SUPORTE DA VBC AUTOMAÇÃO COMERCIAL LTDA

Monografia apresentada ao Curso de Bacharelado em Sistemas de Informação da Universidade Estadual de Montes Claros, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Orientadora: **Prof.ª. CHRISTINE MARTINS DE MATOS, MESTRA**

Montes Claros, 12 de novembro de 2015.

Membros:

_____

Orientadora**: PROFESSORA CHRISTINE MARTINS DE MATOS, MESTRA.**
Universidade Estadual de Montes Claros

_____

PROFESSORA MARILÉE PATTA, DOUTORA.
Universidade Estadual de Montes Claros

_____

PROFESSOR RAFAEL MORENO RIBEIRO DO NASCIMENTO, MESTRE.
Universidade Estadual de Montes Claros

Montes Claros - MG

Novembro/2015

AGRADECIMENTOS

Primeiramente agradeço a minha família pelo apoio durante todo o período acadêmico, em especial aos meus pais Belmiro Rodrigues de Amorim e Rosângela Maia de Amorim por terem me dado a oportunidade de ter uma educação de qualidade e assim chegar ao fim do ensino superior.

Agradeço a minha orientadora, Christine Martins de Matos, por todo o conhecimento, paciência e confiança dedicados a mim durante a elaboração do trabalho de conclusão de curso.

Por fim, agradeço aos meus amigos por toda a ajuda e apoio nos estudos em grupo, no desenvolvimento deste trabalho e nos momentos felizes, tristes e delicados que se sucederam nestes quatro anos de universidade.

RESUMO

O presente trabalho tem como objetivo descrever o desenvolvimento do *software* denominado de Sistema para gerenciamento comercial integrado com os serviços de suporte da VBC Automação Comercial LTDA, provendo o gerenciamento de informações referentes à negociação de produtos e/ou serviços da empresa, assim como sua posterior exportação para outros sistemas existentes na empresa. Esta característica é garantida por meio do armazenamento de dados referentes a contatos e que podem vir a se tornar clientes da empresa, fazendo com que não seja necessário o recadastramento dos mesmos dados em outro local do sistema. A gerência dos dados é realizada por meio dos relatórios emitidos pelo sistema, dando assim a oportunidade do administrador acompanhar de forma quantitativa o desempenho de vendedores, representantes comerciais e gerentes da empresa. A aplicação foi desenvolvida utilizando das linguagens Java EE, Javascript, HTML e CSS, tendo como banco de dados o MySQL. A aplicação deste sistema visa a informatização de um processo já existente dentro da empresa que utilizando das tecnologias citadas anteriormente, garante o maior aproveitamento das informações catalogadas durante as negociações tomadas pelos funcionários e assim, fazer com que se baseando nas mesmas sejam feitos o planejamento de vendas e definição de metas da empresa. Como resultado do desenvolvimento foi obtido uma aplicação que permite o controle dos dados e provê formas de gerência da informação por meio do armazenamento e emissão de relatórios acerca de resultados da empresa nos aspectos de negociação de seus produtos e/ou serviços.

Palavras-chave: Java EE, *Web*, Gerenciamento Comercial, Negociações.

ABSTRACT

This paper aims to describe the development of software called for integrated business management with the VBC Automação Comercial LTDA support services system, providing information management relating to the trading of goods and/or services of the company as well as its subsequent export to other systems within the company. This feature is ensured by means of data storage related to contacts and that may become clients of the company, making the re-registration of the same data is not required elsewhere in the system. The management of data is performed through reports issued by the system, thus giving the administrator the opportunity to quantitatively monitor the performance of vendors, sales representatives and managers. The application was developed using the languages Java EE, Javascript, HTML and CSS, with the database MySQL. The application of this system aims to computerize an existing process within the company using the aforementioned technologies, ensures better use the cataloged information during the negotiations taken by employees and thus make based on the same are made sales planning and defining business goals. As a result of the development it was obtained an application that allows control of data and provides forms of information management by storing and reporting on the company's results in the aspects of trading their products and/or services.

Keywords: Java EE, Web, Business Management, Negotiations.

LISTA DE FIGURAS

LISTA DE SIGLAS E ABREVIATURAS

AJAX	*Asynchronous Javascript and XML*
API	*Application Programming Interface*
CERN	*Organisation Européenne pour la Recherche Nucléaire*
CRM	*Customer Relationship Management*
CSS	*Cascading Style Sheets*
CT-E	Controle de Transporte Eletrônico
ECF	Emissor de cupom fiscal
EJB	*Enterprise Java Beans*
HTML	*Hyper Text Markup Language*
ISO	*International Organization for Standardization*
JAVA EE	Java *Enterprise Edition*
JAVA ME	Java *Micro Edition*
JAVA SE	Java *Stantard Edition*
JPA	*Java Persistence API*
JSF	*JavaServer Faces*
JSP	*JavaServer Pages*
LMC	Livro de Movimentação de Combustíve
MVC	*Model View Controller*
NF-E	Nota Fiscal Eletrônica
PAF-ECF	Programa de Aplicativo Fiscal – Emissor de Cupom Fiscal
PIS/COFINS	Programa de Integração Social/Contribuição para Financiamento da Seguridade Social
RA	Registro de Atendimento
SGBD	Sistema de Gerenciamento de Banco de Dados
SGML	*Standard Generalized Mark-up Language*
SPED	Sistema Público de Escrituração Digital
SQL	*Structured Query Language*
UML	*Unified Modeling Language*
URL	*Uniform Resource Locator*
USP	Universidade de São Paulo
W3C	*World Wide Web Consortium*

WWW	*World Wide Web*

SUMÁRIO

1 INTRODUÇÃO

Um desafio para a maioria das empresas da atualidade é a gestão de clientes. Segundo Carvalho (2012), clientes são pessoas ou organizações que adquirem algum produto ou serviço para consumo, estimulando assim a distribuição de capital na economia. Desse modo pode-se perceber que o alicerce principal para a sustentação da empresa são os clientes. Porém, antes de uma venda, a empresa pode passar por um longo processo de conversas e negociações a fim de entrar em um acordo e enfim, fechar ou não o negócio. Diversos fatores podem influenciar diretamente na quantidade de negócios fechados dentro da empresa e estes precisam ser trabalhados da melhor forma possível.

O mundo dos negócios é composto características que podem contribuir ou não com o seu sucesso. Não só o preço, mas o atendimento, a concorrência, a organização, o conhecimento amplo sobre o produto e/ou serviço comercializado, entre outros, podem influenciar diretamente em uma negociação de compra e venda, principalmente daquelas em andamento. Segundo Tavares (2000), a grande maioria dos negócios não concretizados tiveram como motivo a falta de planejamento ou erros nas atitudes tomadas.

Sperber (1982) define que planejamento tem como objetivo traçar rotas e estratégias para chegar a um objetivo específico. No ramo das vendas, empresas e/ou vendedores que fazem todo um planejamento inicial antes de iniciar uma negociação tem maiores chances de sucesso, já que estão preparados para possíveis adversidades que possam ocorrer durante a negociação. Este planejamento pode ser feito com base em negociações anteriores e em troca de experiências entre vendedores do mesmo ramo.

A falta de planejamento é um fator preocupante durante os processos empresariais. Empresas que não se preocupam com documentação e com a gestão da informação não conseguirão fazer um bom planejamento de vendas e consequentemente não terão as informações necessárias para subsidiar uma boa venda.

Para que se tenha acesso a informações de vendas passadas, a gestão de informação dentro da organização deve ser feita de maneira padronizada e constante, para que diante da grande variedade de possíveis clientes, o vendedor consiga se situar e traçar a melhor forma de conduzir a negociação. Segundo Tavares (2000) o vendedor não deve se preocupar somente em conseguir o menor preço, mas sim em construir uma parceria com o cliente, oferecendo o melhor de si durante a pré-venda para que, depois de concluída a negociação, consiga a fidelização do cliente e possíveis indicações.

Sendo assim, a utilização de um sistema específico para o controle e gerenciamento de clientes é essencial para permitir um fluxo de informações na empresa e produzir bons resultados nas tarefas exercidas internamente.

O essencial é que o processo de venda esteja disponível para diretores e gerentes para que estes possam estar sempre monitorando e verificando possíveis erros e também se o planejamento proposto pela empresa está sendo seguido. Essas informações podem ser repassadas entre estes e os vendedores de várias formas como reuniões, telefonemas, *e-mails* e etc. Uma forma centralizada de distribuição de informação dentro da organização pode facilitar a comunicação entre os membros da equipe e também possibilita a padronização da troca de informações internamente. Sendo assim, como se pode fazer informações de vendas e clientes esteja organizada e disponível para toda a equipe?

A ferramenta proposta para este trabalho, desenvolvida para uso na empresa VBC Automação Comercial LTDA, possibilita o controle das negociações que estão em andamento na mesma e permite um melhor planejamento de como serão feitas, já que os usuários terão acesso as negociações passadas e assim, possam estudá-las e buscar a melhor maneira de lidar com novos clientes. Com a visualização de relatórios que serão emitidos pelo sistema, será possível verificar o rendimento de cada vendedor, assim como fazer uma avaliação de como os clientes estão sendo atendidos, auxiliando também no pós-venda, que é um fator que influencia a fidelização do cliente. Para trabalhar a centralização das informações geradas pelo sistema, também propõe-se que estas sejam exportadas para outros sistemas existentes na empresa.

Sendo assim, este trabalho tem-se como objetivo geral desenvolver uma ferramenta *web* que permita acompanhar e gerenciar as negociações de vendas de serviços da empresa VBC Automação Comercial Limitada, bem como sua posterior integração com outros sistemas existentes na empresa. Seus objetivos específicos são: relacionar dados referentes a diretores, gerentes, vendedores, clientes e negociações, possibilitando assim a uma melhor gestão de informação; analisar, de acordo com históricos de venda, como as negociações são executadas pelos usuários; acompanhar, com base em relatórios e dados cadastrados pelos usuários, as negociações em andamento; exportar os dados de clientes para sistemas de gerenciamento comercial, de clientes e/ou de suporte existente na empresa.

Durante o desenvolvimento do sistema, foram utilizados os conceitos de pesquisa metodológica descritos por Lakatos e Marconi (2003) e informações baseadas no dia a dia do setor comercial da VBC Automação Comercial. Como técnicas de pesquisa foram utilizadas observação, entrevistas e testes.

A entrevista tem como objetivo obter informações sobre um determinado assunto. Para a realização do trabalho, foi utilizada a pesquisa não estruturada, que segue como uma conversa e permite ao entrevistador abordar diferentes temas. Foram entrevistados os gerentes, representantes comerciais e o diretor do setor, buscando maiores detalhes sobre os processos internos da área. Desse modo, foi possível entender e estruturar como funciona o setor comercial da VBC, para que então fosse possível traçar os primeiros aspectos que estão presentes no sistema desenvolvido.

Com base na observação, que Lakatos e Marconi (2003) definem como análise e obtenção de informações a partir dos aspectos do que estão acontecendo no ambiente analisado, foi possível obter dados sobre o andamento da negociação do início ao fim, bem como a forma de trânsito da informação dentro do setor. Assim, foi possível observar que dados referentes as negociações eram salvos utilizando-se de planilhas eletrônicas, tornando o trabalho pouco eficiente e inseguro, já que a exclusão dos arquivos comprometeria toda a informação armazenada. Também por conta desta forma de controle, não era possível gerar relatórios sobre as informações obtidas, impedindo assim uma análise mais profunda acerca dos resultados.

Com base nas informações obtidas por meio das observações e entrevistas, foi possível realizar a análise dos requisitos necessários para o desenvolvimento do sistema, elaborando assim o desenho do banco de dados, do diagrama de classes e das telas do sistema, que são mostradas nas seções 3 e 4 do documento. Foram realizados testes funcionais durante o desenvolvimento para assegurar com que o comportamento do sistema estivesse de acordo com o projetado.

Para apresentação dos resultados obtidos com o desenvolvimento do sistema, o trabalho realizado está organizado em 5 capítulos, sendo o primeiro com a introdução; o segundo com a fundamentação teórica necessária para o desenvolvimento do trabalho; o terceiro capítulo contem a descrição dos pontos mais relevantes do desenvolvimento do trabalho; o capítulo quarto contendo as considerações finais, seguido das referências.

2 FUNDAMENTAÇÃO TEÓRICA

Para o desenvolvimento da aplicação *web* são necessários conhecimentos na área de tecnologia e desenvolvimento de sistemas, além de conhecimentos sobre a empresa VBC Automação Comercial Ltda, que receberá o sistema.

2.1 VBC Automação Comercial Ltda.

A VBC Automação Ltda., empresa que tem a matriz situada na cidade de Montes Claros, Minas Gerais, atua no ramo de desenvolvimento e comercialização de sistemas, serviços de assistência técnica e consultoria para postos de combustíveis desde 1995. Fornece soluções para emissor de cupom fiscal (ECF), nota fiscal eletrônica (NF-e), controle de transporte eletrônico (CT-e), gerador de arquivos para o Sistema Público de Escrituração Digital (SPED) Fiscal e Programa de Integração Social/Contribuição para Financiamento da Seguridade Social (PIS/COFINS), controle de frotas e gerenciamento empresarial. Atualmente, a empresa possui mais de 400 clientes e com a crescente demanda por sistemas de informação em postos de gasolinas, e este número tende a aumentar (VBC AUTOMAÇÃO, 2014).

Para emissão de cupons, a empresa desenvolveu o sistema Saga ECF, que é um sistema utilizado em caixas, que faz comunicação com impressoras fiscais e concentradores de combustíveis. Com ele, os dados recebidos pelo concentrador de combustíveis, provenientes das bombas presentes nos postos de gasolina, referentes aos abastecimentos, são processados e disponibilizados para emissão de cupons fiscais. Ele também faz a venda de qualquer outro tipo de produto, pois também funciona como emissor de cupom fiscal para lojas de conveniência, de peças e restaurantes.

O sistema é um Programa de Aplicativo Fiscal – Emissor de Cupom Fiscal (PAF-ECF), que segue as regras definidas pelo Conselho Nacional de Política Fazendária, do Ministério da Fazenda, que disponibiliza no Manual do Desenvolvedor de Programa Aplicativo Fiscal Emissor de Cupom Fiscal, todo um conjunto de normas e técnicas que devem ser aplicados em todos os sistemas que envolvem a venda de produtos, emissão de cupons e outros arquivos fiscais.

Utilizando das mesmas tecnologias do Saga ECF, um outro sistema da empresa é o Sagaline Frotas, que faz o controle de abastecimentos para empresas de transporte que possuem suas próprias bombas de combustível, não sendo assim necessária a emissão de

cupom fiscal que adquirem os combustíveis diretamente da distribuidora. Com este sistema, as próprias empresas podem fazer o controle do combustível consumido por cada motorista, liberando as bombas somente mediante o uso de um *chip* de identificação.

Outro sistema, denominado de Sagaline Gerencial é responsável por integrar e gerenciar todos os dados provenientes do Saga ECF, Sagaline Frotas, emissores de NF-e, CT-e e arquivos do SPED Fiscal e PIS/COFINS. Ele possui rotinas de planos de contas, controles de contas a pagar e a receber, gerenciamento de clientes, vendas, controles de estoque e do Livro de Movimentação de Combustível (LMC).

Internamente, a empresa faz o uso do Registro de Atendimento (RA) VBC, um sistema desenvolvido pela VBC para registro e gerenciamento de atendimentos. O sistema tem como objetivo auxiliar no funcionamento do setor de atendimento ao cliente, conforme são recebidas ligações de clientes e afins, são abertos chamados no sistema, que a partir disso, passa a receber informações dos mesmos.

2.2 Sistema de gerenciamento de relacionamento com o cliente

Segundo Laudon e Laudon (2007), os sistemas de gerenciamento de relacionamento com o cliente ou c*ustomer relationship management* (CRM) em inglês tem como principal função a integração e armazenamento de dados sobre o relacionamento entre empresa e clientes. Informações como telefones, endereços, contatos do cliente são armazenadas e analisadas pelo sistema, podendo gerar informações como valor do cliente para a empresa, quantidade de vendas, entre outros, podendo assim, com base nos dados coletados reavaliar novos produtos e modos de venda, por exemplo.

As informações capturadas pelo CRM, após analisadas e consolidadas, são distribuídas entre os pontos de contato da empresa. Cada ponto de contato é um canal do qual a empresa entra em contato com o cliente, podendo ser telefone, *e-mail*, serviços de suporte, dispositivos, entre outros. A aplicação desenvolvida usa de alguns aspectos do CRM como a captura de dados dos clientes e o fornecimento de informações sobre os produtos adquiridos, provendo assim um melhor relacionamento com a empresa, desde o fechamento do contrato, e posteriormente ter suas informações disponíveis para distribuição em outros setores da empresa.

2.3 Orientação a objetos

Tendo-se firmado como um dos métodos mais eficientes para desenvolvimento de *software* a partir da década de 90, a orientação a objetos segundo Rumbaugh (2006), pode ser tratada como um método de organizar o *software* com base em uma coleção de objetos distintos, que ditam e conectam a estrutura dos dados e o comportamento de todo o sistema.

Dall'Oglio (2007) define a orientação a objetos como um paradigma que define a construção de um sistema. Sistemas que antes eram construídos com base em um conjunto de procedimentos e variáveis agrupados por algum contexto em comum, passaram a ter uma ótica próxima ao mundo real. Fazendo uma analogia entre os objetos do mundo real e os objetos lógicos, é possível fazer uma estrutura programática para montar a arquitetura inicial da aplicação que está sendo desenvolvida.

A orientação a objetos utiliza, em sua arquitetura, as classes. Uma classe representa um objeto e contêm variáveis e métodos que as manipulam. Então, ao se declarar a classe conta, por exemplo, deve-se imaginar quais características implicam para a existência da mesma no mundo real. Portanto, uma classe com suas variáveis devidamente preenchidas, torna-se um objeto. A Figura 1 mostra a declaração de uma classe em um ambiente de desenvolvimento.

Figura 1 - Declaração da classe conta

```
class Conta {
    int numero;
    String dono;
    double saldo;
    double limite;

    // ..
}
```

Fonte: CAELUM, 2013a, p.45

Como acontece no mundo real, objetos podem relacionar-se entre si para formar um objeto maior. Por exemplo, o objeto bicicleta é composto pelos objetos roda, banco, guidom, parafuso, entre outros. Portanto, para a existência do objeto bicicleta, todos os outros devem ser ligados a ele.

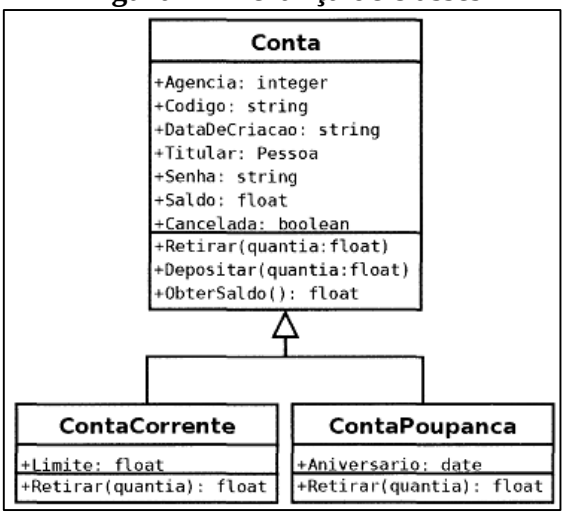
Outras características podem estar presentes nas classes, o que pode aumentar a sua gama de utilização. São definidas como herança, polimorfismo e abstração, apresentadas no subitem 2.3.1.

2.3.1 Herança, Polimorfismo e Abstração.

Dall'Oglio (2007) explica que maneira mais simples de entender o conceito de herança entre classes é imaginar uma árvore genealógica. Tem-se a presença de indivíduos no topo e na base da árvore onde, os indivíduos das camadas inferiores herdam características dos indivíduos das camadas mais altas.

Assim como define Rumbaugh (2006 p 39), "a superclasse mantém atributos, operações e associações comuns; as subclasses acrescentam atributos, operações e associações específicos ". Dessa forma, trabalhando com a herança entre classes, o desenvolvedor ao manipular uma classe filha (subclasse), pode acessar métodos e variáveis da classe pai (superclasse), podendo assim prover a centralização e diminuição do código. A Figura 2 representa graficamente o conceito de classes interligadas por herança.

Figura 2 - Herança de classes



Fonte: DALL'OGLIO, 2007, p.99

Analisando-se a Figura 2 percebe-se a presença da classe pai "conta" e as classes filhas "ContaCorrente" e "ContaPoupanca". Assim implementadas, as classes filhas serão compostas por suas variáveis e métodos locais e também pelos presentes na classe pai.

O polimorfismo tem como característica permitir que a subclasse tenha os métodos com os mesmos nomes da superclasse. Essa característica implica que, o método da subclasse pode conter códigos exclusivos de sua classe para assim, após alguma operação específica da mesma, tenha-se continuidade no método padrão da superclasse (RUMBAUGH, 2006).

Como se pode observar na Figura 2, as 3 classes referenciadas possuem o método “Retirar(quantia)”, porém quando chamado pela classe “ContaCorrente”, ela sobrescreve a chamada da superclasse, podendo executar ações diferentes das presentes na superclasse e, após isso, pode executar ações da mesma.

Porém, deve-se ficar atento, pois como ressalta Rumbaugh (2006), nunca se deve sobrescrever métodos de uma superclasse de forma com que ela seja incoerente com as características originais que foram herdadas, já que isso pode levar a confusão sobre os métodos e atributos presentes na classe e a suposições sobre o funcionamento da mesma.

A abstração de classe faz com que seja possível proteger classes do acesso direto. Na Figura 3 são apresentados as 3 classes, “Conta”, “ContaPoupanca” e “ContaCorrente”. Porém não faz sentido o desenvolvedor instanciar a superclasse “Conta”, já que será útil para o sistema somente as classes filhas. Desse modo, deve-se abstrair a classe conta, implementando a mesma com a notação *abstract*. Com essa operação, não será possível instanciar a classe “Conta”, ficando aceitável fazer somente com as suas classes filhas. Ao contrário da notação *abstract*, a notação *final* faz com que a classe não possa ser utilizada como superclasse. Desse modo, com essa notação, a classe “ContaCorrente” por exemplo, nunca poderá ser instanciada como superclasse.

2.4 Modelagem de sistemas com UML

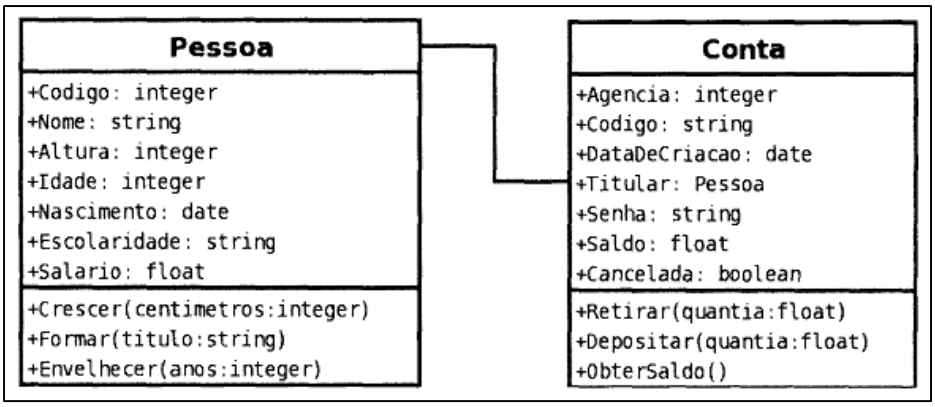
Segundo Pressman (2010) a *Unified Modeling Language* (UML), ou linguagem de modelagem unificada em português, tem como função fornecer um modelo padronizado para a representação completa de um projeto de software.

A UML foi criada nos anos 90, por Grady Booch (engenheiro de *software*), Jim Rumbaugh (Cientista da computação) e Ivar Jacobson (Cientista da computação), utilizando de um grupo de notações de modelagem concorrentes e nos dias de hoje se encontra na versão 2.0, sendo também um padrão da *International Organization for Standardization* (ISO). A UML 2.0 fornece 13 tipos de diagramas para modelagem de *software* e para o desenvolvimento deste sistema, utilizou-se o digrama de classes.

Muller (2002) define o diagrama de classes como um diagrama estrutural ou estático com o qual o desenvolvedor ou projetista pode modelar a estrutura do sistema de classes que será usado como base para a o desenvolvimento da arquitetura do sistema desenvolvido. Ele é desenhado com o uso de caixas divididas em partes horizontais. A parte superior de cada caixa apresenta o nome da classe, a parte do meio os atributos, e a parte

inferior apresenta as operações e comportamento das classes. A Figura 3 demonstra como as classes podem ser representadas graficamente no diagrama de classes.

Figura 3 - exemplo de diagrama de classes



Fonte: DALL'OGLIO, 2007, p.94

A representação do sistema por meio de diagramas demonstra de forma mais clara para os envolvidos no projeto, como serão construídas as funções e logísticas de todo o sistema. Portanto os diagramas, que são desenhados após a análise de requisitos do sistema, são utilizados como um guia para que o desenvolvedor desenvolva a aplicação de acordo com o projeto, podendo assim construir um sistema com integridade e bases bem definidas, entregando assim um produto completo e refinado.

2.5 Banco de dados

Elmasri e Navathe (2005, p. 4) definem dados como "fatos que podem ser gravados e que possuem um significado implícito". Podem-se citar como fatos os nomes, números de telefone, endereços de *e-mail*, entre outros. Portanto um conjunto de dados pode ser definido como um banco de dados.

Quando se trabalha com empresas de grande ou pequeno porte, que contém algum tipo de informatização, é necessário o uso de técnicas e *softwares* para o gerenciamento, para que assim a empresa possa utilizar dados para armazenamento de informações e gerência da empresa. Esse fato, para organizações de grande porte é indispensável, pois o volume de dados gerados diariamente não os tornam de fácil análise e manipulação para o ser humano e, desse modo, foram criados os sistemas de banco de dados.

Elmasri e Navathe (2005, p. 4) definem um banco de dados como uma coleção de dados relacionados, e ainda que “O banco de dados é uma coleção lógica e coerente de dados com algum significado inerente. Uma organização de dados ao acaso (randômica) não pode ser corretamente interpretada como um banco de dados”. Sendo assim, Silberschatz, Korth e Sudarsham (1999, p. 1) definem um sistema de banco de dados como “um conjunto de dados associados a um conjunto de programas para acesso a esses dados”. Ou seja, este conjunto de dados, que geralmente é chamado de banco de dados ou base de dados, tem como função armazenar e gerenciar informações de programas instalados no computador. Ele é constituído de uma ou mais tabelas que são desenvolvidas com base nos requisitos do sistema que está sendo desenvolvido. Para isso deve-se primeiro entender primeiramente o conceito de um sistema gerenciador de banco de dados (SGBD).

O SGBD, segundo Elmasri e Navathe (2005) é uma coleção de programas que permite a criação e manipulação de bancos de dados. Sendo assim, ele provê a execução dos processos de definição, construção, manipulação e compartilhamento de bancos de dados entre usuários e sistemas, enquanto características de um SGBD:

- definição: a definição de um banco de dados implica na especificação de todos os tipos de dados que serão manipulados pelo SGBD e a definição de suas estruturas e restrições;
- construção: a construção de um banco de dados refere-se ao processo de armazenar um banco de dados em alguma mídia controlada pelo SGBD. Esta mídia pode estar presente no computador do usuário, ou em um dispositivo remoto, que pode ser acessado por meio da internet ou outro tipo de conexão de rede;
- manipulação: a manipulação de banco de dados implica nos processos de pesquisa e atualização dos dados presentes no mesmo. A atualização pode incluir, excluir e editar dados presentes no banco, sempre respeitando as informações inseridas pelo usuário e,
- compartilhamento: o compartilhamento permite a usuários e programas terem acesso aos dados presentes no banco de forma concorrente, ou seja, sem que um impeça o outro.

Também é de responsabilidade do SGBD prover a proteção e a manutenção do banco de dados. A proteção implica na proteção do sistema contra falhas ou mau funcionamento do *hardware* (parte física dos computadores) ou *software* (parte lógica dos

computadores) bem como contra acessos não autorizados e maliciosos. A manutenção implica em manter os dados do banco íntegros, já que os mesmos geralmente tem um ciclo de vida de muitos anos, portanto o SGDB deve fazer com que os dados estejam seguros perante as evoluções e possíveis alterações no banco de dados.

Para a manipulação dos dados, os sistemas utilizam de uma linguagem de consulta chamada *Structured Query Language* (SQL) ou Linguagem de Consulta Estruturada, em português. Conforme afirmam Silberschatz, Korth e Sudarsham (1999), além de fornecer ferramentas de consulta, a SQL possui recursos para gerenciamento completo da base de dados, como alterações na estrutura ou a criação de funções que podem ser ativadas automaticamente para alterar dados do banco. Também podem ser utilizadas outras tecnologias para manipulação dos dados e assim facilitar o desenvolvimento da aplicação.

Por se tratar do desenvolvimento de um sistema com arquitetura totalmente orientada a objetos, foi utilizado um *framework* para gerenciamento de banco de dados entre a aplicação e o SGDB. Um *framework* tem como função criar uma ponte entre as diversas tecnologias usadas durante o desenvolvimento, podendo assim centralizar os códigos utilizados durante o desenvolvimento. Diante das diversas opções disponíveis como Spring Data, Eclipse Link, Castor entre outros, foi utilizado o Hibernate, na variação Hibernate ORM.

Segundo Hibernate (2014), o Hibernate ORM é uma tecnologia que fornece aos desenvolvedores uma maior facilidade no desenvolvimento de aplicações que necessitam de manipular dados. Ele atua em conjunto com o SGDB e com as tecnologias providas pelo Java. O Hibernate ORM trabalha utilizando os conceitos de orientação a objeto, e desse modo permite com que o controle do banco de dados seja feito totalmente através de objetos definidos pelo desenvolvedor. Assim, ao definir e preencher um objeto, o próprio *framework* gera dinamicamente o código SQL e faz a inserção no SGBD, garantindo maior agilidade e confiabilidade no desenvolvimento da aplicação.

2.6 Java

Oracle (2012) define a Java como uma tecnologia que atua como linguagem de programação e uma plataforma. A linguagem de programação é uma linguagem orientada a objetos de alto nível, com métodos e sintaxes próprias. A plataforma é um ambiente específico que permite que a linguagem Java seja executada. Existem algumas plataformas Java e cada uma é específica para um ambiente de desenvolvimento:

- *Java Standard Edition* (Java SE): esta plataforma Java contém todos os objetos básicos da linguagem atuando como o núcleo das funcionalidades presentes na mesma e permite o acesso a banco de dados, interface gráfica, rede, entre outros. Geralmente, é utilizada para o desenvolvimento de aplicações *desktop*;
- *Java Enterprise Edition* (Java EE): está contida acima da plataforma Java SE e provê um ambiente para desenvolvimento de aplicações que permitem a execução em larga escala, escalabilidade, confiabilidade e maior segurança para aplicações executadas em rede;
- *Java Micro Edition* (Java ME): esta plataforma Java contém uma versão "resumida" do Java SE, permitindo assim a sua execução em dispositivos mais simples como telefones móveis de baixa capacidade de processamento. Ela também possui bibliotecas próprias para uso nestes dispositivos e geralmente atua em conjunto com serviços Java EE;
- JavaFX: esta plataforma prove um ambiente de desenvolvimento de aplicações *web* com uma interface gráfica que facilita o desenvolvimento e *design* das mesmas. Ela possui bibliotecas que permitem um maior uso de recursos do *hardware* do cliente, permitindo assim o desenvolvimento de interfaces com mais detalhes gráficos com a presença de animações e outros objetos que exigem maior processamento. Trabalha em conjunto com o Java EE.

Com o intuito de desenvolver uma aplicação *web* de alta performance, foi utilizada neste sistema a plataforma Java EE que também é definida por Caelum (2013b) como uma série de especificações com riqueza de detalhes que permite ao desenvolvedor, criar diversas aplicações mudando a forma como as páginas *web* eram construídas.

Faria (2015) ressalta que o Java EE também possui funcionalidades que permitem a criação de aplicações Java distribuídas, suportando assim escalabilidade, segurança e integridade a outros sistemas, requisitos hoje cada vez mais solicitados para o uso em aplicações de grandes corporações. As funcionalidades mais conhecidas desta tecnologia são:

- *Servlets*: são componentes Java executados no servidor, permitindo assim a criação conteúdo totalmente dinâmico na página *web* desenvolvida;
- *JavaServer Pages* (JSP): uma extensão dos *servlets*, permitindo a adição de códigos Java e conteúdo diretamente na página *web*. As informações

enviadas pelo *Servlet* podem ser tratadas e modificadas diretamente no JSP;

- *JavaServer Faces* (JSF): é um *framework web* baseado em Java que garante maior facilidade na criação e *design* de interfaces *web*, buscando trazer a mesma facilidade do desenvolvimento de interfaces em aplicações *desktop*;
- *Java Persistence* API (JPA): é um *Application Programming Interface* (API) ou Interface de Programação de Aplicações em português, padrão do Java para persistência (armazenamento) de dados, utilizando do conceito de mapeamento objeto-relacional. Essa tecnologia permite o armazenamento sem que usuário necessite de escrever códigos em SQL, facilitando o desenvolvimento e aumentando a produtividade;
- *Enterprise Java Beans* (EJB): o EJB trabalha em conjunto do JPA e fornece uma conexão abstraída e facilitada entre o banco de dados e o servidor da aplicação, garantindo produtividade, estabilidade e segurança durante as transações do mesmo.

No desenvolvimento de aplicações *web* utilizando o Java EE, usa-se o *Servlet* para controlar e direcionar as requisições feitas pelo usuário do sistema *web*. Ele é um dos responsáveis pelo funcionamento correto de cada tela e função do sistema. Para facilitar e agilizar o desenvolvimento do sistema, foi utilizado o componente VRaptor.

O VRaptor é uma biblioteca *Servlet* open source, que teve seu desenvolvimento iniciado na Universidade de São Paulo (USP) e possui diversos colaboradores no Brasil e outros países. Ele foi desenvolvido com base no *Servlet* padrão oferecido pela biblioteca Java EE, porém já possui métodos pré-definidos a fim de proporcionar uma fácil e rápida implementação.

Em conjunto com o Java, para permitir a exportação de dados do sistema desenvolvido, foram utilizadas as tecnologias do JasperReports, que segundo Jaspersoft (2015) é uma biblioteca Java de código aberto que oferece uma interligação com o mecanismo de relatórios utilizado pelo JasperReports, fornecendo assim uma forma descomplicada de se exportar dados da aplicação desenvolvida para uma vasta gama de formatos como PDF, XLS, entre outros. A Jaspersoft também fornece a aplicação iReport, que permite o desenho dos *layouts* que serão utilizados na confecção dos relatórios pela aplicação.

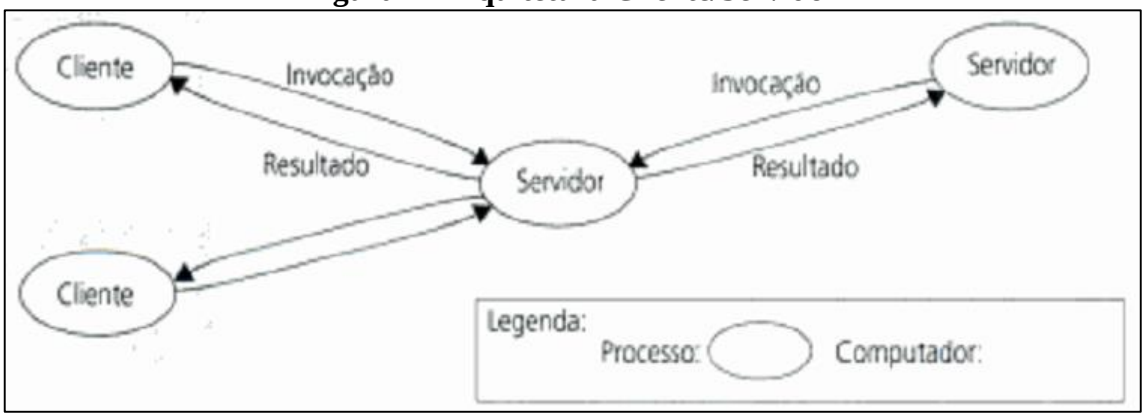
2.7 Desenvolvimento *Web*

É indispensável que empresas façam o uso de aplicações para exercer seus serviços. Isso acontece, pois utilizando das informações providas por estas ferramentas, elas puderam se tornar mais competitivas e executar suas tarefas com mais rapidez e segurança. Desse modo, qualquer empresa que deseja crescer no mercado atualmente, deve estar informatizada e de acordo com as últimas tendências tecnológicas. Uma das novas tendências existentes é o uso de aplicações *web*. As aplicações *web* são desenvolvidas com o objetivo de tornar os trabalhos acessíveis de qualquer localidade ou dispositivo com conexão com a internet, geralmente representados por computadores, *tablets* e *smartphones*.

Para o desenvolvimento foram utilizados os conceitos da arquitetura cliente-servidor. A arquitetura cliente-servidor segundo Tanenbaum (2008) é dividido em duas partes. O servidor é um processo que executa serviços específicos como sistemas de banco de dados, por exemplo. Já o cliente é o processo que requisita este serviço, por meio de uma conexão com o servidor. Ao receber alguma requisição, o servidor faz uma análise, processa e retorna a informação requerida pelo cliente.

Este processo pode ser observado nos caixas eletrônicos em bancos. Os caixas funcionam como clientes e a cada utilização, um saque, por exemplo, ele faz uma requisição ao servidor do banco no qual é realizada a operação desejada e retornada a resposta para o caixa, que de acordo com a mesma pode liberar ou não a quantia desejada pelo usuário. Um servidor também pode fazer requisições ao cliente, caso necessite de informações extras ou então caso necessite consultar outro servidor. A Figura 4 demonstra de uma maneira simples, o funcionamento desse tipo de arquitetura.

Figura 4- Arquitetura Cliente/Servidor



Fonte: COULORIS, 2007, p.43.

2.8 Ferramentas de desenvolvimento *Web*

Neste item são apresentadas as tecnologias que foram utilizadas para permitir o desenvolvimento da aplicação no ambiente *web*.

2.8.1 Design de interfaces web

Ragget *et al* (1997) citam que a ideia da internet surgiu perante a necessidade de interligar os diversos centros de pesquisa espalhados pelo mundo que já possuíam redes computacionais internas, mas nada em nível global, de modo que, além de se poder acessar pesquisas de outros centros, o usuário pudesse acessar outros arquivos relacionados a estes por meio de alguma ligação ou *link* entre os mesmos.

Standard Generalized Markup Language (SGML), ou Linguagem de marcação generalizada padrão em português que Ragget, *et al* (1997) descrevem como um método internacional para marcação de texto, que os estruturava em parágrafos, cabeçalhos, listas, entre outros, foi utilizado como pilar principal para a criação do *Hyper Text Markup Language* (HTML), ou Linguagem de Marcação de Hipertexto em português. O fato de o SGML poder ser implementado em qualquer computador foi um ponto chave pois havia interesse que qualquer *browser* pudesse ler as informações disponibilizadas na *web*.

Utilizando das principais *tags* ou etiquetas em português do SGML, foram criadas as primeiras regras do HTML, com os P (*paragragh*, parágrafo em português), H1 até H6 (*heading*, título em português, dos níveis 1 a 6), OL e UL (*orded* e *unordered lists*, listas ordenadas e desordenadas em português), LI (*list itens*, itens da lista em português) entre outros. Uma das *tags* adicionadas especialmente para uso na internet foi a HREF, que tinha como objetivo ligar um documento ou computador a outro na mesma. Além disso, cada *tag* pode ter atributos internos para trazer maior customização no uso da mesma, como o *id* que identifica uma *tag* específica ou o *value* que define um valor ao elemento. A Figura 5 demonstra um exemplo de uso da sintaxe da linguagem HTML e o resultado da mesma em um *browser*.

Figura 5 - Exemplo de documento HTML

Fonte: PRÓPRIA, 2015, utilizando Sublime Text 2 e Microsoft Edge.

A fim de trazer maior customização e diferentes visuais para as páginas *web*, foi criado o *Cascading Style Sheets* (CSS), que segundo W3C (2015a) é uma linguagem de folhas de estilo utilizada para tratar de elementos gráficos exibidos, assim como definir tipos de letra, cores, espaçamento, entre outros, em uma página *web*. Tabeless (2006) cita a criação do CSS no ano de 1996, por Håkon Wium Lie e Bert Bos, ambos cientistas da computação, que criaram este padrão e conseguiram sua homologação junto a W3C para recomendação CSS para desenvolvimento de páginas *web*.

Dall'Golio (2007) ressalta que o objetivo do CSS é prover a separação entre o formato dos dados e seus conteúdos. Tem como função definir cores, gráficos, formatos e etc. Seu uso pode ser feito meio de comandos que são implementados em meio ao código HTML com a *tag style*, estilo em português, ou inserido dentro de algum elemento que permita o uso do atributo *style*, ou pode ser construído em um arquivo separado que é ligado ao documento posteriormente. Ele pode tratar elementos de acordo com a *tag*, *id* ou outros elementos do HTML. A Figura 6 mostra a aplicação do código CSS no exemplo mostrado na Figura 5, no qual utilizando do atributo *background* dentro da *tag body*, foi feita a alteração da cor do fundo da página, e com a *tag div* e os atributos *border e text-align* foram, criadas uma borda ao redor do texto "Hello World" e feita a centralização do mesmo.

Figura 6 - Exemplo de customização de uma página HTML com CSS

Fonte: PRÓPRIA, 2015, utilizando Sublime Text 2 e Microsoft Edge.

No desenvolvimento deste trabalho, foi utilizado a biblioteca CSS Metro UI CSS, que possui uma série de padrões de estilo baseadas no estilo de interfaces do sistema operacional Windows 8.

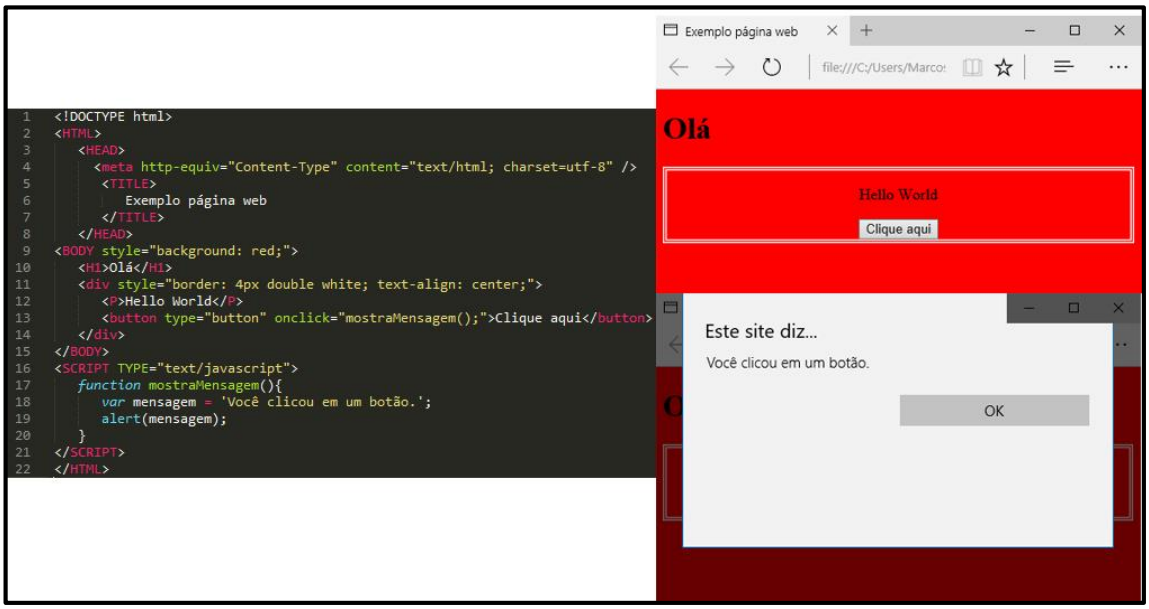
2.8.2 Javascript

O uso da internet trouxe diversas facilidades na troca de informações, e cada vez mais, passou a se ter a necessidade de maior interação entre a página e o usuário. As primeiras páginas da internet, que antes eram totalmente estáticas, passaram a ser mais tangíveis e dinâmicas com o advento dos *Applets* Java.

Os *applets* Java segundo Goodman (2007) são pequenos programas desenvolvidos em Java que são transferidos pelo navegador no momento da sua utilização e são apagados após o usuário se mover para outro lugar na internet, podendo assim executar ações como animações e alteração de elementos da página em tempo de execução durante a navegação. Porém, a utilização de *applets* era burocrática, já que a empresa responsável pelo navegador devia licenciar o uso dos mesmos e nem todos os computadores conseguiam executá-los com eficiência, além de que, aplicações maliciosas poderiam ser transferidas secretamente em conjunto dos *applets* para infecção dos dispositivos.

Goodman (2007) ainda cita que, em 1995, a Netscape, empresa detentora de um dos navegadores mais utilizados na época, em parceria com a Sun, que até então era proprietária do Java, iniciou o desenvolvimento do LiveScript, linguagem feita por meio de *scripts*, que são pequenos códigos que podem ser executados em tempo real em aplicativos, que tinha a função de acessar informações do servidor ou se conectar a outros serviços sem atualizar a página atual. Com a popularidade do LiveScript, seu nome foi mudado para JavaScript, mas apesar dos nomes parecidos, a linguagem não tem ligação direta com o Java, possuindo apenas a sintaxe semelhante. O JavaScript pode ser utilizado diretamente em conjunto com algumas *tags* HTML, como a *button*, que cria um botão e pode receber uma ação JavaScript com o atributo *onclick*. O código pode ser escrito dentro da *tag script* e assim, ao se abrir a página, o mesmo já é carregado automaticamente e fica pronto para uso. A Figura 7 mostra a aplicação deste exemplo.

Figura 7 - Utilizando código JavaScript



Fonte: PRÓPRIA, 2015, utilizando Sublime Text 2 e Microsoft Edge 20.1.

Buscando-se maior facilidade e maior gama de funções para o uso do JavaScript, foi utilizado o recurso jQuery, que segundo jQuery (2015), é uma biblioteca de funções rápida, pequena, e rica em características para facilitar a implementação de códigos em JavaScript. Esta biblioteca contém diversos tipos de funções que abstraem diversas rotinas padrões existentes no JavaScript, facilitando assim a manipulação de dados e de elementos HTML e CSS. Com o jQuery também é possível fazer o uso de funções *Asynchronous Javascript and* XML (AJAX), que tornam possíveis requisições ao servidor, sem que seja necessário o carregamento completo da página.

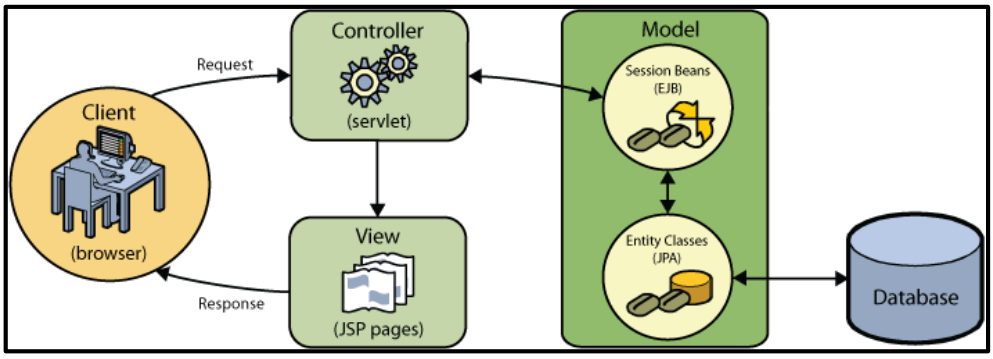
Para tratamento das informações retornadas pelo servidor durante uma requisição feita utilizando o AJAX, foi utilizado o JavaScript *Object Notation* (JSON) que a ECMA International (2013) descreve com um formato de texto que visa facilitar a troca de mensagens entre linguagens de programação. O JSON possui uma sintaxe específica para armazenamento de informações que permite então com que sejam enviadas entre diferentes ambientes da aplicação desenvolvida sem que se perca a sua integridade. Sendo assim, utilizando deste formato é possível a representação de objetos por meio de texto sem que se percam as informações armazenadas.

2.8.3 Model View Controller (MVC)

Segundo Dall'Oglio (2007) o modelo *Model View Controller* (MVC) ou modelo, visão e controlador, é um método de desenvolvimento *web* que possui três camadas, cada uma responsável por um aspecto do ambiente do sistema desenvolvido. Gamma *et al* (2000) define que o *Model* atua como o objeto de aplicação, a *View* na apresentação da interface na tela e o *Controller* gerencia a maneira como a interface do usuário reage às entradas do mesmo.

Como se pode observar na Figura 8, o Java EE trabalha de acordo com o modelo MVC, portanto pode ter suas especificações esquematizadas como tal. A camada *Model*, que tem como função a criação do objeto que representam itens que serão mostrados ao usuário e gravados no banco de dados, representado como os elementos EJB e JPA. O JPA provê a conexão direta com o banco de dados enquanto que o EJB, a cada usuário que utiliza o sistema, faz requisições ao JPA e captura os objetos necessários para serem manipulados.

Figura 8 - MVC utilizando Java EE



Fonte: NETBEANS, 2014.

A *View* é a parte em que o usuário tem interação direta, ou seja, onde estarão presentes os formulários e opções que o usuário pode preencher para serem gravados no banco de dados. Essa camada não tem nenhuma influência sobre a aplicação e somente apresenta os dados, não contendo assim regras de negócio. Na Figura 8 a *View* é representada pelo JSP, que no caso do Java EE, pode conter regras de negócio. O que acontece é que, a cada requisição, o servidor verifica as regras de negócio internas e as regras de negócio presentes no arquivo JSP, que pode conter além de códigos em HTML, JavaScript, entre outros, expressões em Java e só após isso, gera uma página em HTML que é enfim lida pelo *browser*. Desse modo, a *View* pode ser gerada de acordo com as informações que estão sendo manipuladas pelo usuário, permitindo o desenvolvimento de páginas totalmente dinâmicas.

O *Controller* é onde acontece o processamento de tudo que foi enviado pelo usuário, ou seja, onde se encontram todas as regras de negócio. Os dados que ele recebe são o conjunto dos objetos criados pelo *Model* e pela *View*, que serão, depois de validados, adicionados ou não ao banco de dados. Como se pode observar, este modelo traz vantagens para o desenvolvedor, pois como o MVC trabalha com camadas diferentes, alguns itens podem ser reutilizados, facilitando na manutenibilidade da aplicação e/ou sistema desenvolvido e diretamente na produtividade de desenvolvimento.

2.9 Integração de sistemas

Organizações estão cada vez mais informatizadas e, portanto dependem do uso de aplicações para que, seus gerentes possam administrá-las de forma eficaz e confiável. Acontece que, nem sempre empresas de pequeno porte conseguem custear o desenvolvimento de aplicações específicas para suas necessidades, optando assim pelo uso de uma ou mais aplicações multiempresariais, que podem atender a diversos nichos de mercado. Porém, nos casos em que são utilizadas mais de uma aplicação para o tratamento de uma grande quantidade de dados do mesmo tipo, torna-se necessário uma forma de interligação entre os diferentes sistemas utilizados na empresa, surgindo assim a necessidade da integração de sistemas.

Segundo Sordi e Marinho (2007), a integração de sistemas traz diversos benefícios para a organização como a redução do retrabalho, já que os dados entre os sistemas serão duplicados, fazendo com que os sistemas utilizados na empresa “interajam” entre si e se mantenham atualizados. A organização também passará a ter uma maior agilidade e capacidade de atender a novas demandas ou regras do mercado, pois alterações que possam ser necessárias em sua base de dados serão aplicadas com maior facilidade.

3 DESENVOLVIMENTO DO SISTEMA PROPOSTO

Neste capítulo é mostrado o processo de desenvolvimento do Sistema para Gerenciamento Comercial e como é feita a exportação de dados para os serviços de suporte presentes na VBC Automação Comercial LTDA. São listados métodos mais relevantes no desenvolvimento do sistema e quais ferramentas descritas anteriormente foram utilizadas a fim de juntas, possibilitarem o desenvolvimento do sistema proposto.

3.1 Ferramentas utilizadas

No desenvolvimento da aplicação, foram utilizados os programas:

- Eclipse Java EE IDE for Web Developers versão Kepler Service Release 1: Integrated Development Enviroment(IDE) para desenvolvimento de aplicações com uso do Java EE;
- Sublime Text 2.0.2: ferramenta para edição de códigos, utilizada para *design* inicial das interfaces;
- Notepad++ versão 6.5: edição de códigos;
- Apache Tomcat 7: servidor responsável pelo funcionamento da aplicação;
- Heidi SQL 9.1: sistema para manipulação de bancos de dados;
- Wampserver: controla o SGBD MySql, que permite o funcionamento do banco de dados e,
- Jaspersoft iReport Designer 5.6.0: responsável pelo *design* dos relatórios.

3.2 Configurações necessárias

O Quadro 1 mostra as configurações mínimas necessárias para se executar o servidor da aplicação.

QUADRO 1 - Requisitos básicos de configuração do servidor da aplicação

Número de ordem	Requisito	Limites aplicáveis
1	RAM	O servidor necessitará de no mínimo 2 GB.
2	HD	A aplicação necessitará de 100 MB (Exceto banco de dados).
3	Internet	É necessária conexão com a internet para o funcionamento.

Fonte: PRÓPRIA, 2015

No Quadro 2 são apresentadas as restrições do *software*, demonstrando quais aplicações foram utilizadas no desenvolvimento do mesmo, já que versões diferentes podem gerar erros na execução da mesma.

QUADRO 2 – Restrições do software desenvolvido

Número de ordem	Restrição	Descrição
1	Ambiente	A aplicação é multiplataforma, ou seja, pode ser executada em qualquer sistema operacional
2	Ambiente	O ambiente de desenvolvimento utilizado é o Java EE versão 7, HTML 5 e JavaScript 1.8
3	Ambiente	O sistema de gerenciamento de banco de dados utilizado é o MySQL versão 5.6.12
4	Ambiente	Foi utilizado o servidor Apache Tomcat versão 7.0.47 ou superior.

Fonte: PRÓPRIA, 2015

No Quadro 3 são mostradas as hipóteses de trabalho, para o uso da aplicação pelo usuário. As restrições devem ser seguidas para possibilitar o uso do aplicativo.

QUADRO 3 - Hipóteses de trabalho para o software desenvolvido

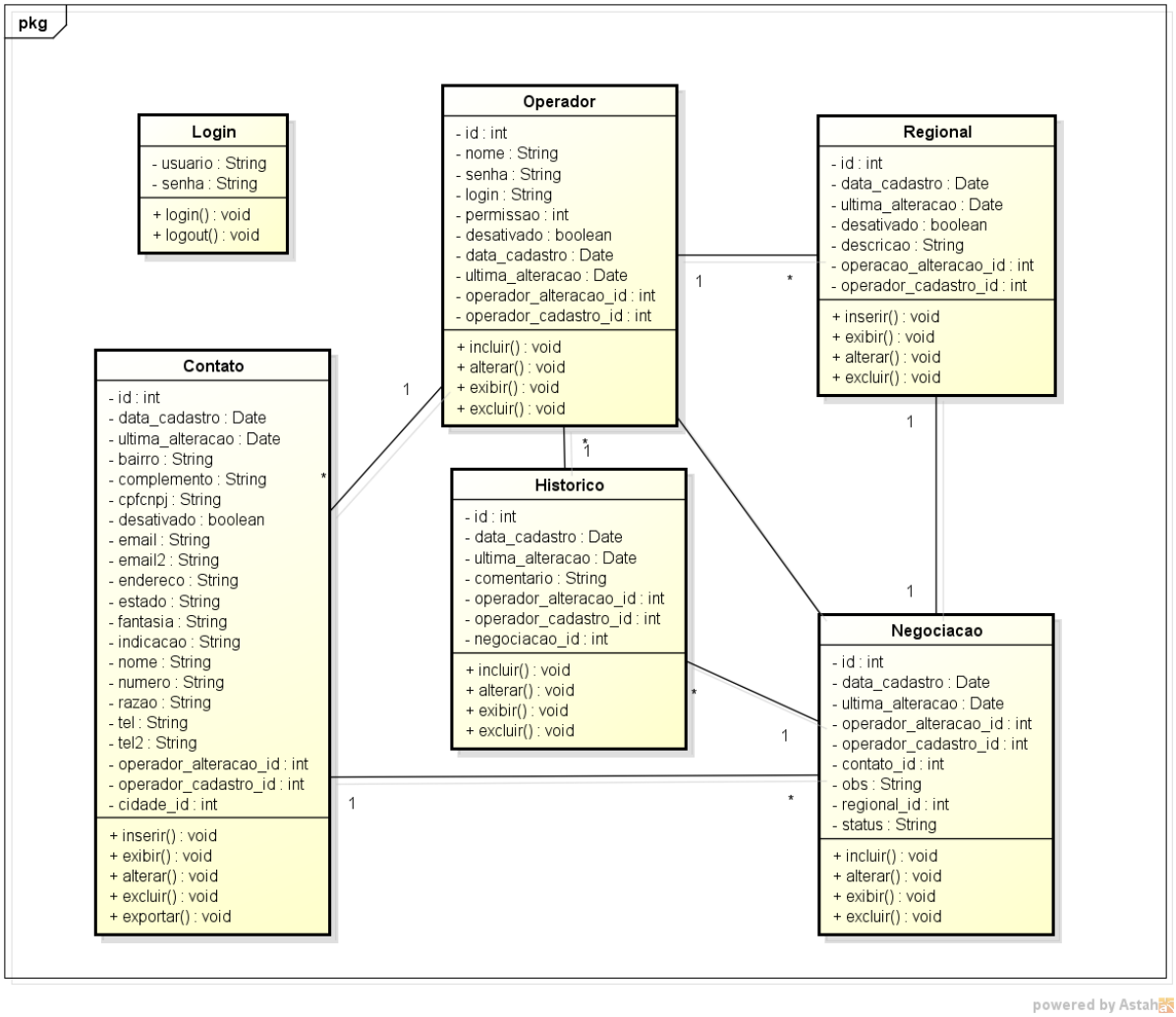
Número de ordem	Restrição	Descrição
1	Devem ser utilizados navegadores Google Chrome ou Mozilla Firefox nas suas versões mais recentes.	Do usuário
2	Deve ser utilizado os sistemas operacionais com suporte aos navegadores Google Chrome ou Mozilla Firefox.	Do usuário
3	Deve ser utilizado Windows Server 2009 ou superior	Da empresa contratada para hospedar a aplicação.
4	Deve se utilizar servidor MySQL 5.6.12 ou superior.	Da empresa contratada para hospedar a aplicação.
5	Deve se utilizar servidor Apache Tomcat versão 7.0.47 ou superior.	Da empresa contratada para hospedar a aplicação.

Fonte: PRÓPRIA, 2015

3.3 Requisitos de dados persistentes

Uma das características principais de uma aplicação é a integridade dos dados que serão inseridos pelos usuários. Para garantir essa qualidade, a aplicação deve conter uma modelagem de banco de dados que seja coerente e que garanta segurança e confiabilidade ao *software*. Segundo Pressman (2010), a modelagem com base em classes demonstra os objetos que o sistema irá manipular durante suas operações. A Figura 9 demonstra o diagrama de classe do sistema proposto.

Figura 9 - Diagrama de Classe



Fonte: PRÓPRIA, 2015, utilizando o Astah Comunity versão 6.7

Na Figura 10 é demonstrado um trecho do código de configuração do banco de dados na aplicação, retirada do arquivo “hibernate.cfg.xml”. Este arquivo garante a comunicação do Hibernate com o SGDB MySQL.

Figura 10 - Configuração da conexão do banco de dados

```
<property name="hibernate.connection.username">
    root
</property>
<property name="hibernate.connection.password">
</property>
<property name="hibernate.connection.url">
    jdbc:mysql://localhost/gerenciador
</property>
<property name="hibernate.connection.driver_class">
    com.mysql.jdbc.Driver
</property>
<property name="hibernate.dialect">
    org.hibernate.dialect.MySQL5InnoDBDialect
</property>
```

Fonte: PRÓPRIA, 2015, utilizando Notepad++ versão 6.5

3.4 Funcionamento do MVC com VRaptor

Para desenvolvimento do sistema, foram utilizados *frameworks* e conceitos do padrão MVC, que fazem parte de toda a estrutura do sistema e devem trabalhar de forma integrada para a boa execução da mesma. A seguir é mostrado como este padrão é aplicado no desenvolvimento do sistema, utilizando como base o *framework* VRaptor.

O padrão MVC é utilizado em grande parte das aplicações *web* existentes, pois possui três camadas, fazendo com que todas funcionem em conjunto para a execução do sistema. A primeira camada, o *Model*, como pode ser visto na Figura 11, é definida como uma classe que contém todos os atributos que serão manipulados na mesma e posteriormente salvos no banco de dados. Esta classe quando preenchida torna-se o objeto que irá transitar entre as camadas *View* e *Controller* do MVC, portanto, sua correta definição é o que garante a integridade do sistema e das informações digitadas pelo usuário.

Figura 11 - Trecho do código da classe Contato

```
@Index(name="contatoNomeIdx")
@Column(length=50, nullable=false)
private String nome;

@Column(length=100, nullable = true)
private String fantasia;

@Column(length=100, nullable = true)
private String razao;

@Index(name="contatoCpfCnpjIdx")
@Column(length=20, nullable = false)
private String cpfcnpj;

@Column(length=12, nullable = false)
private String tel;

@Column(length=12, nullable = true)
private String tel2;

@Column(length=100, nullable = false)
private String email;
```

Fonte: PRÓPRIA, 2015, utilizando Notepad++ versão 6.5

A camada *Controller*, como mostra a Figura 12, é responsável por grande parte das validações do sistema, assim como realiza o gerenciamento das requisições no banco de dados. Presente no início nas funções do *Controller* há o *link* que dá acesso a função e a *tag* @Restrito tem como importante função bloquear o acesso a mesma sem que tenha-se algum usuário autorizado pelo sistema.

Figura 12 - Trecho do código do *controller* Contato

```
124    @Restrito
125    @Get("/contato/novo")
126    public void formulario(Contato entity) {
127        if(entity == null){
128            entity = new Contato();
129        }
130        entity.setDesativado(false);
131        entity.setOperacao(Operacao.NOVO);
132        result.include("entity", entity)
133                .include("ufList",UF.values())
134                .include("tipoPessoaList", TipoPessoa.values());
135    }
136
137    @Restrito
138    @Post("/contato")
139    public void adiciona(Contato entity) {
140        if(validaFormulario(entity)){
141            if(operadorWeb.getLogado() != null){
142                entity.setOperadorCadastro(operadorWeb.getLogado());
143                entity.setOperadorAlteracao(operadorWeb.getLogado());
144                entity.setDataCadastro(new Date());
145                entity.setUltimaAlteracao(new Date());
146            }
147            try{
148                dao.salva(entity);
149                result.include("success", i18n("contato.cadastrado.com.sucesso"));
150                result.redirectTo(ContatoController.class).lista();
151            } catch (Exception e){
152                System.out.println(e);
153                result.include("error", e.getMessage());
154            }
155        } else {
156            result.redirectTo(this).formulario(entity);
157        }
158    }
```

Fonte: PRÓPRIA, 2015, utilizando Eclipse Java EE IDE for Web Developers, versão Kepler Service Release 1.

A camada *View*, como pode ser visto na Figura 13, é utilizada para organizar os dados enviados pelas outras camadas e exibir para o usuário, sendo assim a parte onde acontecerá a ponte entre as regras de negócio do sistema e o usuário. Nesta camada também podem aparecer regras de negócio, já que antes de ser de ser exibido ao usuário, o Java EE, utilizando das notações do JSP, pode aplicar regras durante o carregamento da mesma.

Figura 13 - Trecho do código da *view* Regional

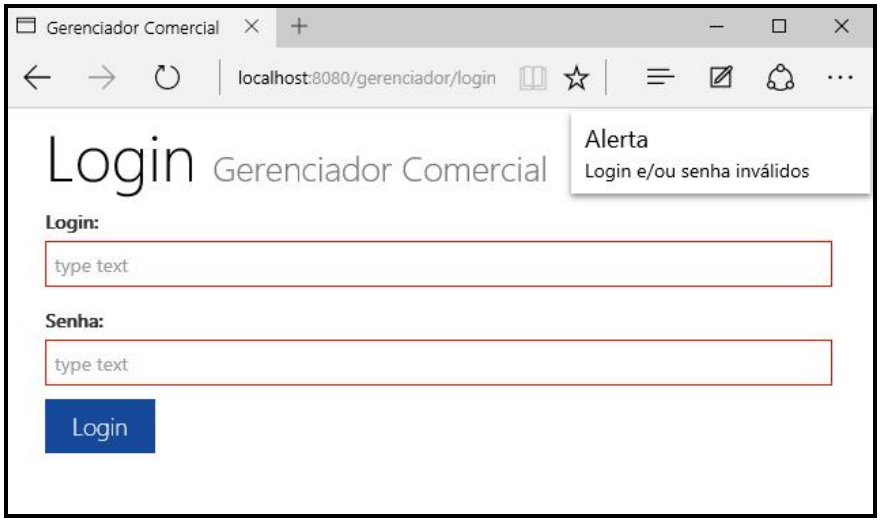
```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<div class="container">
    <div class="grid">
        <geren:cabecalhoForm
            operacao="${entity.operacao.label}"
            controller="${entity.entidade}"
            entidade="${entity.entidade.label}"
        />
            <form id="negociacaoForm" action="${pageContext.request.contextPath}/negociacao" method="POST">
                <input type="hidden" name="entity.operacao" value="${entity.operacao}">
                <c:if test="${entity.id != null}">
                    <input type="hidden" name="_method" value="PUT" />
                    <input type="hidden" id="id" name="entity.id" value="${entity.id}">
                </c:if>
                <c:if test="${entity.operacao eq 'EXIBINDO'}">
                    <div class="row">
                        <div class="col-xs-5 col-sm-3 col-md-2"">
                            <label>Código:</label>
                            <div class="input-control text" data-role="input-control">
                                <input type="text" disabled placeholder="type text" name="entity.id" value="${entity.id}">
                            </div>
                        </div>
                    </div>
                    <br/>
                </c:if>
```

Fonte: PRÓPRIA, 2015, utilizando Eclipse Java EE IDE for Web Developers, versão Kepler Service Release 1.

3.5 Tela de *Login* e tela inicial

A tela de *Login* do sistema, como pode ser observada na Figura 14, contém apenas os campos básicos para acesso ao sistema e já mostra a tela com suas validações. Essas validações provêm a segurança dos dados inseridos no sistema, pois o usuário só terá acesso a ele mediante ao *login* e de acordo com suas permissões. As validações são feitas unindo recursos do da *View*, utilizando JSP e JavaScript e CSS, e de validações feitas no *Controller* e no *Model*.

Figura 14 - Tela de *Login* com validações



Fonte: PRÓPRIA, 2015, utilizando Microsoft Edge 20.1

No sistema, campos que recebem algum tipo de validação, contêm a classe CSS error-state. Esta classe é responsável por contornar o campo de texto em vermelho, assim destacando-o. Para que este campo seja ativado somente quando necessário, é utilizado da *tag* JSP <c:if>. A Figura 15 mostra o código do campo *Login* mostrado anteriormente.

Figura 15 - Trecho do código da view Login

```
16  <div class="row">
17      <label for="login">Login:</label>
18      <div class="input-control text <c:if test="${errorLogin}">error-state</c:if>" data-role="input-control">
19          <input type="text" placeholder="type text" id="login" name="operador.login">
20          <button class="btn-clear" tabindex="-1"></button>
21      </div>
22  </div>
```

Fonte: PRÓPRIA, 2015, utilizando Eclipse Java EE IDE for Web Developers, versão Kepler Service Release 1.

A *tag* <c:if> tem a função de verificar alguma condição imposta pelo desenvolvedor, sendo que o conteúdo dentro da mesma só será exibido caso a condição seja contemplada. Sendo assim, a mesma foi adicionada dentro da classe que define o campo *Login*, fazendo com que, caso a variável errorLogin, presente na linha 18 da Figura 15, esteja definida como verdadeiro, o texto contido na classe seja adicionado a *View* no momento de seu carregamento. A variável errorLogin é preenchida, caso a verificação de *login* e senha de usuário falhe, indicando, assim, que os dados digitados não são válidos. O uso das *tags* <c:if> está presente em todo o sistema e permite o controle dos dados que serão exibidos para o usuário.

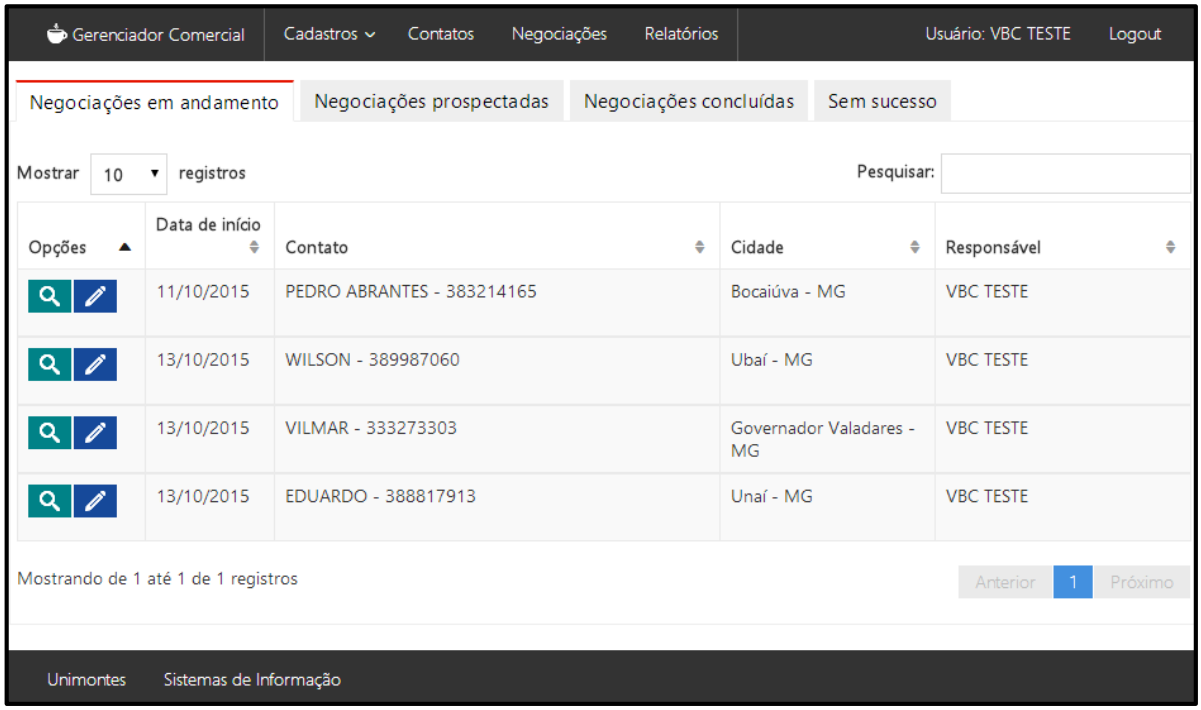
Usuários mais avançados podem observar os *links* que aparecem no navegador conforme se usa o sistema e com isso, tentar acessar rotinas não permitidas, Porém, com os controles realizados diretamente nas funções do *Controller* com a *tag* @Restrito estes *links* tem seu acesso bloqueado, forçando assim o *login*.

Com o controle de usuário são definidos também os tipos de usuário, que são administrador, normal e desenvolvedor. A permissão de Administrador implica na possibilidade de cadastrar e alterar todos os registros disponíveis no sistema, enquanto que, usuários comuns podem somente visualizar os dados não cadastrados por ele e inserir novos dados no sistema. O usuário desenvolvedor tem as mesmas atribuições do usuário administrador, porém tem acesso irrestrito a funções que são bloqueadas, como a edição de dados em rotinas em que a mesma é bloqueada, como por exemplo, as Negociações.

A Figura 16 destaca a tela inicial do sistema, onde são mostradas as primeiras informações para os usuários. Para prover agilidade em localizar as negociações, que são a parte mais acessada do sistema, as principais são mostradas já na tela, sendo as 10 últimas

negociações que foram acessadas pelos usuários. As mesmas são divididas em suas categorias, que são negociações em andamento, prospectadas, concluídas e sem sucesso. Para usuários normais só é exibido os dados referentes às negociações que o mesmo iniciou.

Figura 16 - Tela inicial do sistema



Fonte: PRÓPRIA, 2015. Utilizando Microsoft Edge 20.1

Para o carregamento das tabelas também são utilizadas *tags* JSP. Neste caso, usa-se o comando <c:forEach> que recebe uma variável do tipo lista e permite com que, para cada item presente, seja feita uma ação. Para esta tabela, são retornadas listas contendo os dados referentes a cada aba da tela inicial. Cada posição da lista contém um objeto do tipo Negociação, contendo seus dados, fazendo com que seja necessário somente posicionar as variáveis da lista em sua devida coluna, como é mostrado na Figura 17.

Figura 17- Utilização da tag <c:forEach>

```
<c:forEach var="entity" items="${negociacaoEmAndamento}">
    <tbody>
        <tr>
            <td>
                <div class="toolbar">
                    <geren:botaoVisualizarListagem
                        controller="${entity.entidade}"
                        id="${entity.id}"
                        somenteLeitura="false"
                    />
                    <geren:botaoEditarListagem
                        controller="${entity.entidade}"
                        id="${entity.id}"
                        somenteLeitura="false"
                    />
                </div>
            </td>
            <td><fmt:formatDate value="${entity.dataInicio}" pattern="dd/MM/yyyy"/></td>
            <td>${entity.contato.nome} - ${entity.contato.telF}</td>
            <td>${entity.contato.cidade.nome} - ${entity.contato.cidade.uf}</td>
            <td>${entity.operadorCadastro.nome}</td>
        </tr>
    </tbody>
</c:forEach>
```

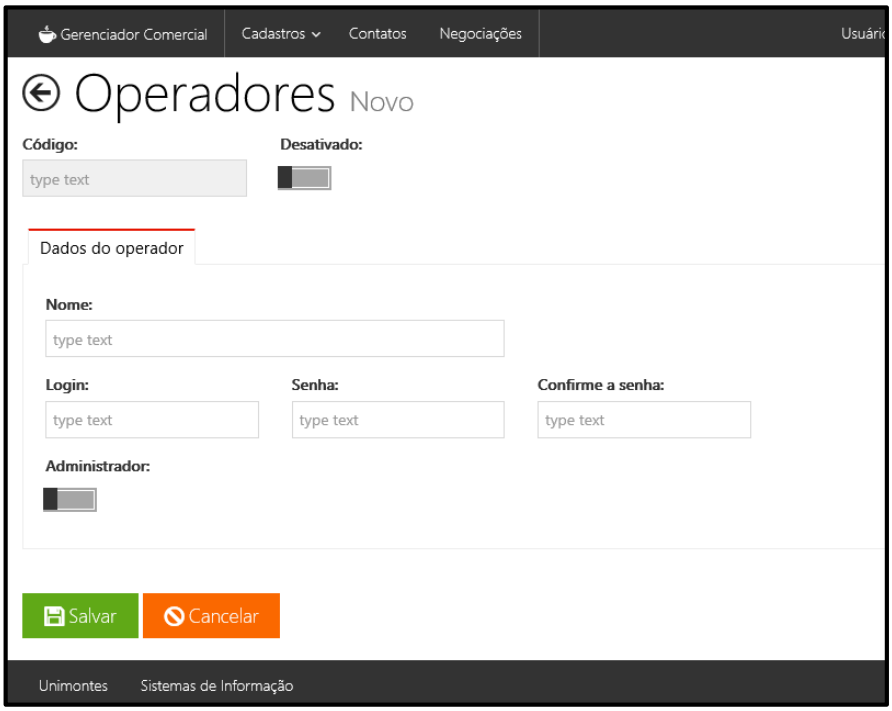
Fonte: PRÓPRIA, 2015, utilizando Eclipse Java EE IDE for Web Developers, versão Kepler Service Release 1.

3.6 Tela cadastro de Operadores

A tela para cadastro de operadores é acessada por todos os tipos de usuário, porém somente usuários do tipo Administrador ou Desenvolvedor podem realizar novos cadastrados. O usuário normal tem disponível somente a opção de editar o seu próprio cadastro, para que possa alterar o seu *login* e/ou senha.

Como se pode observar na Figura 18, na tela de cadastro, o botão de ação “Administrador” discrimina os tipos de usuário, quando ativado ou desativado. O mesmo acontece com o botão “Desativado”, que quando ativo, impede que o usuário modificado tenha acesso ao sistema e o registro fica inacessível em outras partes do sistema para inserção de novos dados. O comportamento do botão “Desativado” também é presente em outras rotinas do sistema, tendo este como um padrão do mesmo.

Figura 18 - Tela Cadastro de Operadores



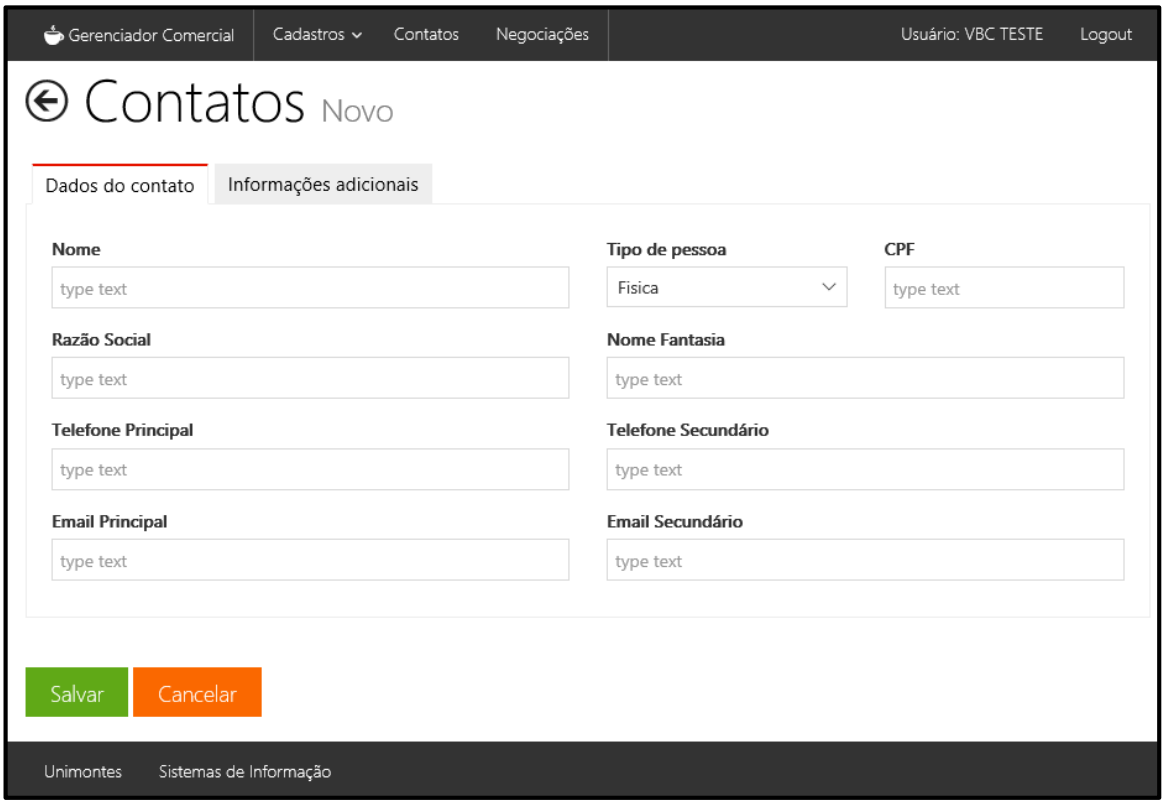
Fonte: PRÓPRIA, 2015. Utilizando Microsoft Edge 20.1

O usuário Desenvolvedor também não é listado no cadastro de operadores do sistema, tendo seu controle realizado internamente no mesmo, a fim de evitar que outros usuários tentem alterar seus dados. Essa validação se faz necessária para garantir que não sejam realizadas alterações sem a autorização necessária e assim não comprometer dados inseridos por outros usuários.

3.7 Tela de cadastro de Contatos

A tela de cadastro de Contatos tem função de agregar os dados dos registros dos contatos que foram inseridos no sistema. Como pode ser observada na Figura 19, a tela possui um formulário para inserção de informações do registro de contato, tais como nome, razão social, telefone, entre outras. Para esta tela são aplicadas algumas funções JavaStript utilizando de recursos das bibliotecas jQuery e AJAX.

Figura 19- Tela cadastro de Contatos



Fonte: PRÓPRIA, 2015. Utilizando Microsoft Edge 20.1

O seletor “Tipo de pessoa” permite ao usuário selecionar se o novo cadastro será um contato que responde por pessoa física ou jurídica, alterando assim a formatação aplicada no campo CPF/CNPJ. Por meio do código mostrado na Figura 20, é feita a leitura do valor no campo “Tipo de pessoa” e feita a alteração no campo CPF/CNPJ e definida a validação a ser feita. O método jQuery *mask* permite ao desenvolvedor a definir uma máscara personalizada ao campo de texto indicado, fazendo com que haja uma validação inicial dos dados informados pelo usuário.

Figura 20 - Código responsável pela formatação do campo CPF/CNPJ

```
function formataCampos(){
    if($("#tipoPessoaSelect").val() == "F"){
        $("#label-cpfCnpj").text("CPF");
        $("#cpfCnpj").mask("999.999.999-99");
    } else if ($("#tipoPessoaSelect").val() == "J"){
        $("#label-cpfCnpj").text("CNPJ");
        $("#cpfCnpj").mask("99.999.999/9999-99");
    }
}

$("#tipoPessoaSelect").on("change", function(){
    formataCampos();
});

$("#cpfCnpj").on("blur", function(){
    var valor = $("#cpfCnpj").val();
    if(valor != null && valor != "" && !valida_cpf_cnpj(valor)){
        mensagemAlerta("Número de documento inválido", "error");
        $("#div-cpfCnpj").addClass("error-state");
    } else {
        $("#div-cpfCnpj").removeClass("error-state");
    }
});
```

Fonte: PRÓPRIA, 2015, utilizando Eclipse Java EE IDE for Web Developers, versão Kepler Service Release 1.

Por meio de uma função feita em jQuery, é realizada a filtragem dos registros de cidade, com base no estado selecionado pelo usuário. A função “atualizaCidade” conforme mostrada na Figura 21, faz com que, ao se alterar o campo estado, seja enviada uma requisição ao *controller*, na função jsonCidade, utilizando da biblioteca AJAX, executando assim uma pesquisa na tabela Cidades com base no estado selecionado pelo usuário. O *controller* então retorna uma lista com os registros encontrados e a mesma redesenha o seletor de cidades com base nos novos registros.

Figura 21 - Função "atualizaCidade"

```
function atualizaCidade(path, codCidade) {
    var selecionado = $("#estadosSelect").val();
    if (selecionado.trim() != 0){
        $.ajax({
            url : path + "/cidade/jsonCidade",
            data : {
                estado : selecionado
            },
            type : "POST",
            dataType : "json",
            success : function(json) {
                var options = [];
                for (var i = 0; i < json.list.length; i++) {
                    if (json.list[i].id == codCidade) {
                        options.push('<option value="' + json.list[i].id
                                + '" selected="selected">' + json.list[i].nome
                                + '</option>');
                    } else {
                        options.push('<option value="' + json.list[i].id + '">'
                                + json.list[i].nome + '</option>');
                    }
                }
                $('#cidadesSelect').find('option').remove().end();
                $('#cidadesSelect').html(options.join(''));
            },
            error : function() {
                alert('erro');
                $('#cidadesSelect').html('<option selected="selected" value="0"></option>');
            }
        });
    } else {
        $('#cidadesSelect').html('<option selected="selected" value="0"></option>');
    }
}
```

Fonte: PRÓPRIA, 2015, utilizando Sublime Text 2.

Com base no resultado recebido da pesquisa, a variável do tipo *lista options* recebe os valores do resultado, que estão armazenados na variável JSON. Com os valores armazenados, a *lista options* é preenchida com os valores retornados da pesquisa em conjunto com a *tag* HTML <option>, construindo assim, uma nova lista e adicionando-a na página em tempo de execução.

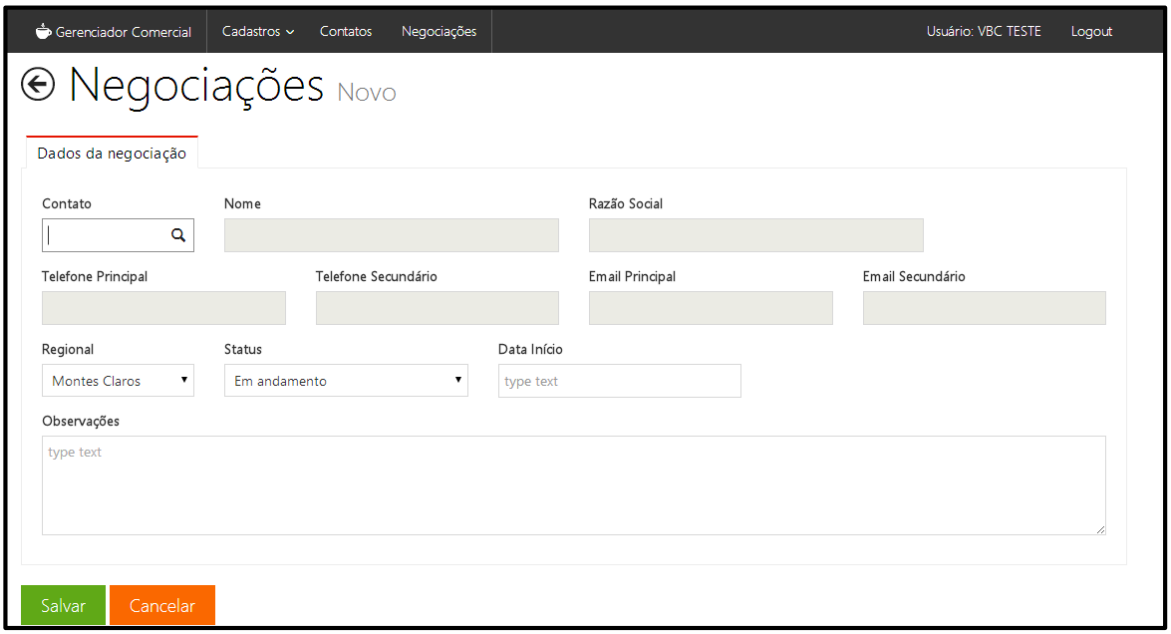
A partir da observação deste item é possível perceber a versatilidade e importância de se trabalhar com JavaScript em ambientes *web*. Com as funções representadas nas Figuras 20 e 21 foi possível modificar aspectos do formulário de cadastro em tempo real, inclusive realizando requisições no *Controller*. Esta possibilidade traz mais rapidez e fluidez na aplicação desenvolvida, pois não se faz mais necessário o carregamento da página inteira somente para alterar um ou mais campos da mesma.

3.8 Tela de cadastro de Negociações

A tela cadastro de Negociações se faz como a mais importante do sistema. É nesta rotina que serão realizadas as operações que, com base nos dados inseridos, terão os relatórios

gerados e processados e o controle das informações cadastradas por usuários. Como se pode visualizar na Figura 22, a tela é composta pelos dados dos Contatos cadastrados, dados referentes à unidade regional responsável (que foram cadastradas anteriormente pela rotina de Cadastro de Regionais) e de uma tabela contendo os comentários referentes ao histórico da negociação. No momento do cadastro de uma negociação, as opções referentes ao histórico ainda não são mostradas ao usuário, pois a mesma necessita que a negociação já esteja cadastrada para poder ter a sua referência no banco de dados.

Figura 22 - Tela cadastro de Negociações



Fonte: PRÓPRIA, 2015, utilizando Microsoft Edge 20.1

O JSP permite a criação de *tags* personalizadas, que executam funções específicas definidas pelo desenvolvedor. Estas *tags* podem ser utilizadas para criação de ações e *layouts* que estarão presentes em vários locais da aplicação desenvolvida, como todos os botões presentes no sistema. A Figura 23 mostra a declaração da *tag* campoPromptContato que contém os campos que mostram informações referentes a contatos na parte superior da tela de Negociações (FIG.22).

Figura 23 - Declaração da tag campoPromptContato

```
33  <geren:campoPromptContato
34      label="contato"
35      id="inputContato"
36      valor="${entity.contato.id}"
37      autofocus="true"
38      name="entity.contato.id"
39      readonly="${!(entity.operacao eq 'NOVO')}"
40      error="errorContato"
41      exibicaoCompleta="true"
42  />
```

Fonte: PRÓPRIA, 2015, utilizando Eclipse Java EE IDE for Web Developers, versão Kepler Service Release 1.

No uso das *tags* personalizadas, o desenvolvedor pode declarar variáveis que serão usadas internamente na mesma para o processamento das informações. As informações contidas são carregadas no momento em que a *view* é processada, seguindo assim os aspectos do padrão MVC descrito por Caelum (2014), na qual a *view* é escrita utilizando a sintaxe HTML em conjunto com diversas *tags* JSP, que no momento do carregamento da página, são removidas. Assim é gerada a página com HTML, JavaScript e CSS puros.

As variáveis destacadas na declaração da *tag* (FIG. 23) podem ser de qualquer tipo, desde que sejam corretamente declaradas internamente, como é mostrado na Figura 24. Ela demonstra um trecho da *tag* citada anteriormente onde são feitas as declarações das variáveis utilizadas para construção da mesma.

Podem ser declaradas também bibliotecas específicas para serem usadas na *tag*. Neste caso foram utilizadas as bibliotecas core, que traz diversas funções padrão do Java e a biblioteca fmt que permite o carregamento de mensagens do sistema e a formatação de dados numéricos. O uso das *tags* é recomendado por Caelum (2013c), pois faz com que haja uma maior padronização de código entre as aplicações desenvolvidas e também evitariam o uso demasiado de *scripts* diretamente no JSP, garantindo maior estabilidade e segurança na aplicação.

Figura 24 - Declaração das variáveis utilizadas na tag campoPromptContato

```
1  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
2  <%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
3  <%@ taglib tagdir="/WEB-INF/tags" prefix="geren"%>
4  <%@ attribute name="id" required="true"%>
5  <%@ attribute name="label" required="true"%>
6  <%@ attribute name="name" required="false"%>
7  <%@ attribute name="valor" required="false"%>
8  <%@ attribute name="error" required="false"%>
9  <%@ attribute name="autofocus" required="false" type="java.lang.Boolean"%>
10 <%@ attribute name="readonly" required="false" type="java.lang.Boolean"%>
11 <%@ attribute name="exibicaoCompleta" required="true" type="java.lang.Boolean"%>
```

Fonte: PRÓPRIA, 2015, utilizando Eclipse Java EE IDE for Web Developers, versão Kepler Service Release 1.

Na Figura 25 é demonstrada a função em Javascript inclusa na *tag* campoPromptContato, que usa da biblioteca AJAX para fazer o carregamento dos dados referentes aos contatos. A função utilizada faz uma requisição ao servidor, que pesquisa o registro no banco de dados conforme o valor que usuário digitar no campo *id*, presente na *tag*. A *Uniform Resource Locator* (URL) que é declarada na linha 116 da Figura 25, chama a função jsonObjContato, que está presente dentro do *controller* responsável por gerenciar suas informações. A função recebe a variável *id*, que é digitada pelo usuário. Após realizar a pesquisa, é retornado um objeto do tipo JSON, que contém uma lista com os resultados obtidos. Caso o objeto esteja preenchido, cada valor incluso no mesmo é preenchido no campo de texto. A linha 121 mostra o momento em que é verificado se a variável JSON está preenchida e caso esteja, os dados são adicionados em seus devidos campos.

Figura 25 - JavaScript responsável por buscar dados de um Contato

```
115         $.ajax({
116             url: "${pageContext.request.contextPath}/contato/jsonObjContato",
117             data:{id: inputId},
118             type: "POST",
119             dataType: "json",
120             success: function(json){
121                 if(json.list){
122                     $('#inputNomeContato').val(json.list[0]);
123                     $('#inputRazaoContato').val(json.list[1]);
124                     $('#inputTelContato').val(json.list[2]);
125                     $('#inputTel2Contato').val(json.list[3]);
126                     $('#inputEmailContato').val(json.list[4]);
127                     $('#inputEmail2Contato').val(json.list[5]);
128                 }else{
129                     $('#inputNomeContato').val("");
130                     $('#inputRazaoContato').val("");
131                     $('#inputTelContato').val("");
132                     $('#inputTel2Contato').val("");
133                     $('#inputEmailContato').val("");
134                     $('#inputEmail2Contato').val("");
135                 }
136             },
137             error: function(){
138                 $('#inputNomeContato').val("");
139                 $('#inputRazaoContato').val("");
140                 $('#inputTelContato').val("");
141                 $('#inputTel2Contato').val("");
142                 $('#inputEmailContato').val("");
143                 $('#inputEmail2Contato').val("");
144             }
145         });
```

Fonte: PRÓPRIA, 2015, utilizando Eclipse Java EE IDE for Web Developers, versão Kepler Service Release 1.

Para o cadastro de dados referentes ao histórico, a tela de cadastro de Negociações deve ser aberta no modo de edição ou alteração. Dessa forma, é exibida na parte posterior da tela, uma sessão contendo os botões de ação e a listagem dos históricos que foram registrados para a negociação selecionada, conforme se pode observar na Figura 26.

Figura 26 - Tela de Negociações exibindo listagem do histórico

Fonte: PRÓPRIA, 2015, utilizando Microsoft Edge 20.1

Os dados cadastrados na sessão de histórico não podem ser alterados por um usuário do tipo comum, já que os mesmos devem refletir as ações tomadas pelo mesmo. Estas informações podem ser usadas posteriormente para avaliação feita pelos gerentes, que poderão também ter acesso às operações de edição e remoção destes itens. Este controle de permissão é feito por meio de um controle no JSP e no *controller*. No momento em que o usuário acessa a rotina, o *controller* envia para a *view* um objeto do tipo Operador, preenchido com os dados do usuário que está com a sessão ativa no momento, bloqueando assim o acesso aos botões de ação de edição e remoção.

Como se pode observar no trecho destacado na Figura 27, que mostra o código presente na *tag* botãoVisualizarListagem, é recebida a variável somenteLeitura, que assume os valores verdadeiro ou falso. No momento do processamento da mesma, a *tag* JSP <c:if> verifica o valor da variável e faz com que o código HTML que está presente na mesma seja carregado para o usuário, de acordo com a permissão vinculada a ele. Neste trecho, a *tag* verifica o tipo do usuário que está carregando a página, e caso o mesmo não tiver a permissão, o botão é carregado com o atributo *disabled*, que impede que se tenha acesso a função, já que não é possível clicar no botão que direciona para a mesma.

Figura 27 - Definição da tag botaoEditarListagem

```
1  <%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
2  <%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
3  <%@ attribute name="controller"     required="true"  type="java.lang.String"%>
4  <%@ attribute name="id"     required="true"  type="java.lang.String"%>
5  <%@ attribute name="somenteLeitura"     required="true"  type="java.lang.Boolean"%>
6
7  <button
8      class="default primary"
9      type="button"
10     <c:if test="${somenteLeitura}">disabled</c:if>
11     onclick="window.location.href='${pageContext.request.contextPath}/${fn:toLowerCase(controller)}/${id}/edita'">
12     <a title="Editar"><i class="icon-pencil"></i></a>
13 </button>
```

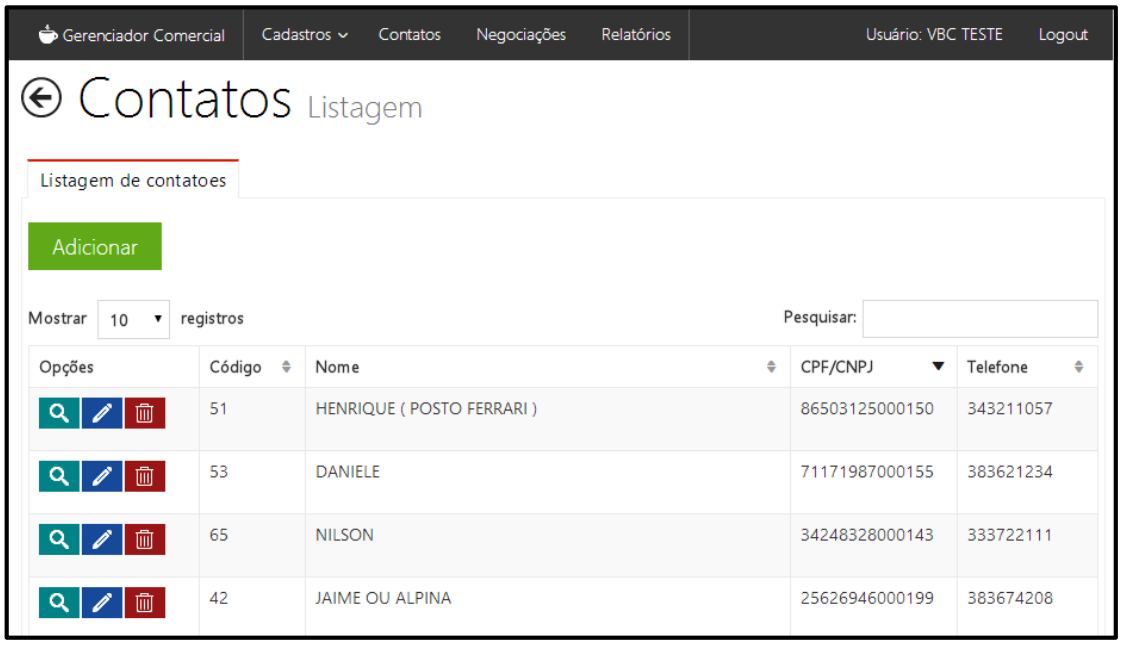
Fonte: PRÓPRIA, 2015, utilizando Eclipse Java EE IDE for Web Developers, versão Kepler Service Release 1.

É importante ressaltar que, por mais que o controle seja feito na *view*, algum usuário com conhecimento avançado pode tentar fazer o acesso direto na função por meio do *link* que direciona a mesma ou então removendo manualmente o atributo *disabled* do botão criado, pois a *tag* @Restrito só impede o acesso a rotina, caso não se tenha algum *login* ativo no momento. Desse modo, todas as requisições tratadas pelo *Controller* que possuem algum tipo de restrição, devem possuir uma verificação de tipo de usuário, garantindo assim, a segurança e a hierarquia de permissões no sistema.

3.9 Tela de listagem

As telas de listagem presentes no sistema são exibidas ao se acessar alguma rotina do mesmo, que pode ser visualizada na Figura 28.

Figura 28 - Tela Listagem de Contatos



Fonte: PRÓPRIA, 2015, utilizando Microsoft Edge 20.1.

As telas seguem um só padrão de *layout*, já que o *design* é feito utilizando de *tags* personalizadas que fazem o desenho do cabeçalho da tela e da tabela, contendo botões de ação adicionar, ao topo, e o grupo de botões visualizar, editar e excluir para cada linha carregada na tabela de listagem.

Com o campo pesquisar presente no topo das tabelas é possível fazer, pesquisas na mesma, filtrando os dados conforme o usuário digita, de acordo com os campos presentes na tela. Desse modo, se na tabela são exibidos os campos id, nome e endereço, por exemplo, estes itens serão os parâmetros utilizados na pesquisa. Com este comportamento de pesquisa e exibição bem como o *layout*, evita se que cada rotina tenha sua própria sessão de listagem e pesquisa, fazendo com que seja mantido um padrão na exibição. Este controle é feito por meio da biblioteca jQuery em conjunto com o *plug-in* Databables, e seu código pode ser visto na Figura 29.

Figura 29 - Uso da biblioteca Datatable

```
1  function montarTabelaNegociacao(path){
2      table = $("#tabelaNegociacao").dataTable({
3          retrieve: true,
4          responsive : true,
5          bAutoWidth : true,
6          bPaginate : true,
7          bFilter : true,
8          bSort : true,
9          bInfo : true,
10         bJQueryUI : false,
11         sPaginationType : 'simple_numbers',
12         bProcessing : false,
13         columnDefs: [ {"targets": 0, "width": 150 },
14                       {"targets": 1, "width": 100 },
15                       {"targets": 2, "width": 150 },
16                       { "orderable": false, "searchable": false, "targets": 0, "width": 86 }],
17         oLanguage : {
18             sUrl : path + '/files/datatables/js/dataTables.pt_BR.txt',
19             sSearch : 'Pesquisar:'
20         }
21     });
22 }
```

Fonte: PRÓPRIA, 2015, utilizando Eclipse Java EE IDE for Web Developers, versão Kepler Service Release 1.

O *plug-in* Datatables faz a personalização e controle automático de tabelas HTML. O controle é feito totalmente do lado do cliente, já que usa do JavaScript. Há diversos parâmetros disponíveis para a personalização das tabelas como responsividade, ordenação, paginação, idioma, padrões no tamanho de colunas, entre outros. A biblioteca também permite que o processamento das tabelas seja feito do lado do servidor, o que garante mais rapidez no processamento quando se trata de um grande volume de dados a serem exibidos. Para se aplicar o *plug-in*, é definido um *id* para a tabela desejada e então o mesmo é acionado utilizando do comando mostrado na linha 2 da Figura 29. A partir daí podem ser definidos os parâmetros para funcionamento do mesmo e assim, o acionamento da função, que pode ser feito por meio do jQuery ou de uma JavaScript.

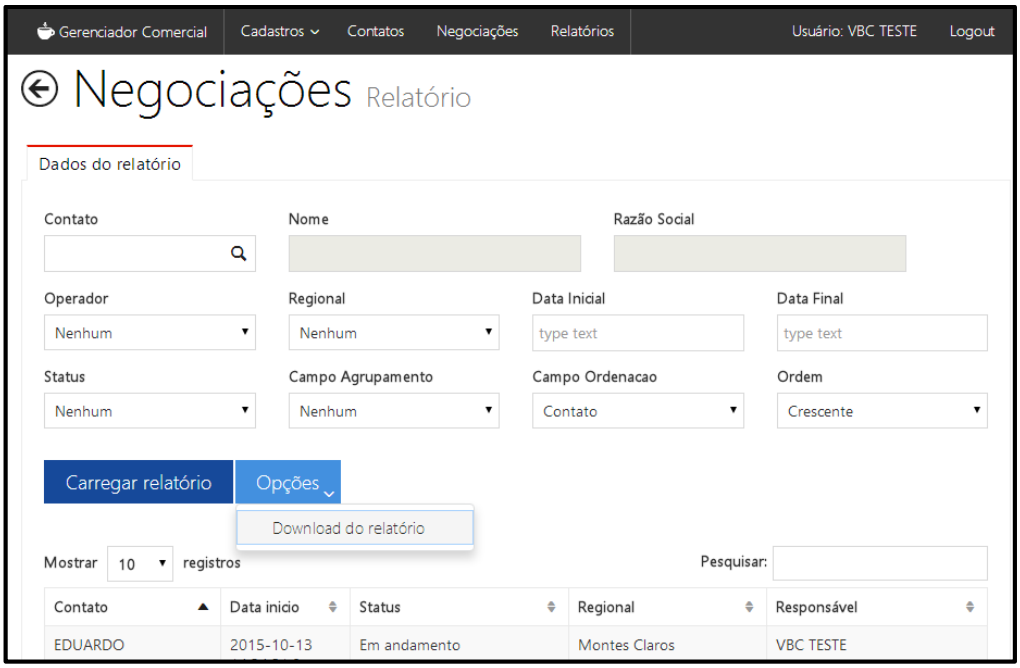
3.10 Funcionamento dos relatórios do sistema

O gerenciamento das informações inseridas no sistema só se faz possível com a emissão de relatórios quantitativos sobre o desempenho de cada operador vinculado ao sistema. Os relatórios estão localizados na mesma função, sendo que o usuário é quem deve fazer a configuração para que o mesmo seja gerado. Este tratamento traz flexibilidade nas informações que serão exibidas pelo usuário, não sendo necessário a sua pré-definição.

Conforme pode se ver na Figura 30, a tela de relatórios traz diversas informações ao usuário, como intervalo de datas e nome do contato por exemplo. Conforme as

informações preenchidas pelos usuários, o relatório faz as filtragens necessárias. Campos que não tem sua informação preenchida, não alteram os resultados emitidos pelo mesmo.

Figura 30 - Tela Relatórios



Fonte: PRÓPRIA, 2015, utilizando Microsoft Edge 20.1.

Para manipular os dados digitados pelos usuários, foi criada uma classe chamada NegociacaoRel. Ela possui os dados que são mostrados na tela de relatórios além de possuir os métodos do *framework* JasperReports, que faz a manipulação de informações e gera o relatório em vários formatos, como PDF, XLS, TXT, entre outros. O JasperReports usa de arquivos do tipo jrxml, que tem sua base feita em XML. Neste arquivo estão definidas as configurações do relatório que será gerado como variáveis, posicionamentos, temas e etc. Os arquivos jrxml podem ser manipulados manualmente, via código, utilizando da ferramenta iReport. Com ela é possível definir o *design* do relatório de maneira mais simples, utilizando da interface gráfica do programa.

Após preencher ou não os dados, o usuário deve clicar no botão carregar. As informações serão processadas e exibidas na tela para que o usuário visualize os dados e assim fazer o *download* do mesmo em formato PDF. Para se garantir a proteção dos dados do sistema, as permissões de usuário também possuem influência para emissão de relatórios. Usuários do tipo comum que utilizarem a rotina só terão acesso aos dados referentes às suas operações do sistema, portanto só serão exibidas suas informações das negociações. Esta

verificação se faz necessária para se garantir a confidencialidade entre os registros cadastrados pelos usuários comuns, sendo que o poder de gerência só é garantido aos usuários administradores. O funcionamento da função que gera o arquivo PDF pode ser observado na Figura 31.

Figura 31 - Trecho do código responsável por gerar relatórios

```
59  public File imprimir(List<Negociacao> negociacao) throws Exception  {
60      String nomeRelatorio = "relatorio.pdf";
61      JasperReport report = JasperCompileManager.compileReport(this.getPathToReportPackage() + "Negociacao.jrxml");
62
63      JasperPrint print = JasperFillManager.fillReport(report, null, new JRBeanCollectionDataSource(negociacao));
64
65      JasperExportManager.exportReportToPdfFile(print, this.path+"br/com/gerenciador/temp"+nomeRelatorio);
66
67      return new File(this.path+"br/com/gerenciador/temp"+nomeRelatorio);
68  }
```

Fonte: PRÓPRIA, 2015, utilizando Eclipse Java EE IDE for Web Developers, versão Kepler Service Release 1.

A função imprimir recebe uma variável do tipo *List* negociação, que contém uma lista de objetos do tipo negociação. O *framework* JasperReport recebe o arquivo do jrxml Negociacao.jrxml que foi criado com o programa iReport e, utilizando da função JasperCompileManager, preenchendo a variável report. A partir daí, a função JasperFillManager, faz a leitura da lista e da variável report e a transcrição da mesma para o tipo de arquivo definido pelo desenvolvedor com a função JasperPrint. Dependendo da aplicação utilizada, o tipo de arquivo pode ser definido pelo usuário, de acordo com a necessidade da aplicação desenvolvida.

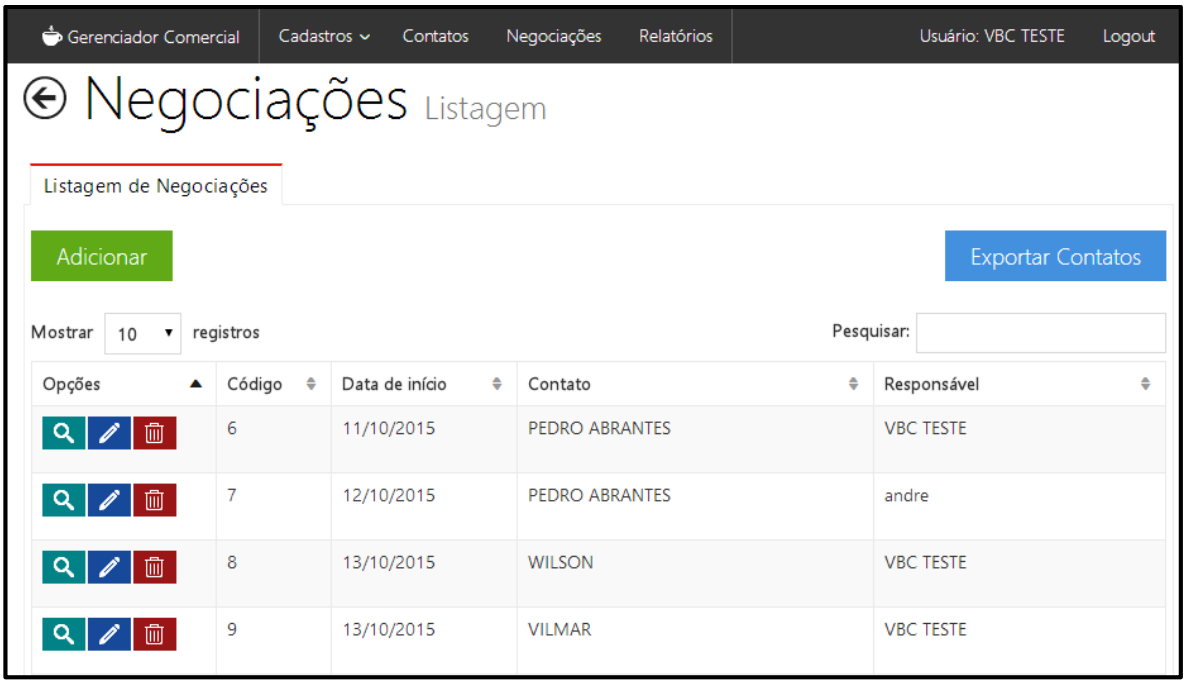
3.11 Exportação de dados com o sistema de suporte da VBC Automação Comercial LTDA

A fim de prover a exportação dos dados para o sistema de gerenciamento de suporte da VBC, o RA Online, foi feito um trabalho acima das informações cadastradas pelos usuários no sistema. Primeiramente, foi realizado o estudo do banco de dados utilizado pelo sistema de suporte, no intuito de se saber quais dados de clientes são necessários para o cadastro no sistema. Desse modo, definem-se as regras que prevalecem para esta operação.

Como é possível observar na Figura 32, é disponibilizado um botão na tela de listagem de negociações para a pesquisa de dados e seu posterior *download*. Os dados que serão exportados são correspondentes aos contatos cujas negociações estiverem com o *status* “Concluída com sucesso”. Ao clicar no botão, será gerado um arquivo do tipo CSV, que o

usuário deverá fazer o *download* e direcioná-lo para o setor responsável pelo sistema de suporte, para assim fazer a importação no sistema de suporte presente na VBC Automação.

Figura 32 - Tela Listagem de Negociações



Fonte: PRÓPRIA, 2015, utilizando Microsoft Edge 20.1.

Para o tratamento dos dados e a criação do arquivo de exportação, foram utilizadas as classes Java FileWriter, que tem a função de criar um arquivo em algum diretório do computador que executa a aplicação, e *File* que faz a manipulação dos arquivos criados entre as funções do programa. A Figura 33 mostra o trecho do código onde o arquivo de exportação é criado e manipulado.

Figura 33 - Trecho do código que gera o arquivo de exportação

```
76  try{
77      Path path = FileSystems.getDefault().getPath(caminho, nomeArquivo);
78      Files.deleteIfExists(path);
79      FileWriter q = new FileWriter(this.path+"br/com/gerenciador/temp/"+nomeArquivo, true);
80      String linha = "nome;fantasia;razao;tipoPessoa;cpfCnpj;tel;tel2;email;"
81              + "email2;cidade;estado;endereco;numero;bairro;complemento\n";
82      q.write(linha);
83      for(Contato contato : contatosList){
84          linha = "";
85          linha = contato.getNome()+";"+
86                  contato.getFantasia()+";"+
87                  contato.getRazao()+";"+
88                  contato.getTipoPessoa().getLabel()+";"+
89                  contato.getCpfCnpj()+";"+
90                  contato.getTel()+";"+
91                  contato.getTel2()+";"+
92                  contato.getEmail()+";"+
93                  contato.getEmail2()+";"+
94                  contato.getCidade().getNome()+";"+
95                  contato.getCidade().getUf()+";"+
96                  contato.getEndereco()+";"+
97                  contato.getNumero()+";"+
98                  contato.getBairro()+";"+
99                  contato.getComplemento()+"\n";
100         q.write(linha); // armazena o texto no objeto x, que aponta para o arquivo
101     }
102     q.close(); // cria o arquivo
103     return new File(this.path+"br/com/gerenciador/temp/"+nomeArquivo);
104 }
```

Fonte: PRÓPRIA, 2015, utilizando Eclipse Java EE IDE for Web Developers, versão Kepler Service Release 1.

A função mostrada na Figura 33 recebe a variável do tipo *List* contatosList, que contém uma lista de objetos do tipo Contato. Esta lista foi preenchida ao se fazer uma pesquisa no banco de dados retornando as negociações que foram concluídas e seus respectivos contatos. A partir da linha 83, cada posição da lista é aberta e seu registro é incorporado a uma *string* que posteriormente será escrita no arquivo de texto criado anteriormente pela classe FileWriter. Ao finalizar a escrita do arquivo, é retornado ao usuário pela classe *File*, que é exibido para o usuário no navegador, para então finalmente ser realizado o *download* do mesmo.

4 CONSIDERAÇÕES FINAIS

O presente trabalho foi desenvolvido em prol de atender ao problema de gerência e disponibilização de informações de vendas e clientes para equipe do setor comercial da VBC Automação Comercial. Anteriormente o controle de dados era feito com base em planilhas eletrônicas e agora, após a informatização do processo utilizando-se de tecnologias *web*, estão disponíveis aos usuários recursos de histórico das negociações em andamento e/ou concluídas. Tal sistema possibilita que as informações estejam disponíveis a qualquer momento e em qualquer dispositivo com acesso à internet, e permitir um número maior de usuários com acesso às informações ao mesmo tempo.

O desenvolvimento do sistema foi de suma importância, pois possibilitou a síntese de vários conceitos que foram trabalhados durante todos os quatro anos passados na universidade. O estudo e a aplicação de diversas tecnologias fez com que os conteúdos trabalhados no curso fossem utilizados. Também houve um aprendizado acerca de tecnologias ou tópicos que não foram muito abordados, como por exemplo, o uso de frameworks que auxiliam no desenvolvimento *web*.

Por se tratar de uma aplicação desenvolvida para o uso no ambiente *web* foi necessário o estudo aprofundado de tecnologias já estabelecidas no mercado como o HTML, CSS, JavaScript, assim como outras que não são tão difundidas como o VRaptor.

Um grande desafio no desenvolvimento do trabalho foi a necessidade de se estudar linguagens, sintaxes, *frameworks* e aplicações diferentes para que juntas, dessem forma ao sistema, fazendo com que a transição entre as tecnologias aplicadas fosse feita cuidadosamente, para que a aplicação fique suscetível a erros. O uso às diversas funções envolvendo a linguagem Javascript, por exemplo, se mostrou um desafio a parte, pois sua forma de trabalho pode variar bastante dependendo dos *frameworks* e bibliotecas utilizadas, em conjunto com o mesmo. Requisições feitas utilizando AJAX, por exemplo, exigem controles no lado do *Controller* e da *View*, exigindo do desenvolvedor atenção redobrada e cuidados especiais no desenvolvimento e nos testes, a fim de garantir funcionalidades íntegras, seguras e livres de erros que possam dificultar a usabilidade da aplicação desenvolvida.

Com o desenvolvimento do sistema, foi possível garantir que os objetivos específicos estabelecidos para o mesmo foram atendidos, de forma com que a relação de dados a diretores, gerentes, vendedores, possíveis clientes e negociações foi desenvolvida como a base do sistema. Estes registros são componentes para o alicerce do sistema, sendo

que somente após o relacionamento de todos os dados cadastrados, é possível haver o gerenciamento das informações e assim a obtenção de dados sobre as operações travadas em um determinado período de uso do sistema.

A possibilidade de análise dos históricos de venda e o acompanhamento das negociações são garantidos pelos cadastros de negociações e seus históricos e emissão de relatórios, garantindo assim maior segurança, já que os dados que antes estavam presentes em diversos arquivos de planilha, passam a ser armazenados no banco de dados do sistema.

Finalmente, com as informações processadas e analisadas, é possível então exportar os dados dos novos clientes adquiridos para o sistema RA VBC que impede a necessidade de retrabalho nos cadastros no sistema de RA Online e na troca de informação entre setores na VBC Automação LTDA.

Garantindo o bom funcionamento e integração entre os objetivos específicos citados na introdução deste documento, é possível então confirmar que o objetivo geral firmado, que é o desenvolvimento de uma ferramenta *web* que permita acompanhar e gerenciar as negociações de venda de produtos e ou serviços e a exportação destes dados para outros sistemas, foi completamente contemplado, fazendo com que sistema desenvolvido esteja pronto para implantação e uso dos usuários.

Em projetos futuros relacionados ao sistema, deverão ser aplicados testes em conjunto com o usuário final, em seu ambiente de produção, para que assim sejam feitos ajustes e se desenvolva novas funcionalidades para o uso no sistema, possibilitando assim o refinamento de construção. Como sistemas podem ser altamente voláteis, é de se esperar que novos requisitos apareçam conforme o mesmo passe a ser utilizado livremente no ambiente de produção. Também será pensado um processo mais automatizado para que seja feita a exportação de dados, já que a desenvolvida atualmente exige que a ação direta dos usuários, que podem causar algum erro por conta de operações incorretas. A forma atual de exportação de dados foi desenvolvida desta forma, pois há uma limitação na forma como o sistema de suporte da VBC trabalha, sendo assim necessário aguardar a autorização da empresa para que alguma melhoria seja feita.

Enfim, com a implantação, testes e validação do sistema em sua totalidade, pode-se então dar-se como concluído o desenvolvimento inicial do mesmo, restando assim a VBC Automação Comercial Ltda propor novas funcionalidades e melhorias que forem desejadas no sistema.

REFERÊNCIAS

CAELUM, **Desenvolvimento Ágil para Web 2.0 com VRaptor, Hibernate e AJAX**: Curso Fj-28. 2013a. Disponível em: <http://www.caelum.com.br/apostila-vraptor-hibernate/> Acesso em: 23 out. 2014.

CAELUM, **Java e Orientação a Objetos**: Curso Fj-11. 2013b. Disponível em:<http://www.caelum.com.br/apostila-java-orientacao-objetos/> Acesso em: 23 out. 2014.

CAELUM, **Java para desenvolvimento web**: Curso Fj-21. 2013c. Disponível em: <http://www.caelum.com.br/apostila-java-web/> Acesso em: 23 out. 2014.

CARVALHO, Fábio C. A. de**, Gestão do Conhecimento**. São Paulo: Pearson Education do Brasil, 2012.

COULOURIS, Georges. **Sistema distribuídos**: conceitos e projeto. Porto Alegre: Bookman, 2007.

DALL'OGLIO, Pablo. **PHP**: Programando com Orientação a objetos. São Paulo: NOVATEC, 2007.

ECMA Internacional. The JSON Data Interchange Format. Disponível em: <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>. Acessado em: 27 ago 2015.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de Banco de dados**. São Paulo: Pearson Addison Wesley, 2005.

FARIA, Thiago. **Java EE 7 com JSF, PrimeFaces e CDI**, 2015, p. 14-15. Disponível em: <http://cafe.algaworks.com/livro-java-ee-7-com-jsf-primefaces-e-cdi/> Acesso em: 08 ago. 2015.

GAMMA, Erich et al. **Padrões de Projeto**: Soluções reutilizáveis de software orientados a objetos. Porto Alegre: Bookman, 2000.

GOODMAN, Danny. **JavaScript**: A bíblia. Rio de Janeiro: Campus, 2001.

HIBERNATE, **Hibernate ORM**: Idiomatic persistence for Java and relational databases, 2014. Disponível em: <http://hibernate.org/orm/> Acesso em: 09 ago. 2015.

JASPERSOFT, **Getting Started with JasperReports Library**, 2015. Disponível em: <http://community.jaspersoft.com/wiki/getting-started-jasperreports-library> Acesso em 22 ago. 2015.

JQUERY, **What is jQuery?**, 2015. Disponível em: <https://jquery.com/> Acesso em: 09 ago. 2015.

LAKATOS, Eva Maria; MARCONI, Marina de Andrade. **Fundamentos de metodologia científica**. 5. ed. São Paulo: Atlas, 2003.

LAUDON, Jane P.; LAUDON, Kenneth C. **Sistemas de Informação Gerenciais.** São Paulo: Prentice Hall, 2007.

MULLER, Robert J. **Projeto de banco de dados**: usando UML para modelagem de dados. São Paulo: Berkeley Brasil, 2002.

NETBEANS, **The NetBeans E-commerce Tutorial**. 2014. Disponível em: <https://netbeans.org/images_www/articles/73/javaee/ecommerce/design/mvc-diagram.png> Acesso em 21 out. 2015.

ORACLE, **Your First Cup: An Introduction to the Java™ EE Platform,** 2012. Disponível em: <http://docs.oracle.com/javaee/6/firstcup/doc/docinfo.html> Acesso em: 08 ago. 2015.

PRESSMAN, Roger S; GRIESI, Ariovaldo. **Engenharia de software**. 7. São Paulo: Bookman, 2010.

RAGGETT, Dave et al. Ragggeet on HTML 4. Adisson-Wesley Professional, 1997.

RUMBAUGH, James et al. **Modelagem e projetos baseados em objetos com UML 2**. Rio de Janeiro: Campus, 2006.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Sistema de banco de dados**. 3. ed. São Paulo: Makron Books, 1999.

SORDI, José Osvaldo de; MARINHO, Bernadete de Lourdes. **Integração entre Sistemas**: Análise das Abordagens Praticadas pelas Corporações Brasileiras. 2007, p. 84. Disponível em: <www.spell.org.br/documentos/download/6498> Acesso em: 29 out. 2014.

SPERBER, Philip. **A Ciência das negociações**. Rio de Janeiro: Tecnoprint, 1982.

TABELESS, **Uma breve história do CSS**, 2006. Disponível em: <http://tableless.com.br/uma-breve-historia-do-css/> Acesso em: 28 de ago. 2015.

TANENBAUM, Andrew S. **Sistemas Distribuídos**: princípios e paradigmas. 2. ed. - São Paulo: Pearson Prentice Hall, 2008.

TAVARES, Clovis. **A mágica do marketing**: como obter lucros e dominar as cartas da negociação no novo milênio. 10. ed. São Paulo: Navegar, 2000.

VBC AUTOMAÇÃO, **Sobre a empresa**, 2014. Disponível em: <http://vbcautomacao.com.br/empresa.asp> Acesso em: 22 nov. 2014.

W3C, **HTML e CSS**. 2015a. Disponível em: <http://www.w3.org/standards/webdesign/htmlcss>. Acesso em: 09 mar. 2015.

W3C, **Javascript Web APIs**. 2015b. Disponível em: <http://www.w3.org/standards/webdesign/script> Acesso em: 02 set. 2015.