UNIVERSIDADE TECNOLOGICA FEDERAL DO PARANÁ
III CURSO DE ESPECIALIZAÇÃO EM TECNOLOGIA JAVA

ROBERTO ROSIN

SISTEMA PARA GERENCIAMENTO DE EMPRESA PRESTADORA DE SERVIÇOS

MONOGRAFIA DE ESPECIALIZAÇÃO

PATO BRANCO
2015

ROBERTO ROSIN

SISTEMA PARA GERENCIAMENTO DE EMPRESA PRESTADORA DE SERVIÇOS

Trabalho de Conclusão de Curso, apresentado ao III Curso de Especialização em Tecnologia Java, da Universidade Tecnológica Federal do Paraná, câmpus Pato Branco, como requisito parcial para a obtenção do título de Especialista.
Orientador: Prof. Vinicius Pegorini

PATO BRANCO

2015

SISTEMA PARA GERENCIAMENTO DE EMPRESA PRESTADORA DE SERVIÇO

Por

Roberto Rosin

Esta monografia foi apresentada às 15h00 do dia 04 de novembro de 2015 como requisito parcial para a obtenção do título de ESPECIALISTA, no III Curso de Especialização em Tecnologia Java, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. O acadêmico foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. Msc. Vinicius Pegorini
Orientador
UTFPR – Campus Pato Branco

Prof. Msc. Rubia Eliza de Oliveira Schulz Ascari
Banca
UTFPR – Campus Pato Branco

Profa. Msc. Soelaine Rodrigues Ascari
Banca
UTFPR – Campus Pato Branco

Prof. Msc. Robisen Cris Brito
Coordenador do curso de Especialização
UTFPR – Campus Pato Branco

RESUMO

ROSIN, Roberto Rosin. Sistema para gerenciamento de empresa prestadora de serviços. 2015.53f. Monografia (Trabalho de especialização) – Departamento Acadêmico de Informática, Universidade Tecnológica Federal do Paraná, Campus Pato Branco. Pato Branco, 2015.

O uso do computador para o gerenciamento dos principais processos de empresas tornou-se uma prática comum. Dessa maneira, este trabalho visa desenvolver um aplicativo para controle de uma empresa prestadora de serviços. A aplicação será desenvolvida para plataforma web, para diminuir os custos de manutenção da empresa. Para o desenvolvimento será empregado o Spring Framework, e a metodologia de desenvolvimento TDD. O trabalho foi desenvolvido pensando na criação de componentes, tanto no lado servidor quanto no lado cliente da aplicação, o que, apesar de um esforço inicial despendido, proporcionou uma grande agilidade no desenvolvimento. Como resultado final foi desenvolvido uma aplicação para o controle de uma empresa prestadora de serviços, permitindo o cadastro de clientes, operações financeiras e serviços.

Palavras-chave: Spring, Java para Web, Desenvolvimento Orientado a Testes

ABSTRACT

ROSIN, Roberto Rosin. System management for services provider companies. 2015.53f. Monograph (Working specialization) - Academic Department of Informatics, Federal Technological University of Paraná, Campus Pato Branco. Pato Branco, 2015.

The use of computers for the management of key processes of companies become a common practice. Thus, this work aims to develop an application for control of a services provider company. The application was developed for web platform to reduce the company's maintenance costs. For the development will be used the Spring Framework, and will be employee TDD development methodology. The work was developed thinking in the creation of components to accelerate the development, on the server side and on the implementation of the client side, which, despite an initial effort to create the components, provided a great agility in development. The result was an application to control a services provider company, enabling the customer management, financial transactions and services control.

Keywords: Spring, Java for web, Test Driven Development

LISTA DE SIGLAS E ABREVIATURAS

ACID – *Atomicity, Consistency, Isolation, Durability*
AOP - *Programação Orientada a Aspectos*
API – *Application Programming Interface*
CRUD - *Create, Retrieve, Update, Delete*
CSS – *Cascading Style Sheets*
EJB - *Enterprise Java Beans*
HTML – *HiperText Markup Language*
HTTP - *HyperText Transfer Protocol*
IBM - *International Business Machines*
IDE - *Integrated Development Environment*
Java EE - *Enterprise Edition*
Jaxb - *Java Architecture for XML Binding*
JCP – *Java Community Process*
JDBC – *Java Database Connectivity*
JDK – *Java Development Kit*
JDO – *Java Data Objects*
JMS – *Java Message Service*
JNDI - *Java Naming and Directory Interface*
JPA – *Java Persistence API*
JRE – *Java Runtime Environment*
JSON – *JavaScript Object Notation*
JSP – *Java Server Pages*
JSR – *Java Specification Requests*
JVM – *Java Virtual Machine*
MVC – *Model, View, Controller*
MVCC - *Multi-Version Concurrency Control*
ODBC - *Open DataBase Connectivity*
ORM – *Object Relational Mapping*
OXM – *Object XML Mapping*
PHP – *Hypertext Preprocessor*
POJOs - *Plain Old Java Object*
REST – *Representational State Transfer*
RF – *Requisitos Funcionais*
RNF – *Requisitos Não Funcionais*
SQL – *Structured Query Language*
TCL - *Tool Command Language*
TDD – *Test Driven Development*
UML – *Unified Modeling Language*
Unified EL – *Unified Expression Language*
URL - *Universal Resource Locator*
XML – *Extensible Markup Language*

LISTA DE FIGURAS

Figura 1 - Interface do Eclipse 21
Figura 2 - Interface do pgAdmin 21
Figura 3 - Interface do Visual Paradigm 23
Figura 4 - Overview do Spring Framework (SPRING, 2014) 24
Figura 5 - Modelagem do sistema ..... **Erro! Indicador não definido.**
Figura 6 - Diagrama de Casos de Uso 34
Figura 7- Tela de Login 36
Figura 8 - Tela Inicial do sistema 37
Figura 9 - Listagem de Clientes 38
Figura 10 - Aba Dados Gerais 38
Figura 11 - Aba Endereço 39
Figura 12 - Aba Contatos 39
Figura 13- Estrutura de Pacotes da aplicação 41
Figura 14 - Teste com falhas 47
Figura 15 - Teste Passando. 47
Figura 16 - Código renderizado pelo navegador 48

LISTA DE QUADROS

Quadro 1 - Tecnologias Utilizadas ..... **Erro! Indicador não definido.**
Quadro 2- Requisitos Funcionais 35
Quadro 3 - Requisitos Não Funcionais 35

LISTA DE CÓDIGOS FONTE

Listagem 1 – Código Ruby que faz a leitura de um nome e uma busca. 15
Listagem 2 – Código para leitura de um nome e uma pesquisa em janela. 15
Listagem 3 - Definindo o spring-boot como projeto pai 40
Listagem 4 - Declaração de dependências 40
Listagem 5 - Classe MainServer 41
Listagem 6 - CRUDController 42
Listagem 7 - Exibindo o arquivo index.jsp 42
Listagem 8 - ResponseBody 42
Listagem 9- FornecedorController 43
Listagem 10 - CRUDService 44
Listagem 11 - EntityToMapImpl 45
Listagem 12 - CidadeRepository 45
Listagem 13 - CRUDServiceTest 46
Listagem 14 - Tag CNPJ 48
Listagem 15 - Uso da tag cnpj 48
Listagem 16 - Tag grid 49
Listagem 17 - Código da classe CRUDService 50
Listagem 18 - Classe ColumnsFinder 50
Listagem 19 - Classe CreateJsonGrid 51
Listagem 20 - Classe FieldValueFinder 51

Sumário

1 INTRODUÇÃO 11
1.1 CONSIDERAÇÕES INICIAIS 11
1.2 OBJETIVOS 12
1.2.1 Objetivo Geral 12
1.2.2 Objetivos Específicos 12
1.3 JUSTIFICATIVA 12
1.4 ORGANIZAÇÃO DO TEXTO 13
2 REFERENCIAL TEÓRICO 14
2.1 JAVA 14
2.2 INVERSÃO DE CONTROLE E INJEÇÃO DE DEPENDÊNCIAS 15
2.3 TDD – TEST DRIVEN DEVELOPMENT 17
3 MATERIAIS E MÉTODO 19
3.1 MATERIAIS 19
3.1.1 Java JDK 20
3.1.2 Eclipse 20
3.1.2 pgAdmin 21
3.1.3 PostgreSQL 22
3.1.4 Visual Paradigm 22
3.1.5 Spring Framework 23
3.1.7 Ferramentas Complementares 29
3.2 MÉTODOS 29
4 RESULTADO 31
4.1 ESCOPO DO SISTEMA 31
4.2 MODELAGEM DO SISTEMA 32
4.3 APRESENTAÇÃO DO SISTEMA 36
4.4 IMPLEMENTAÇÃO DO SISTEMA 40
5 CONCLUSÃO 52
REFERENCIAS 53

1 INTRODUÇÃO

Este capítulo apresenta as considerações iniciais, com uma visão geral do assunto no qual o trabalho se insere, os objetivos, a justificativa e a organização do texto.

1.1 CONSIDERAÇÕES INICIAIS

O computador está cada vez mais presente nas empresas para gerenciar os processos que ocorrem em cada uma delas, desde uma simples pastelaria até uma grande distribuidora de alimentos, ou uma produtora de petróleo. Cada uma dessas empresas tem um processo específico e precisa de um sistema que atenda às suas necessidades.

Um sistema que permite o controle dos processos de uma pequena empresa prestadora de serviços é importante para permitir um melhor gerenciamento dos recursos financeiros e humanos. Permitindo assim ganho de produtividade e um melhor controle sobre os processos que ocorrem no dia-a-dia.

Com os serviços prestados armazenados em um banco de dados, pode-se ter um controle melhor sobre as atividades realizadas dentro do estabelecimento e também em cada cliente. Outro benefício que o armazenamento traz é um histórico das atividades realizadas pela empresa, sendo possível obter dados sobre como estava a atuação da empresa nos meses anteriores, ou até em anos anteriores.

O controle financeiro também é facilitado com um sistema computacional. As informações sobre os pagamentos efetuados ou não pelos clientes ficam armazenadas e podem ser consultadas facilmente à qualquer momento, auxiliando na possível necessidade de cobrança de algum documento que não foi pago, ou informando se determinado cliente está em dia com as parcelas. Também auxilia no momento de pagamento de contas, apresentando uma forma de consultar todos os vencimentos da empresa. Apresenta também uma forma de calcular o fluxo do caixa da empresa, que permite saber qual é a margem de lucro ou prejuízo que a empresa está tendo.

Os sistemas de informação, de uma forma geral auxiliam as empresas a controlarem seus processos, porém, algumas empresas não tem estrutura para ter um sistema rodando em um servidor dentro da mesma, sendo assim, um sistema que rode na web auxilia no controle dos processos da empresa, sem ser oneroso quanto ao armazenamento da aplicação.

Visando agregar inovação e otimizar processos dos estabelecimentos de prestação de serviços, este trabalho se propõe a desenvolver uma aplicação web que faça o controle financeiro e o controle de prestação de serviços da empresa, permitindo a emissão de relatórios sobre os serviços prestados, dos clientes inadimplentes e financeiros.

1.2 OBJETIVOS

O objetivo geral apresenta a finalidade principal da realização do trabalho realizado e os objetivos específicos complementam o resultado obtido com o objetivo geral.

1.2.1 Objetivo Geral

Desenvolver um sistema web com interface rica que possibilite o gerenciamento de recursos financeiros e de serviços prestados por uma empresa.

1.2.2 Objetivos Específicos

- Apresentar o *framework* Spring no desenvolvimento no lado servidor da aplicação;
- Apresentar o uso de padrões de projetos no desenvolvimento da aplicação web;
- Apresentar o uso da metodologia de desenvolvimento orientado a testes.

1.3 JUSTIFICATIVA

O sistema deverá ser desenvolvido para suprir a necessidade de uma empresa do ramo de prestação de serviços de refrigeração e rebobinagem de motores elétricos em gerenciar o setor financeiro da empresa e os serviços que são prestados por ela, bem como auxiliar no controle de produtos de terceiros que possam estar em posse da empresa para conserto.

Foi definido o uso da tecnologia Java, por estar consolidada no mercado, contendo uma comunidade forte de usuários, o que pode facilitar a busca por soluções de possíveis problemas que possam vir a ocorrer durante a codificação, e por ser o alvo principal dos estudos do curso ao qual se refere este trabalho.

Para auxiliar no desenvolvimento do projeto, será utilizado o *framework* Spring. Segundo a empresa ZeroTurnAround, que desenvolve sistemas para auxiliar o desenvolvimento de aplicações Java, cerca de 40% das aplicações Java para web utilizam o *framework* Spring. O *framework* possibilita a fácil criação de controladores para receber as requisições das páginas web, a fácil criação de repositórios para acesso aos dados da aplicação e injeção de dependências. Outra vantagem é que todas essas funcionalidades podem ser obtidas como poucas linhas de configuração.

O conceito de *Test Driven Development* (Desenvolvimento Orientado a Testes) TDD, juntamente com o *framework* JUnit, será utilizado por diminuir o número de erros que o sistema final pode apresentar, e por facilitar os testes unitários que devem ser aplicados no sistema. São utilizados conceitos de boas práticas de programação, código limpo e padrões de projetos para facilitar a manutenção do projeto e deixar o código mais legível, respeitando os conceitos de Orientação a Objetos.

1.4 ORGANIZAÇÃO DO TEXTO

Este texto está organizado em capítulos, dos quais este é o primeiro e apresenta a ideia e o contexto do sistema, incluindo os objetivos e a justificativa.

O Capítulo 2 apresenta o referencial teórico que fundamenta a proposta conceitual do sistema desenvolvido. Apresenta ainda a metodologia de desenvolvimento TDD.

O Capítulo 3 apresenta os materiais utilizados para o desenvolvimento do sistema, e o método que foi utilizado.

O Capitulo 4 apresenta o sistema que foi desenvolvido, apresentando exemplos da modelagem e da implementação. A modelagem é exemplificada através de documentos e diagramas. A implementação é exemplificada através da apresentação de partes do código fonte e de telas do sistema com descrição de suas funcionalidades.

No Capítulo 5 está a conclusão com as considerações finais.

2 REFERENCIAL TEÓRICO

Neste capitulo serão apresentados os referenciais teóricos sobre as principais tecnologias utilizadas no desenvolvimento deste trabalho. Este capitulo será dividido em sessões, sendo que cada uma irá corresponder à uma tecnologia utilizada.

2.1 JAVA

Em 1991 a Sun Microsystems lançou um projeto de pesquisa que resultou em uma linguagem baseada em C++ chamada Oak. Mais tarde, quando se descobriu que já existia uma linguagem de programação com esse nome, então foi sugerido o nome Java em uma cafeteria. Em 1993, quando a web explodiu a Sun viu uma oportunidade de lançar a sua linguagem e lançou os Applets, para proporcionar conteúdo dinâmico à web. O Java foi oficialmente apresentado em 1995, chamando atenção da comunidade por causa do enorme interesse na WEB (DEITEL; DEITEL, 2009).

Uma das principais características da linguagem Java é a portabilidade, um mesmo código Java pode ser executado em vários sistemas operacionais diferentes, isso é possível graças a JVM (*Java Virtual Machine*).

A plataforma Java EE (*Enterprise Edition*) é desenvolvida pela *Java Community Process* (JCP), que é responsável por todas as tecnologias Java. Para definir quais são as tecnologias que são implementadas pelo Java EE foram criadas as *Java Specification Requests* (JSRs). O trabalho da comunidade Java no âmbito JCP ajuda a garantir os padrões da tecnologia Java, estabilidade e compatibilidade entre plataformas (ORACLE, 2014).

A plataforma Java EE utiliza um modelo de programação simplificado, sendo opcional o uso de arquivos descritores XML (*Extensible Markup Language*), dando opção ao desenvolvedor para utilizar anotações diretamente no arquivo Java, e o servidor Java EE irá configurar o componente na implantação e em tempo de execução. Com anotações o desenvolvedor adiciona informações diretamente no elemento afetado, seja ele uma classe, um método ou um atributo (ORACLE, 2014).

No Java EE, a injeção de iependência pode ser aplicada a todos os recursos que precisam de um componente, efetivamente escondendo a criação e pesquisa de recursos a partir do código do aplicativo.

2.2 INVERSÃO DE CONTROLE E INJEÇÃO DE DEPENDÊNCIAS

Para Martin Fowler (FOWLER, 2005), “Inversão de Controle é um fenômeno comum quando estendemos *frameworks*, sendo muitas vezes comum que seja uma característica definidora de um Framework”.

O trecho de código da Listagem 1, escrito na linguagem Ruby, faz a leitura de um nome e de uma busca digitados através de linha de comando. Nessa interação, o código do programador é que está no controle, é ele que decide quando chamar os métodos *process_name* e *process_quest*.

```
#ruby
puts 'What is your name?'
name = gets
process_name(name)
puts 'What is your quest?'
quest = gets
process_quest(quest)
```

Listagem 1 – Código Ruby que faz a leitura de um nome e uma busca.

No entanto, se for utilizado um *framework* para criação de janelas, o código ficaria de acordo com a Listagem 2.

```
require 'tk'
root = TkRoot.new()
name_label = TkLabel.new() {text "What is Your Name?"}
name_label.pack
name = TkEntry.new(root).pack
name.bind("FocusOut") {process_name(name)}
quest_label = TkLabel.new() {text "What is Your Quest?"}
quest_label.pack
quest = TkEntry.new(root).pack
quest.bind("FocusOut") {process_quest(quest)}
Tk.mainloop()
```

Listagem 2 – Código para leitura de um nome e uma pesquisa em janela.

Há uma grande diferença no fluxo de controle entre esses dois trechos de código, em particular no controle de quando os métodos *process_name* e *process_quest* serão chamados. No exemplo utilizando linha de comando, o programador tem controle sobre

quando eles serão chamados. Utilizando um *framework* para criação de janelas o programador não tem controle sobre quando os métodos serão chamados, ele entrega esse controle para o *framework* de janelas, que é quem vai decidir quando chamar os métodos, com base nas ligações que foram feitas ao criar o formulário. Este fenômeno é chamado de Inversão de Controle.

Uma característica importante de um framework é o momento em que os métodos definidos pelo usuário para adequar a sua aplicação ao framework são chamados. Estes métodos geralmente são chamados a partir do próprio framework e não a partir do código do usuário. Isso faz com que o framework, muitas vezes, faça o papel de programa principal na coordenação das atividades da aplicação. Essa inversão no controle da aplicação dá aos frameworks o poder de servir como um esqueleto extensível, sendo apenas necessário que os métodos fornecidos pelo desenvolvedor se adaptem aos algoritmos genéricos definidos pelo framework.

Segundo Martin Fowler*, "inversão de controle é a chave que torna um framework diferente de uma biblioteca. Uma biblioteca é essencialmente um conjunto de funções que você pode chamar, cada chamada faz um trabalho e retorna para o cliente".* Diferente disso, um *framework* incorpora algum comportamento abstrato. Para usá-lo, o programador deve inserir o seu código em vários lugares, seja por subclasses ou por uma ligação com as classes do *framework*. É nesses pontos que o *framework* chama seu código (FOWLER, 2005).

Outra maneira de se fazer isso, é por meio da criação de eventos. O .NET é um exemplo disso. O programador pode declarar eventos em *widgets* e ligar um método por meio de um *delegate* (FOWLER, 2005).

A inversão de controle como apontada acima funciona bem em casos isolados, mas em alguns casos precisa-se fazer a chamada de vários métodos em um ponto de extensão. Para isso, o *framework* irá disponibilizar interfaces, que o programador deve implementar para estender o comportamento do *framework*. Um exemplo disso são os EJBs (*Enterprise Java Beans*), onde o programador pode implementar vários métodos que são chamados pelo container em diversos pontos do ciclo de vida (FOWLER, 2005).

Exemplos mais simples de inversão de controle podem ser encontrados no padrão *Template Method*, onde a superclasse define o fluxo de controle e as subclasses estendem essas classes, sobrescrevendo métodos ou implementando métodos abstratos para fazer a extensão. Assim também no *JUnit*, o código do *framework* chama os métodos

anotados com *@Before*, *@After* e *@Test*. O *framework* faz a chamada e o código reage, novamente o controle é invertido (FOWLER, 2005).

Segundo Martin Fowler (FOWLER, 2005), "há uma confusão sobre o significado de inversão de controle, devido ao aumento de contêineres IoC. Algumas pessoas confundem o princípio geral com os estilos específicos de inversão de controle (como injeção de dependência) que estes contêineres usam".

A questão é, que aspecto de controle esses contêineres estão invertendo? Em geral, eles procuram por uma implementação de um *plugin*. Eles asseguram que uma classe ou *plugin* siga uma convenção que permita um módulo montador, que injeta a implementação de um objeto em uma classe (FOWLER, 2004).

> Assim, eu acho que nós precisamos de um nome mais específico para este padrão. Inversão de Controle é um termo genérico demais, deixando as pessoas confusas. Como um resultado de muita discussão com vários defensores da IoC, nós estabelecemos o nome *Dependency Injection (Injeção de Dependências)*. (FOWLER, 2004).

Segundo Martin Fowler (FOWLER, 2004), "a ideia básica da Injeção de Dependência é ter um objeto que popula um campo em um objeto com uma implementação apropriada".

2.3 TDD – TEST DRIVEN DEVELOPMENT

TDD é uma da prática de desenvolvimento de software sugerida por diversas metodologias ágeis. Nela, o desenvolvedor escreve testes automatizados de maneira constante ao longo do desenvolvimento, muitas vezes, antes mesmo de ter uma implementação. Essa inversão no ciclo de desenvolvimento traz diversos benefícios para o projeto, como baterias de testes maiores, e maior qualidade no código, pois para escrever bons testes, o desenvolvedor é obrigado a fazer bom uso de orientação a objetos (ANICHE, 2014).

Para Aniche (2014), praticar TDD "nos ajuda a escrever um software melhor, com mais qualidade, e um código melhor, mais fácil de ser mantido e evoluído".

Para Beck (BECK, 2010), o TDD tem duas regras principais:

- Escreve-se código novo apenas se um teste automatizado falhou
- Elimina-se duplicação.

> Essas duas regras implicam em uma ordem para as tarefas de programação.
>
> 1. Vermelho – Escrever um pequeno teste que não funcione e que talvez nem mesmo compile inicialmente.
> 2. Verde – Fazer rapidamente o teste funcionar, mesmo contendo algum pecado necessário no processo.
> 3. Refatorar – Eliminar todas as duplicatas criadas apenas para que o teste funcione. (BECK, 2010).

Seguindo este conceito, o programador cria a classe de testes, cria o primeiro método de teste, e após isso começa a criar a estrutura que vai atender ao teste, como as classes e interfaces. Após criar as classes, o programador cria um código simples, apenas para fazer o teste passar, e depois, verifica se não existe duplicação no código. Em seguida escreve mais um teste, escreve o código para fazer este teste passar e em seguida refatora o código para remover duplicações. Este ciclo se repete até que o resultado esperado seja atendido.

3 MATERIAIS E MÉTODO

Neste capítulo estão descritas as tecnologias e ferramentas utilizadas no desenvolvimento do sistema.

A principal tecnologia utilizada no desenvolvimento do *backend*[1] do sistema foi a linguagem Java. No *frontend*[2] foram utilizadas as tecnologias HTML5 (*HiperText Markup Language*, versão 5), CSS3 (*Cascading Style Sheets*) e JQuery, além de PostgreSQL (*Structured Query Language*) como banco de dados. Para fornecer acesso às funcionalidades da linguagem Java foi utilizado o JDK 8 (*Java Development Kit*), e para a codificação do sistema foi utilizada a IDE Eclipse (*Integrated Development Environment*). Para acesso ao banco de dados foi utilizado o pgAdmin e para modelagem de dados o Visual Paradigm.

3.1 MATERIAIS

A seguir serão apresentadas as principais ferramentas utilizadas para o desenvolvimento do sistema.

Ferramenta/ Tecnologia	Versão	Referência	Finalidade
Visual Paradigm	12.2	http://www.visual-paradigm.com/	Documentação da modelagem baseada na UML (*Unified Modeling Language*).
Linguagem Java	JDK 1.8	http://www.oracle.com	Linguagem de programação.
Eclipse IDE	Luna RC3 Release (4.4.0RC3)	https://www.eclipse.org	Ambiente de desenvolvimento.
PostgreSql	9.2	http://www.postgresql.org/	Banco de dados.
PGAdmin III	1.16.1	http://www.pgadmin.org/	Administrador do banco de dados.
Spring Boot	1.1.8.RELEASE	http://projects.spring.io/spring-boot/	Framework para auxiliar na criação da aplicação.

[1] Servidor do sistema. Gerencia regras de negócio e provê a API de acesso a dados e serviços.
[2] Cliente do sistema. Trabalha com interação com o usuário. Provê as telas para consulta e entrada de dados.

Spring	4.2.0	http://projects.spring.io/spring-framework/	Framework para criação de aplicações web e Injeção de Dependências.
Spring Data JPA	1.8.2	http://projects.spring.io/spring-data-jpa/	Framework para persistência dos dados.
Spring Security	4.0.2	http://projects.spring.io/spring-security/	Framework para gerenciar a segurança.
Maven		https://maven.apache.org/	Ferramenta para gerenciamento das dependências do projeto.
Bootstrap	3.3.2-1	http://getbootstrap.com/	Framework para páginas web responsivas.
JQuery	2.1.3	http://jquery.com/	Framework javascript.
JQueryUI	1.11.3	http://jqueryui.com/	Biblioteca de componentes javascript para interfaces.
Datatables	1.10.4	http://www.datatables.net/	Plugin JQuery para tabelas dinâmicas.
Lombok	1.16.14	https://projectlombok.org/	Plugin para gerar os métodos Getters e Setters automaticamente no Eclipse.

Quadro 1 - Tecnologias Utilizadas

3.1.1 Java JDK

Para criar aplicações Java o desenvolvedor precisa do JDK. O JDK é um kit de desenvolvimento Java que possui o *Java Runtime Environment* (JRE), o compilador Java e as bibliotecas e APIs Java (*Application Programming Interface*).

3.1.2 Eclipse

O Eclipse é uma IDE para desenvolvimento de softwares em várias linguagens como Java, C, C++ e PHP (*Hypertext Preprocessor*). Também fornece *plug-ins* para modelagem de dados e geração de relatórios, embora menos utilizados na comunidade. Foi criado pela IBM (*International Business Machines*) e teve seu código aberto para a comunidade em 2001. O Eclipse é mantido pela Eclipse Foundation, uma organização independente e sem fins lucrativos. A interface do Eclipse está ilustrada na Figura 1.

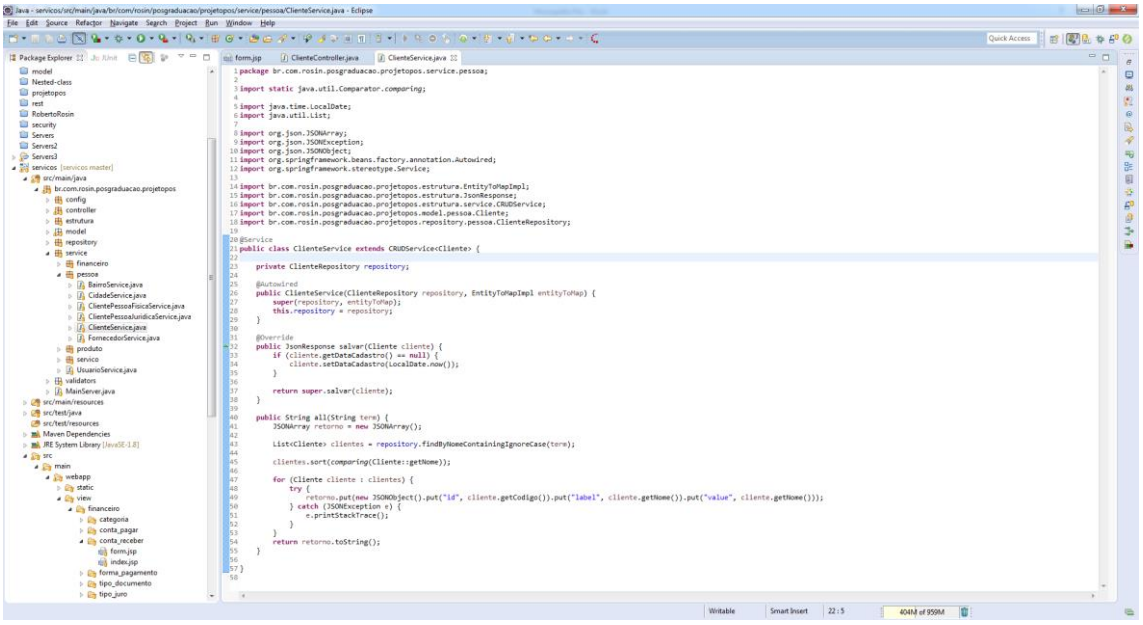

Figura 1 - Interface do Eclipse

3.1.2 pgAdmin

O pgAdmin é uma ferramenta de administração de banco de dados *Open Source* para o PostgreSQL. No aplicativo é possível executar consultas SQL, gerenciar dados das tabelas, gerenciar tabelas, *triggers*, funções e bases de dados do PostgreSQL. A interface do pgAdmin está ilustrada na Figura 2.

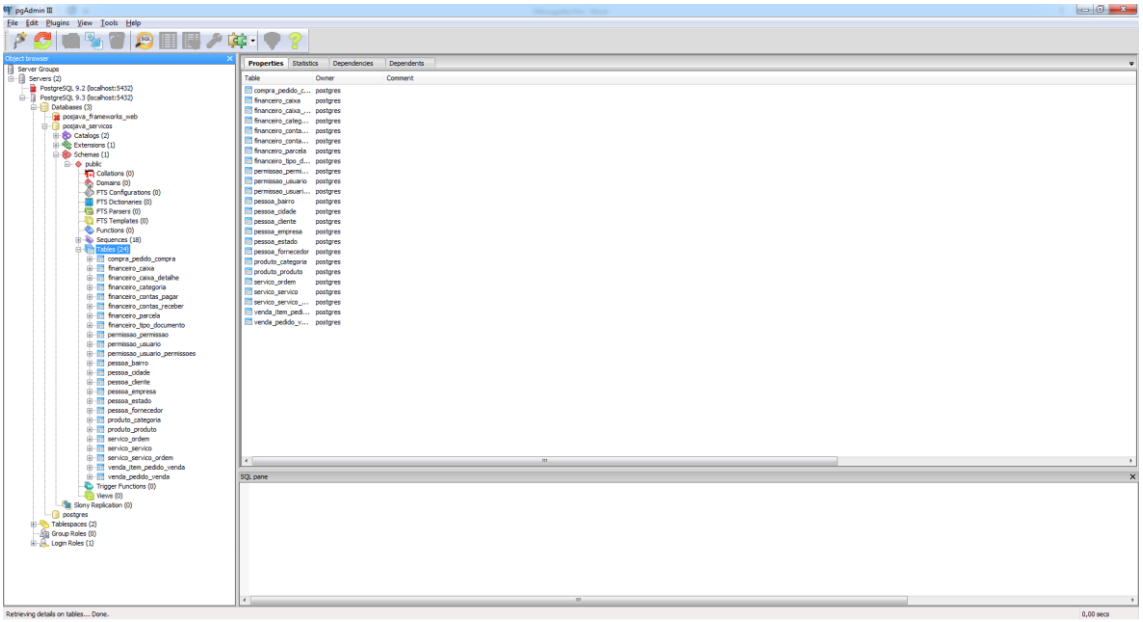

Figura 2 - Interface do pgAdmin

3.1.3 PostgreSQL

O PostgreSQL é um sistema de banco de dados objeto-relacional *Open Source*. Tem mais de 15 anos de desenvolvimento ativo e uma arquitetura comprovada que ganhou uma forte reputação de confiabilidade, integridade de dados e correção. Ele é executado em todos os principais sistemas operacionais, incluindo Linux, UNIX e Windows. É totalmente compatível com ACID (*Atomicity, Consistency, Isolation, Durability*), tem suporte completo para chaves estrangeiras, *views*, triggers e procedimentos armazenados. Inclui os tipos de dados do SQL: 2008, incluindo INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL E TIMESTAMP. Suporta armazenamento de grandes objetos binários, incluindo imagens, sons ou vídeo. Ele tem interfaces de programação nativas para C/C++, Java, .Net, Perl, Python, Ruby, Tcl (*Tool Command Language*), ODBC (*Open DataBase Connectivity*) entre outros (POSTGRE, 2015).

Um banco de dados de classe empresarial, o PostgreSQL possui recursos sofisticados, como *Multi-Version Concurrency Control* (MVCC), ponto de recuperação, *tablespaces*, replicação assíncrona, transações aninhadas (*savepoints*), backups online, um sofisticado planejador/otimizador de consultas e um sistema de tolerância a falhas. É altamente escalável. Existem sistemas PostgreSQL ativos em ambientes de produção que gerenciam 4 terrabytes de dados (POSTGRE, 2015).

3.1.4 Visual Paradigm

O Visual Paradigm é uma ferramenta para representação de UML. Permite a criação de vários diagramas propostos na UML, como o Diagrama de Classes, Diagrama de Casos de Uso, Diagrama de Sequencia entre outros. A interface do Visual Paradigm pode ser vista na Figura 3 (VISUAL-PARADIGM, 2015).

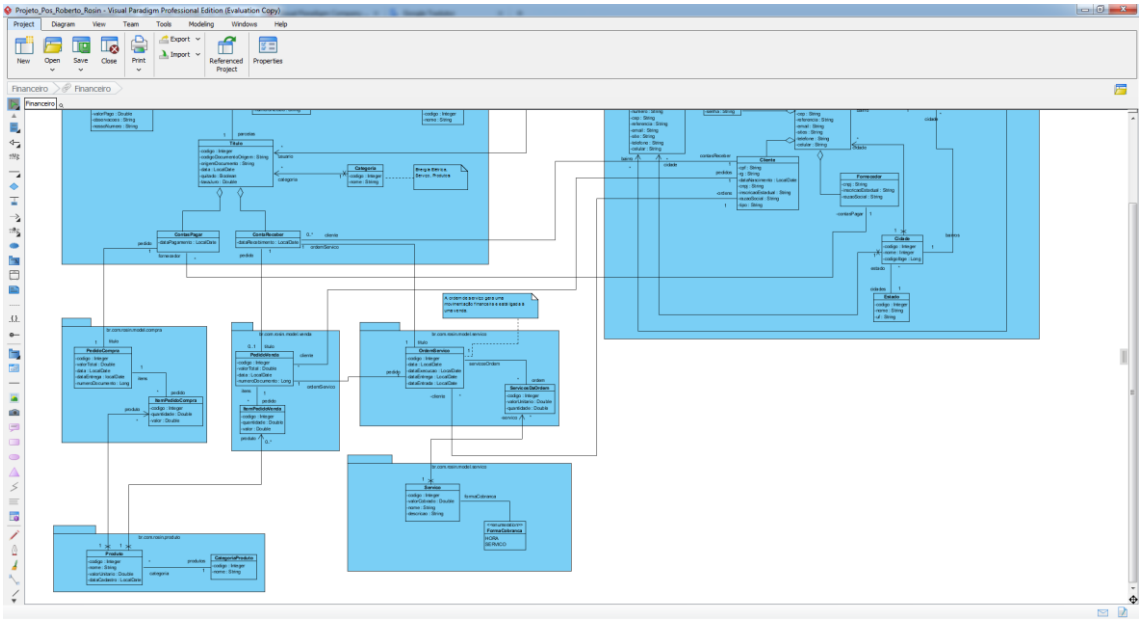

Figura 3 - Interface do Visual Paradigm

3.1.5 Spring Framework

O Spring Framework é uma solução leve e pronta para a construção de aplicativos corporativos que fornece o suporte de infraestrutura abrangente e modular para o desenvolvimento de aplicações Java. Sua modularidade permite que sejam utilizados apenas os módulos necessários, sem necessidade de carregar os demais componentes do *framework*. É projetado para ser não-intrusivo, deixando sua lógica de negócio livre das dependências do *framework*, porém, em sua camada de integração (como a camada de acesso a dados), podem existir algumas dependências com bibliotecas Spring ou com a tecnologia de acesso a dados, mas esses códigos são fáceis de serem isolados na sua base de código. O Spring lida com a infraestrutura para que o desenvolvedor possa se concentrar na aplicação.

O Spring foi desenvolvido para ser uma alternativa ao Java EE, quando a especificação Java EE era muito burocrática, tirando proveito disso para se torar o framework para a linguagem Java mais utilizado nos dias de hoje. Como seu objetivo principal é ser uma alternativa ao Java EE, foram criados em torno de 20 módulos agrupados em: Core, Container, Data Access/Integration, Web, AOP (Programação Orientada a Aspecto), Instrumentation, Messaging e Test, como mostrado na Figura 4.

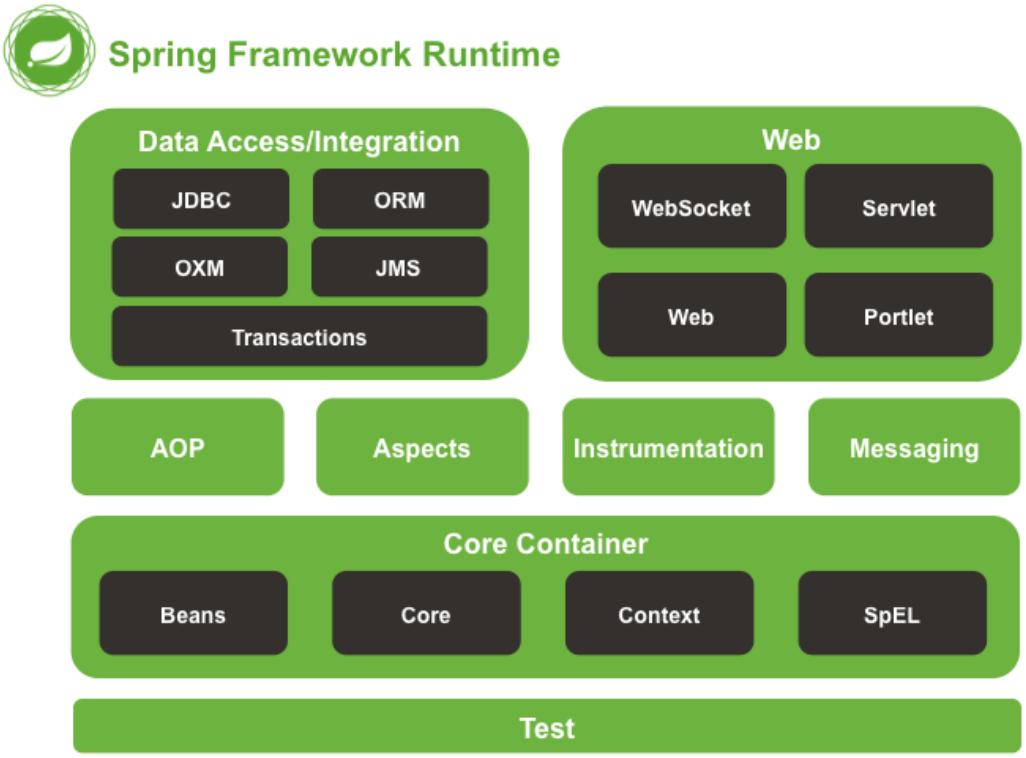

Figura 4 - Overview do Spring Framework (SPRING, 2014)

3.1.5.1 Spring Core Container

Segundo a documentação online do Spring Framework, "o *core container* é composto pelos módulos spring-core, spring-beans, spring-context, spring-context-support e *spring-expression* (*Spring Expression Language*)" (SPRING, 2014).

Os módulos *spring-core* e *spring-beans* proveem partes importantes do *framework*, como IoC e Injeção de Dependências. O Bean Factory é uma implementação sofisticada do padrão *Factory*, eliminando a necessidade da programação de *Singletons* e permitindo separar as configurações da lógica do programa.

O módulo *spring-context* é construído sobre a base dos módulos *spring-core* e *spring-beans*, e fornece uma maneira de acessar objetos de forma semelhante ao JNDI (*Java Naming and Directory Interface*). Adicionando a estas características outras herdadas do módulo *spring-beans* provê suporte à internacionalização, propagação de

eventos, carregamento de recursos e criação transparente de contexto. Tem como ponto principal a interface *ApplicationContext*.

Como consta na documentação online do Spring (2014) Framework, "o módulo *spring-context-support* provê suporte para a integração de bibliotecas de terceiros para cache (EhCache, Guava, JCache), envio de e-mails (JavaMail), agendamento (CommonJ, Quartz) e *template engines* (FreeMarker, JasperReposrts, Velocity)".

O módulo *spring-expression* é uma extensão da *Unified Expression Language (Unified EL)*. Ele provê uma poderosa ferramenta para manipulação de objetos no cliente JSP (*Java Server Pages*) da aplicação (SPRING, 2014).

3.1.5.2 Spring AOP e Spring Instrumentation

O módulo *spring-aop* provê suporte para Programação Orientada a Aspectos (AOP) e o módulo *spring-aspects* provê integração com o AspectJ.

O módulo *spring-instrument* fornece suporte a instrumentação de classe e implementações de *classloader* para serem usados em certos servidores de aplicativos. O módulo *spring-instrument-tomcat* contém agentes de instrumentação do Spring para Tomcat.

3.1.5.3 Spring Messaging

O módulo *spring-messaging* possui abstrações-chave para o projeto *Spring Integration*, como *Message*, *MessageChannel*, *MessageHandler* e outros, que servem de base para projetos baseados em mensagens. O módulo também inclui um conjunto de anotações para mapeamento, similar ao Spring MVC (*Model, View, Controller*) (SPRING, 2014).

3.1.5.4 Spring Data Access/Integration

A camada responsável por acesso a dados / integração consiste nos módulos JDBC (*Java Database Connectivity*), ORM (*Object Relational Mapping*), OXM (*Object XML Mapping*), JMS (*Java Message Service*), e módulos de transação.

O módulo *spring-jdbc* elimina a necessidade de código JDBC provendo uma abstração da camada JDBC e auxilia na análise de códigos de erros específicos do banco de dados

O módulo *spring-ctx* fornece o gerenciamento de transações para classes que implementem interfaces especiais e para todos os POJOs (*Plain Old Java Object*) (SPRING, 2014).

O módulo *spring-orm* fornece funcionalidades para integração com *frameworks* de Mapeamento Objeto Relacional como JPA (*Java Persistence API*), JDO (*Java Data Objects*) e Hibernate.

O módulo *spring-oxm* provê uma camada de abstração que suporta implementações de Mapeamento Objeto/XML, como *JAXB*, *Castor*, *XMLBeans*, *JBX* e *XStream* (SPRING, 2014).

O módulo *spring-jms* contém recursos para produzir e consumir mensagens. A partir da versão 4.1, provê integração com o módulo *spring-messaging* (SPRING, 2014).

3.1.5.5 Spring Web

A camada *web* do *framework* Spring consiste nos módulos *spring-web*, *spring-webmvc*, *spring-websocket* e *spring-webmvc-portlet.*

O módulo *spring-web* é responsável por fornecer os recursos de integração básica com a web, como upload de arquivos, e por iniciar o container IoC com a utilização de *Servlet Listeners*.

O módulo *spring-webmvc* provê as características do padrão MVC e a criação de serviços REST (Representational State Transfer). Fornece uma separação clara entre a lógica de negócios e o código de domínio da aplicação, e integra-se com todos os outros recursos do Spring Framework.

O módulo *spring-webmvc-portlet* fornece a implementação MVC para ser utilizado num ambiente de *Portlet* e espelha a funcionalidade do módulo de *spring-webmvc* (SPRING, 2014).

3.1.5.6 Spring Test

O modulo *spring-test* fornece funcionalidades para realização de testes unitários e testes de integração dos componentes Spring com JUnit ou TestNG. Fornece uma carga consistente do ApplicationContext para testes de aplicação e também objetos *mock* para testar seu código isoladamente (SPRING, 2014).

3.1.5.7 Spring IoC Container

O container IoC do Spring Framework é responsável pelo processo de Injeção de Dependências, que é o processo pelo qual a instância de uma dependência do objeto é definida, sendo por meio de argumentos no construtor, argumentos para um método fábrica, ou propriedades que são definidas na instância do objeto após a sua construção.

Os pacotes *org.springframework.beans* e *org.springframework.context* são a base para o contêiner de IoC. A interface *BeanFactory* provê um avançado mecanismo de configuração possibilitando o gerenciamento de qualquer tipo de objeto. A interface *ApplicationContext* é uma sub-interface de *BeanFactory*. Ela provê integração com as funcionalidades do Spring AOP, como recursos de mensagens para internacionalização, publicação de eventos e contextos específicos para a camada de aplicação, como o *WebApplicationContext* para uso em aplicações web (SPRING, 2014).

A interface *BeanFactory* provê a configuração do framework e algumas funcionalidades básicas, e a interface *ApplicationContext* adiciona funcionalidades específicas para empresas.

No Spring, os objetos que formam a espinha dorsal da sua aplicação e que são gerenciadas pelo contêiner do Spring IoC são chamados de *beans*. Um *bean* é um objeto que é instanciado, montado e gerenciado pelo contêiner IoC do Spring. Um *bean* é somente mais um entre muitos objetos de sua aplicação (SPRING, 2014).

A interface responsável pelo contêiner IoC do Spring é a *org.springframework.context.ApplicationContext*. As implementações dessa classe são responsáveis por instanciar, configurar e montar os beans. O contêiner do Spring busca

os beans baseado em metadados, que podem estar representados em arquivos XML, anotações ou configurados em códigos Java.

3.1.7 Ferramentas Complementares

Além dos programas citados, foi também utilizado um navegador web para realizar testes na aplicação. A aplicação foi testada nos navegadores Chrome[3], Mozila Firefox[4] e Opera[5].

3.2 MÉTODOS

Antes de iniciar o desenvolvimento do software, foi realizada uma pesquisa com as principais tecnologias que seriam utilizadas, visando entender o funcionamento e quais funcionalidades são disponibilizadas pelas mesmas. Após um estudo breve sobre as tecnologias, foi realizada uma breve análise e montado um diagrama para representar as classes de modelo do sistema. Com o diagrama pronto foi iniciada a codificação.

O método adotado para o desenvolvimento engloba as fases de análise, desenvolvimento e testes.

a) Análise: para realizar a análise do projeto, foram utilizados conhecimentos do autor deste trabalho, adquiridos ao observar o funcionamento de uma oficina de rebobinagem de motores elétricos, além de entrevistas informais com o proprietário.

A partir desses conhecimentos foi criado um Diagrama de Classes da UML para representar as classes de entidade do sistema e para servir de base para o desenvolvimento.

Restavam ainda algumas dúvidas sobre o funcionamento de algumas funções do sistema, para as quais foram realizadas reuniões com o orientador deste trabalho para serem discutidas.

b)Desenvolvimento (codificação): após a análise foi iniciada a codificação do sistema. A linguagem utilizada para codificação do servidor da aplicação foi Java, e no cliente foram utilizadas tecnologias como HTML5, CSS3 e JQuery. Durante alguns momentos da codificação, quando oportuno,

[3] Disponível em: https://www.google.com.br/chrome/browser/desktop/
[4] Disponível em: https://www.mozilla.org/pt-BR/firefox/new/
[5] Disponível em: http://www.opera.com/pt-br

foi utilizada a metodologia de Desenvolvimento Orientado a Testes (TDD – Test Driven Development) Kent Beck (BEKC, 2003), em que são desenvolvidos testes unitários antes da codificação.

c) Testes: os testes foram realizados pelo autor do trabalho a fim de garantir que as funcionalidades apresentadas não apresentem erros de codificação.

4 RESULTADO

Este capítulo apresenta o sistema que foi desenvolvido para plataforma web que é o resultado deste trabalho. Inicialmente são apresentados o sistema e uma descrição do mesmo. Em seguida, será detalhada a parte realizada na análise. Por fim, será apresentada a codificação do sistema e exemplificado o uso das tecnologias utilizadas no desenvolvimento.

4.1 ESCOPO DO SISTEMA

O aplicativo desenvolvido é um sistema computacional para controle de uma empresa prestadora de serviços de rebobinagem de motores e consertos de aparelhos elétricos. O sistema permite o cadastro de usuários, clientes, fornecedores, produtos, criação de ordens de serviços, vinculação de ordens de serviço com contas a receber, contas a pagar, e baixas de contas a receber e a pagar.

O sistema compreende um aplicativo para gerenciar alguns aspectos de uma empresa prestadora de serviços de manutenção. No sistema podem ser informados os clientes para os quais os serviços serão prestados. Podem ser também informados produtos que, posteriormente, podem ser vinculados a um serviço prestado por meio de uma Ordem de Serviço, além de ter um cadastro para informar quais são os serviços prestados e qual é o valor padrão cobrado pelo serviço. A ordem de serviço será sempre para um cliente, e deverá ter um serviços vinculados à ela, bem como poderá ter produtos. Uma ordem de serviço gera uma Conta a Receber, que pode ter várias parcelas. Cada parcela de conta a receber poderá ter várias baixas, representando os pagamentos efetuados pelo cliente.

O sistema permite também a manutenção de Contas a Pagar, que podem estar relacionadas a entrada de algum produto no sistema ou a eventos que estão fora do escopo do sistema, como contas de água, telefone e energia elétrica.

4.2 MODELAGEM DO SISTEMA

A Figura 5 apresenta o Diagrama de Classes criado para dar base ao desenvolvimento do sistema. Ele representa as entidades do sistema, que são as classes que representam as tabelas do banco de dados, também conhecidas como classes de modelo. O diagrama foi dividido conforme a divisão de pacotes da aplicação.

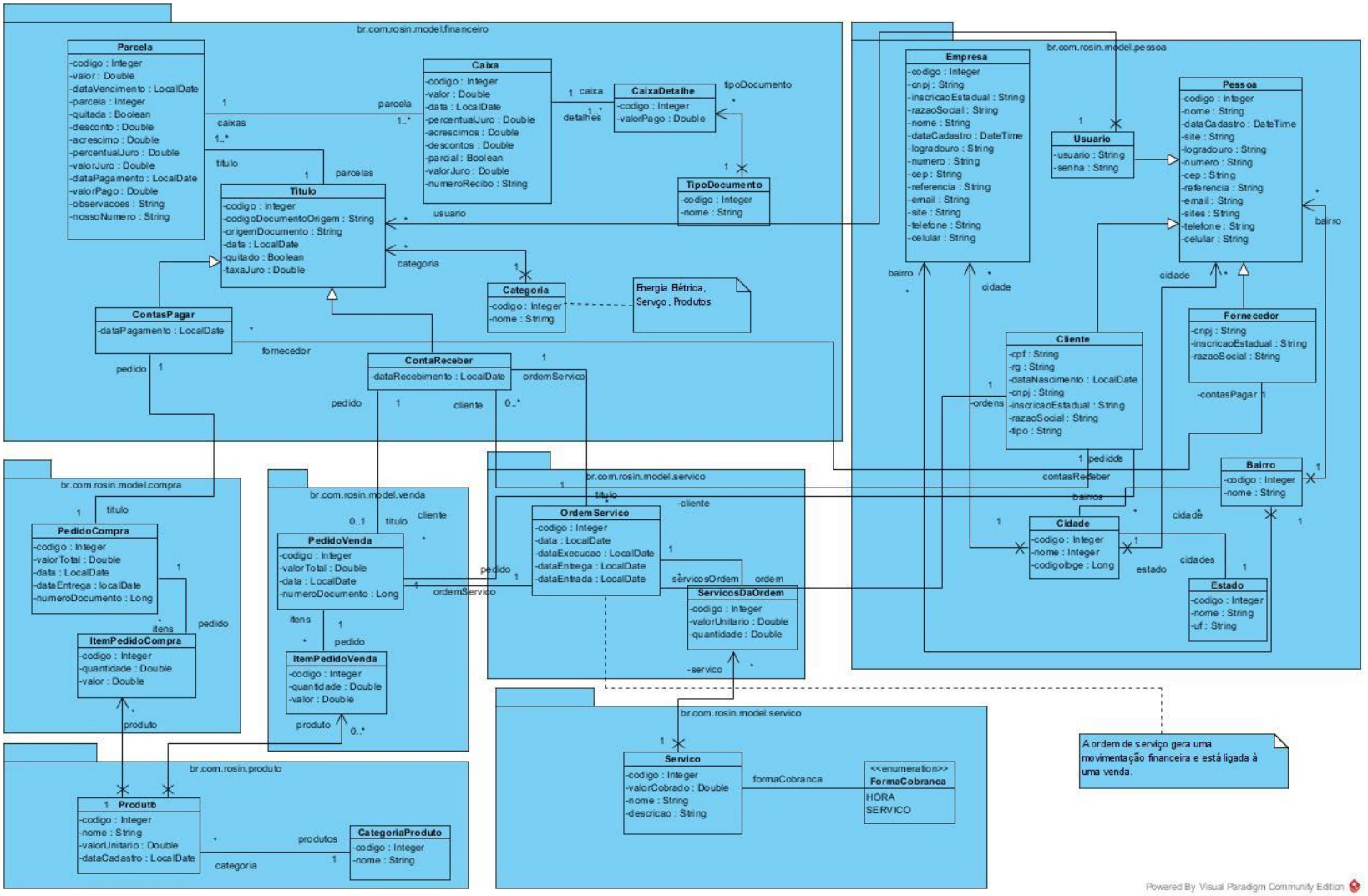

Figura 5- Modelagem do Sistema

A Figura 6 apresenta o diagrama dos casos de uso do sistema, considerando o usuário como ator do sistema, possuindo acesso à todas as operações.

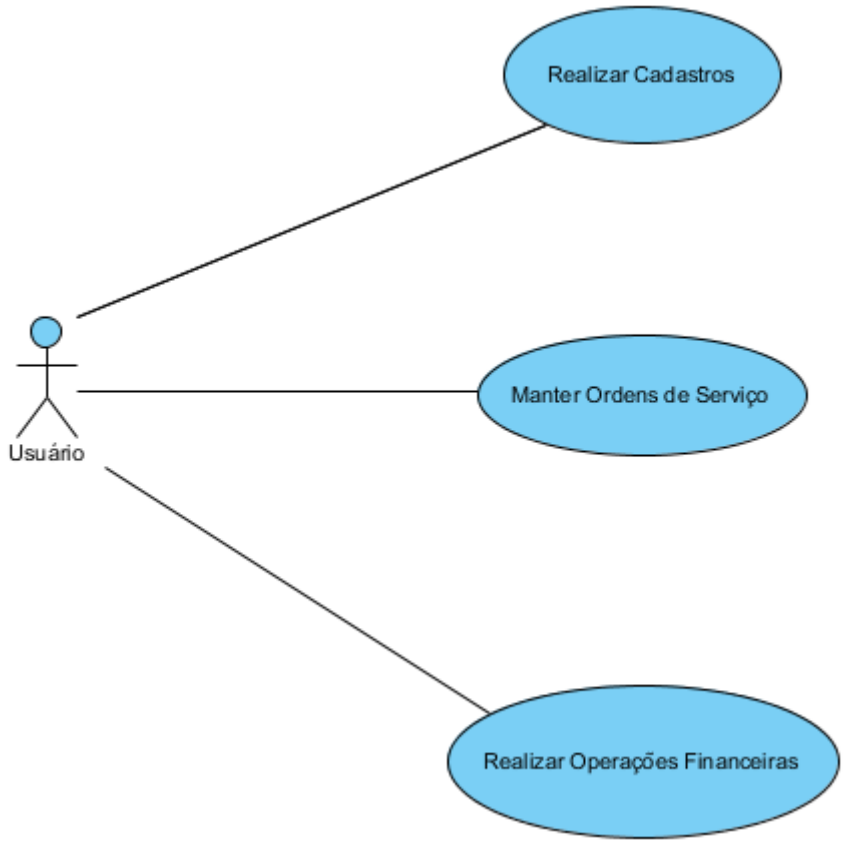

Figura 6 - Diagrama de Casos de Uso

Com base no diagrama de casos de uso apresentado na Figura 6, foram criados os Requisitos Funcionais (RF) e Requisitos Não Funcionais (RNF), mostrados nos Quadros 2 e 3, respectivamente.

	Caso de Uso	**Requisito**	**Descrição**
RF01	Realizar Cadastros	Cadastro de Clientes	O sistema deve permitir o cadastro de clientes, validando os tipos de campos, e campos obrigatórios.
FR02	Realizar Cadastros	Cadastros de Fornecedores	O sistema deve permitir o cadastro de fornecedores, validando os tipos dos campos e os campos obrigatórios
RF03	Realizar Cadastros	Cadastro de Categoria de Produtos	O sistema deve permitir o cadastro de categorias para classificar os produtos
RF04	Realizar Cadastros	Cadastro de Produtos	O sistema deve permitir o cadastro de produtos que podem ser utilizados em uma Ordem de Serviço
RF05	Realizar Cadastros	Categorias do Título	O sistema deve permitir o cadastro de Categorias para classificar as operações financeiras

RF06	Realizar Cadastros	Tipos de Documento	O sistema deve permitir o cadastro de tipos de documento para identificarem a forma de pagamento de um título
RF07	Realizar Cadastros	Serviços	O sistema deve permitir o cadastro dos serviços que serão realizados e informados na Ordem de Serviço
RF08	Ordem de Serviço	Ordem de Serviço	O sistema deve permitir o cadastro de Ordens de Serviço
RF09	Operações Financeiras	Contas a Pagar	O sistema deve permitir o lançamento de Contas a Pagar
RF10	Operações Financeiras	Contas a Receber	O sistema deve permitir o lançamento de Contas a Receber
RF11	Operações Financeiras	Baixa de Contas a Pagar	O sistema deve permitir a baixa das Contas a Pagar
RF12	Operações Financeiras	Baixa de Contas a Receber	O sistema deve permitir a baixa das Contas a Receber

Quadro 2- Requisitos Funcionais

	Requisito	**Descrição**
RNF01	Cadastro de Clientes	O campo CPF é obrigatório e deve ser válido e único para cada cliente do tipo Pessoa Física.
RNF02	Cadastro de Clientes	O campo RG é obrigatório e deve ser único para o cliente do tipo Pessoa Física
RNF03	Cadastro de Clientes	O campo CNPJ é obrigatório e deve ser válido e único para cada cliente do tipo Pessoa Jurídica
RNF04	Cadastro de Fornecedores	O campo CNPJ é obrigatório e deve ser válido e único para cada Fornecedor
RNF05	Ordem de Serviço	É obrigatória a inclusão de serviços na Ordem de Serviço
RNF06	Ordem Serviço	Após o fechamento da Ordem de Serviço, a mesma não poderá mais ser alterada ou excluída
RNF07	Operações Financeiras	Após ser realizada a baixa de uma parcela, uma conta a pagar ou a receber não poderá mais ser alterada ou excluída

Quadro 3 - Requisitos Não Funcionais

4.3 APRESENTAÇÃO DO SISTEMA

O sistema possui controle de usuários, sendo que somente alguém com um usuário e senha pode acessar o sistema. Após acessar o sistema não existem restrições de acesso às funcionalidades do sistema. A Figura 7 apresenta a tela de login do sistema.

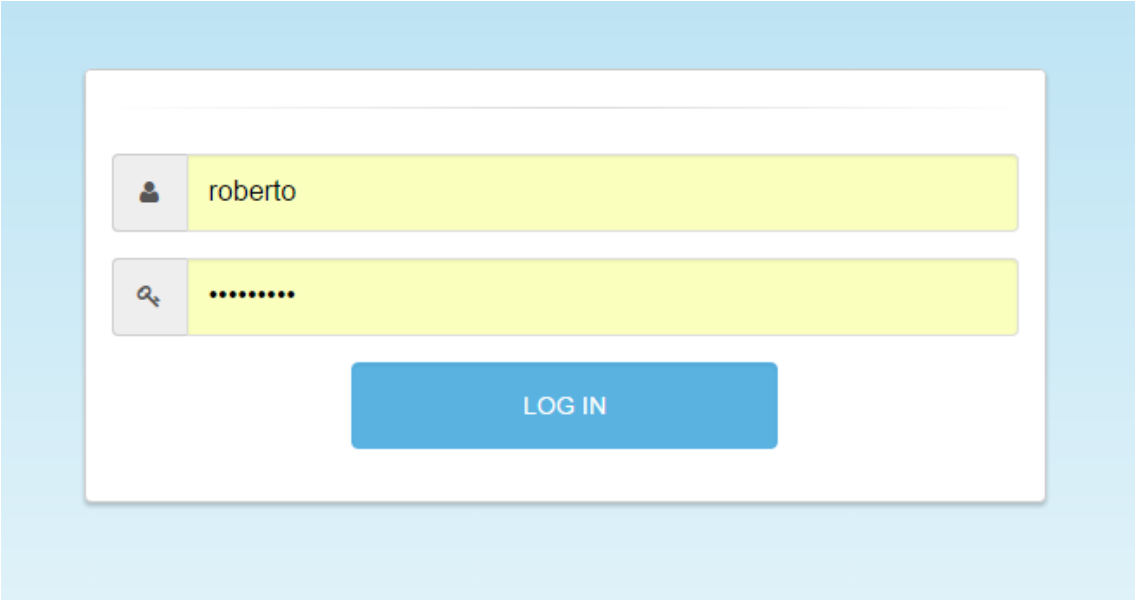

Figura 7- Tela de Login

Após informar o usuário e a senha, o usuário é redirecionado para a página principal do sistema, ou para a página que estava tentando acessar antes de ser redirecionado para a tela de login. A página inicial apresenta gráficos que são úteis para o gerenciamento da empresa, como as entradas e saídas de caixa dos últimos 10 dias, o saldo do caixa diário no mesmo período, contas a pagar e contas a receber. A tela inicial é apresentada na Figura 8.

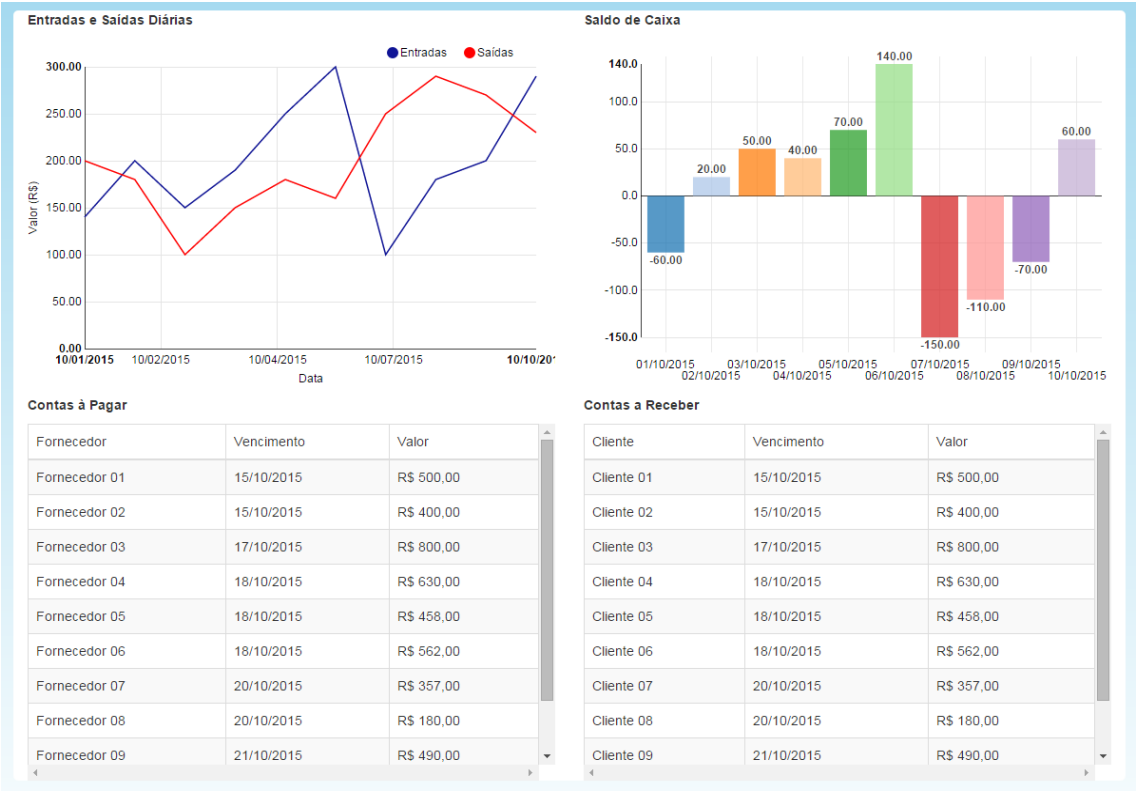

Figura 8 - Tela Inicial do sistema

Por meio do menu, o usuário pode ter acesso a todas as operações do sistema, como cadastro de Clientes, Fornecedores, Produtos, criação de Ordem de Serviço e de Títulos financeiros. As telas do sistema seguem um mesmo padrão, as quais são compostas por uma listagem dos dados já cadastrados no sistema, e a possibilidade de alteração, exclusão ou inclusão de registros. A Figura 9 apresenta a listagem de clientes do tipo Pessoa Física, na qual é possível visualizar todos os clientes cadastrados no sistema e também acessar a funcionalidade de cadastrar um novo cliente. A função de pesquisar por todos os campos da listagem também está presente nessa tela.

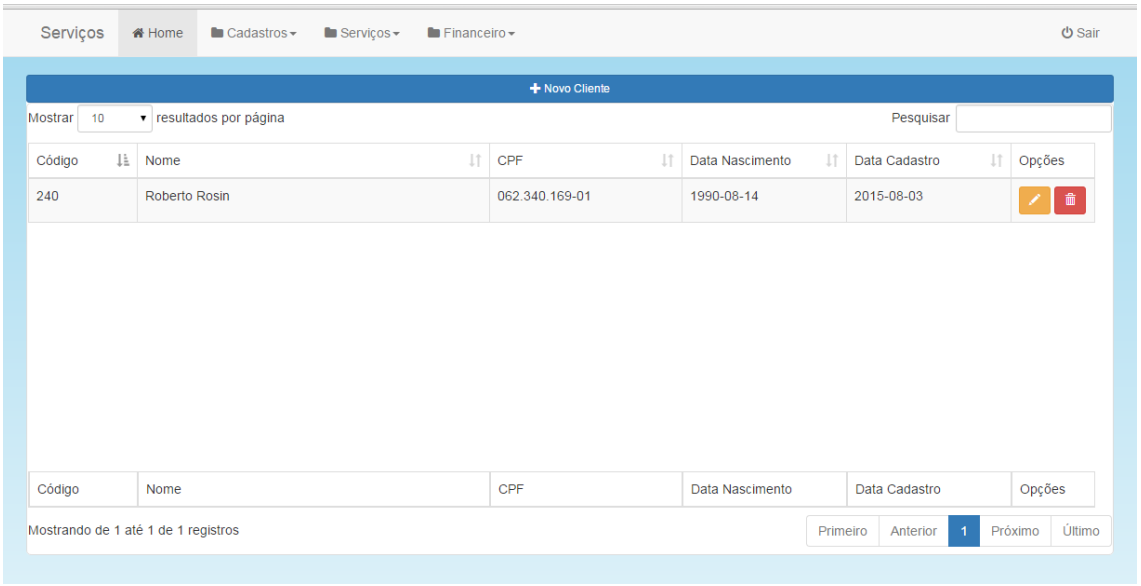

Figura 9 - Listagem de Clientes

Ao clicar no botão *Novo Cliente* é apresentada uma tela modal para que o usuário digite os dados do novo cliente, com possibilidade de salvar ou fechar a janela. A tela de cadastro de clientes é dividida em 3 abas. Na aba Dados Gerais, são apresentados os campos Nome, CPF, RG e Data de Nascimento, como mostrado na Figura 10. Na aba Endereço são apresentados os campos referentes ao endereço do cliente, como mostrado na Figura 11. Na aba Contato, são apresentados os campos de contato, como mostrado na Figura 12.

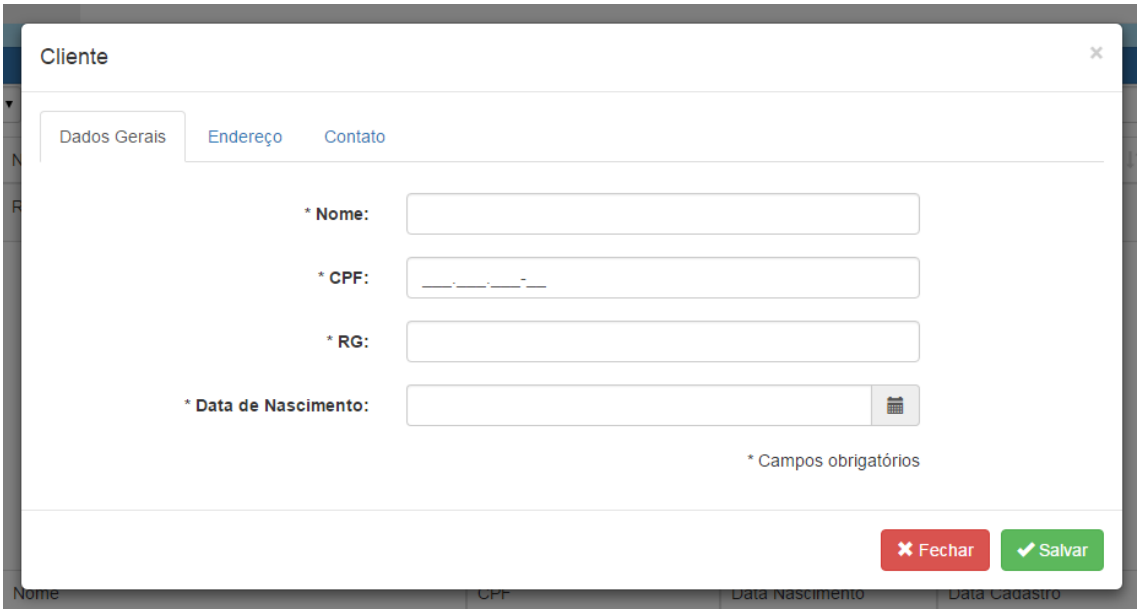

Figura 10 - Aba Dados Gerais

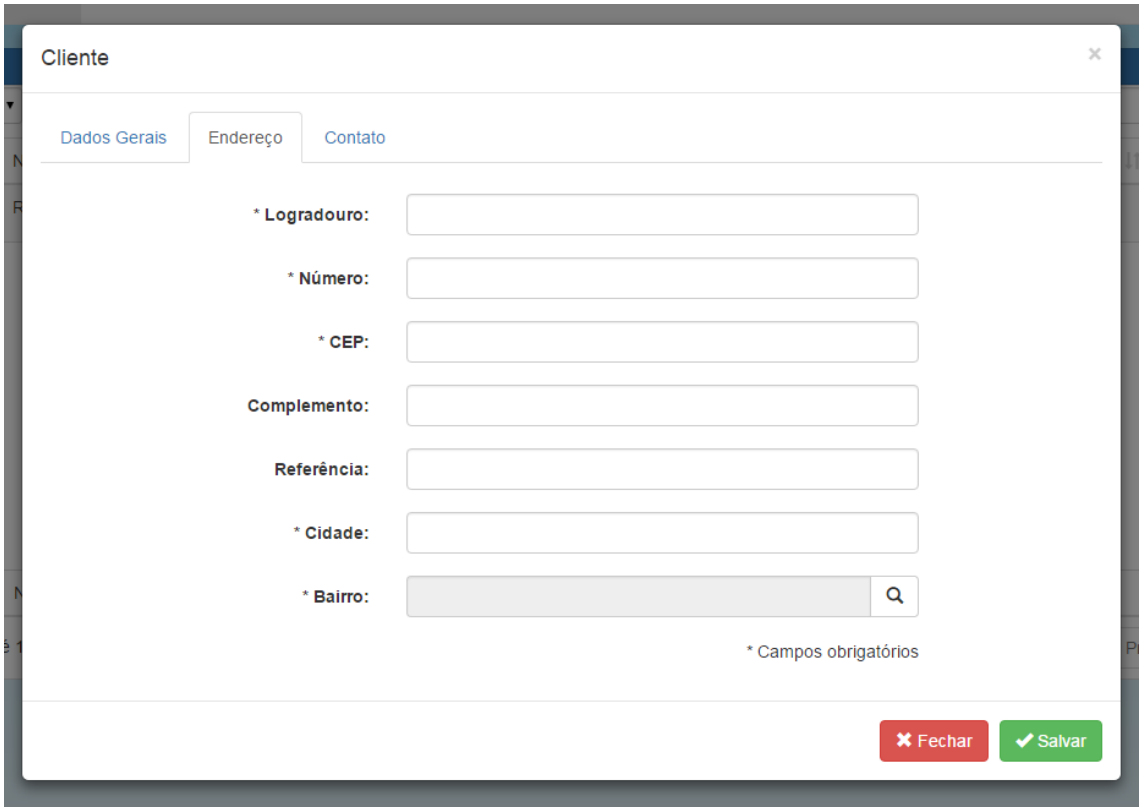

Figura 11 - Aba Endereço

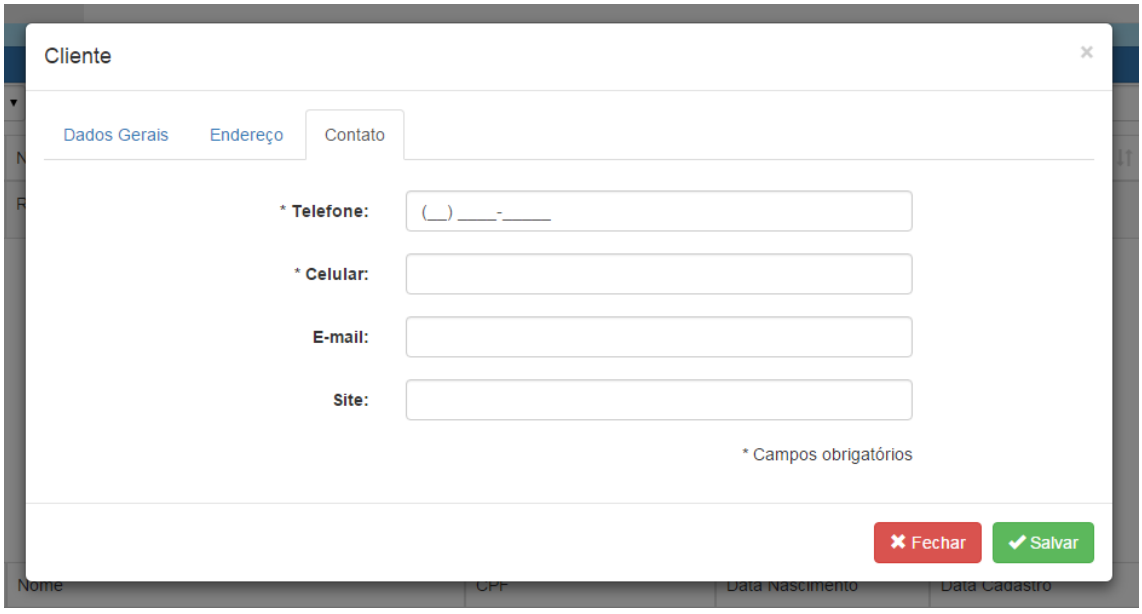

Figura 12 - Aba Contatos

As demais telas seguem o mesmo padrão de listagem e uma janela para inserção de dados. Para a construção das telas e do *layout* foi utilizado o *framework* Bootstrap, que provê funcionalidades para construção de *layouts* responsivos. A validação de dados obrigatórios é feita através do *framework* Parsley, que provê funcionalidades para validação de dados obrigatórios. Quando um campo obrigatório não for preenchido, é

mostrado abaixo do campo uma mensagem informando que o mesmo é obrigatório e o campo fica vermelho, para indicar que existe um erro no campo.

4.4 IMPLEMENTAÇÃO DO SISTEMA

O projeto utiliza a plataforma Maven para fazer o gerenciamento de dependências, sendo necessária a configuração do arquivo pom.xml. Nesse arquivo ficam todas as bibliotecas que o projeto necessita. Para a criação de um projeto baseado no *framework* Spring-Boot, é necessário configurar um projeto pai no arquivo pom.xml, como mostrado na Listagem 3.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.1.8.RELEASE</version>
</parent>
```

Listagem 3 - Definindo o spring-boot como projeto pai

Após adicionar o Spring-Boot como projeto pai, são adicionadas as dependências, elas devem ficar dentro da *tag* <*dependencies*>, como mostrado na Listagem 4.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
```

Listagem 4 - Declaração de dependências

O Spring-Boot espera que seja criada uma classe com o método *main* para ser a classe de start da aplicação. Essa classe também serve como configuração para informar para o Spring quais são os pacotes que serão percorridos para encontrar os objetos para a Injeção de Dependências. Todos os pacotes que contêm classes que serão injetadas, devem ficar em pacotes abaixo dessa classe, como na Figura 13. Na Listagem 5 é apresentada a classe *MainServer*.

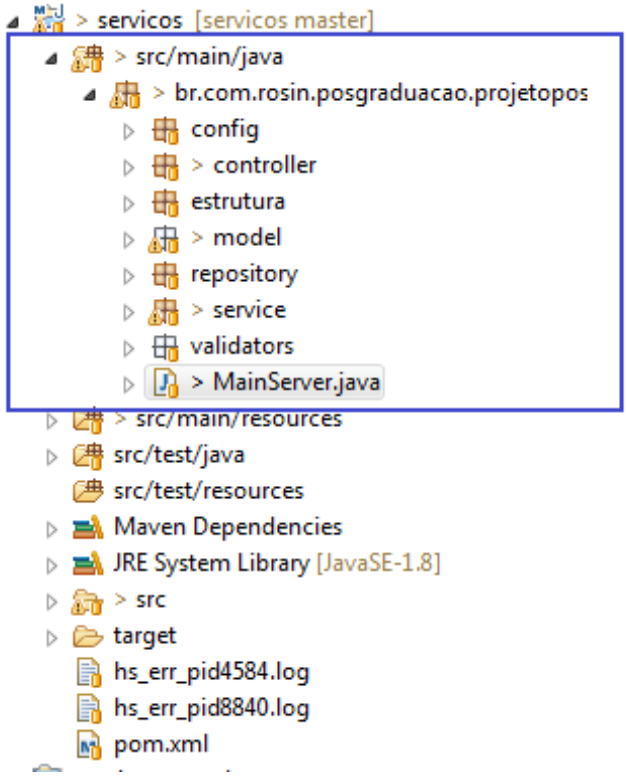

Figura 13- Estrutura de Pacotes da aplicação

```java
package br.com.rosin.posgraduacao.projetopos;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.context.web.SpringBootServletInitializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import br.com.rosin.posgraduacao.projetopos.config.WebSecurityConfig;

@Configuration
@EnableAutoConfiguration
@ComponentScan
public class MainServer extends SpringBootServletInitializer {

    public static void main(String[] args) {
        SpringApplication.run(MainServer.class, args);
    }

    @Bean
    public WebSecurityConfigurerAdapter webSecurityConfigurerAdapter() {
        return new WebSecurityConfig();
    }
}

```

Listagem 5 - Classe MainServer

As requisições feitas ao servidor são tratadas pelos *Controllers*, definidos no pacote *controller*. Para facilitar o reaproveitamento do código e dar agilidade ao desenvolvimento, foi criado um *controller* genérico, chamado CRUDController, que implementa as 4 operações de CRUD. Esse componente foi implementado em uma classe abstrata, obrigando a extensão para a criação de um *controller* para cada novo cadastro do sistema. A Listagem 6 apresenta um trecho do código do CRUDController.

```
13
14 public abstract class CRUDController<T> {
15
16     protected final CRUDService<T> service;
17     private final String initFormParam;
18
19     public CRUDController(CRUDService<T> service, String initFormParam) {
20         this.service = service;
21         this.initFormParam = initFormParam;
22     }
23
24     @RequestMapping(value="/", method=RequestMethod.GET)
25     public String home() {
26         return initFormParam + "/index";
27     }
28
29     @RequestMapping("/novo")
30     public String novo() {
31         return initFormParam + "/form";
32     }
33
34     @ResponseBody
35     @RequestMapping(value="/{codigo}", method=RequestMethod.GET, produces="application/json; charset=UTF-8")
36     public JsonResponse editar(@PathVariable Integer codigo) {
37         return service.getOne(codigo);
38     }
39
```

Listagem 6 - CRUDController

A anotação *@RequestMapping* informa para o Spring qual é o método HTTP e qual URL que o método anotado responde. Na Listagem 7, o código apresentado responde para a URL "/" (*Universal Resource Locator*) quando o método HTTP (*HyperText Transfer Protocol*) for do tipo GET. Ele retorna um texto, que informa qual é o nome do arquivo jsp que será apresentado, um arquivo dentro do diretório recebido no construtor com o nome index.jsp.

```
23
24     @RequestMapping(value="/", method=RequestMethod.GET)
25     public String home() {
26         return initFormParam + "/index";
27     }
```

Listagem 7 - Exibindo o arquivo index.jsp

A anotação *@ResponseBody* é utilizada quando o retorno do método não é uma *view*, como por exemplo um JSON (*JavaScript Object Notation*) ou um XML. O tipo de retorno do método é definido na anotação *@RequestMapping*, no parâmetro *produces*, como apresentado na Listagem 8. A anotação *@PathVariable* recebe o parâmetro que é passado através da URL, no código apresentado na Listagem 8 é recebido o parâmetro *{código}* do tipo *Integer*.

```
33
34     @ResponseBody
35     @RequestMapping(value="/{codigo}", method=RequestMethod.GET, produces="application/json; charset=UTF-8")
36     public JsonResponse editar(@PathVariable Integer codigo) {
37         return service.getOne(codigo);
38     }
39
```

Listagem 8 - ResponseBody

Na Listagem 9 é apresentado um exemplo de um *controller* que herda da classe CRUDController. Nessa classe é definida qual é a rota principal do *controller* com a anotação *@RequestMapping*. O código apresentado é do *controller* de fornecedores, então a rota principal é *fornecedor*. A anotação *@Autowired* é a anotação que provê injeção de dependências. Quando o *controller* for instanciado, o construtor será chamado pelo container IoC do Spring, que será encarregado de montar os parâmetros que o construtor espera.

```
12
13 @Controller
14 @RequestMapping("/fornecedor")
15 public class FornecedorController extends CRUDController<Fornecedor> {
16
17     private FornecedorService service;
18
19     @Autowired
20     public FornecedorController(FornecedorService service) {
21         super(service, "pessoa/fornecedor");
22         this.service = service;
23     }
24
25     @ResponseBody
26     @RequestMapping(value="/", method=RequestMethod.GET, produces="application/json; charset=UTF-8")
27     public String all(String term) {
28         return service.all(term);
29     }
30 }
```

Listagem 9- FornecedorController

O parâmetro esperado no Construtor é uma implementação da classe CRUDService. A classe CRUDService é uma classe que fornece serviços para o *controller*. Todas as regras de negócio são feitas em classes *Service*. Para definir um *service*, utilizamos a anotação @Service. Parte da classe CRUDService é apresentada na Listagem 10.

```java
public abstract class CRUDService<T> {

    protected final JpaRepository<T, Integer> repository;
    protected final EntityToMap entityToMap;

    public CRUDService(JpaRepository<T, Integer> repository, EntityToMap entityToMap) {
        this.repository = repository;
        this.entityToMap = entityToMap;
    }

    public JSONObject listarDadosGrid(String columns) {
        try {
            CreateJsonGrid<T> c = new CreateJsonGrid<>(repository.findAll(), ColumnsFinder.getMapColunas(columns));
            return c.montarJSON();
        } catch (JSONException | NoSuchFieldException | SecurityException | IllegalAccessException
                | IllegalArgumentException | InvocationTargetException | NoSuchMethodException e) {
            e.printStackTrace();
            JSONObject resposta = new JSONObject();
            try {
                resposta.put("SUCCESS", false).put("mensagem", e.getMessage());
            } catch (JSONException je) {
                je.printStackTrace();
            }
            return resposta;
        }
    }
```

Listagem 10 - CRUDService

O método apresentado na Listagem 10, *listarDadosGrid* é utilizado para listar os dados para os grids de dados nas telas. O método recebe um parâmetro que informa quais são as colunas apresentadas na tela. Cada coluna apresentada tem o nome de uma propriedade de uma entidade, que é referenciada com *Generics* passado como parâmetro na instanciação da classe. O método utiliza ainda uma implementação da classe *EntityToMap* para transformar os dados da entidade em um mapa de propriedades.

Na Listagem 11 é apresentado parte do código da classe *EntityToMapImpl*, que implementa a interface *EntityToMap*. Essa classe usa *Reflection* para buscar os atributos da classe e seus respectivos valores. A anotação *@Component* informa ao Spring que esta classe é uma classe que pode ser injetada através da anotação *@Autowired*.

```
18
19 @Component
20 public class EntityToMapImpl implements EntityToMap {
21
22     @Override
23     public Map<String, Object> toMap(Object registro) {
24         Map<String, Object> map = new HashMap<>();
25         Class<?> c = registro.getClass();
26
27         while (c != Object.class) {
28             for (Field field : c.getDeclaredFields()) {
29                 Object value = getFieldValue(field, c, registro);
30                 if (value != null) {
31                     if (value.getClass().equals(LocalDate.class)) {
32                         LocalDate date = (LocalDate) value;
33                         map.put(field.getName(), date.format(DateTimeFormatter.ofPattern("dd/MM/yyyy")));
34                     } else if (value.getClass().equals(LocalDateTime.class)) {
35                         LocalDateTime date = (LocalDateTime) value;
36                         map.put(field.getName(), date.format(DateTimeFormatter.ofPattern("dd/MM/yyyy hh:mm")));
37                     } else {
38                         map.put(field.getName(), value);
39                     }
40                 }
41             }
42
43             c = c.getSuperclass();
44         }
45
46         return map;
47     }
```

Listagem 11 - EntityToMapImpl

Na Listagem 10, a classe CRUDService também recebe no seu construtor, um parâmetro do tipo *JPARepository*. *JPARepository* é uma implementação do Spring para a camada de acesso a dados. Na Listagem 12 é apresentada a classe *CidadeRepository*. Essa classe provê acesso aos dados das cidades no banco de dados. Apenas criando uma interface que implementa a interface *JPARepository* e passando através de *Generics* a entidade e o tipo da chave primária, ela fornece acesso as operações CRUD e alguns métodos de pesquisa.

Para métodos de pesquisa específicos de uma classe, podem ser criados métodos com os nomes dos parâmetros que serão filtrados e o tipo do filtro. No exemplo da Listagem 12, a assinatura do método é *findByNomeContainingIgnoreCase(String nome)*. Esse método irá filtrar todas as cidades cujo nome contenha o nome que foi passado por parâmetro, ignorando maiúsculas e minúsculas.

```
 9 public interface CidadeRepository extends JpaRepository<Cidade, Integer> {
10
11     List<Cidade> findByNomeContainingIgnoreCase(String nome);
12 }
13
```

Listagem 12 - CidadeRepository

Para garantir a qualidade do sistema, foram implementados testes unitários utilizando JUnit. A Listagem 13 apresenta um exemplo de teste unitário utilizando os

frameworks JUnit e Mockito. Na primeira linha pode-se ver a anotação *@RunWith*. Essa anotação inicializa os objetos anotados com *@Mock*. Os métodos anotados com *@Before* são executados antes de cada método de teste, que são anotados com *@Test*. Existem também as anotações *@BeforClass*, que são executadas uma vez antes da classe, *@After*, que executa após cada método de teste e *@AfterClass* que executa sempre após a execução de todos os métodos.

```
22 @RunWith(MockitoJUnitRunner.class)
23 public class CRUDServiceTest {
24
25     private CRUDService<Cliente> service;
26
27     @Mock
28     private JpaRepository<Cliente, Integer> repository;
29
30     @Before
31     public void setup() {
32         this.service = new TestedCRUDService(this.repository);
33     }
34
35     @Test
36     public void deveExecutarOMetodoSalvar() {
37         Cliente cliente = new Cliente();
38         JsonResponse response = service.salvar(cliente);
39
40         Mockito.verify(repository).save(cliente);
41         assertEquals("SUCCESS", response.getStatus());
42     }
43
44     @Test
45     public void deveRetornarMapComPropriedadesEntidadeVazia() {
46         Mockito.when(repository.findOne(1)).thenReturn(new Cliente());
47         JsonResponse response = service.getOne(1);
48
49         assertEquals(new HashMap<String, Object>(), response.getResult());
50     }
```

Listagem 13 - CRUDServiceTest

Os métodos da classe *org.junit.Assert* são os métodos que fazem a validação dos testes. De acordo com o resultado desses métodos o *plugin* do JUnit para o Eclipse informa o resultado dos testes, pintando uma barra com vermelho ou verde, de acordo com o resultado do teste, como é apresentado nas Figura 14 e 15.

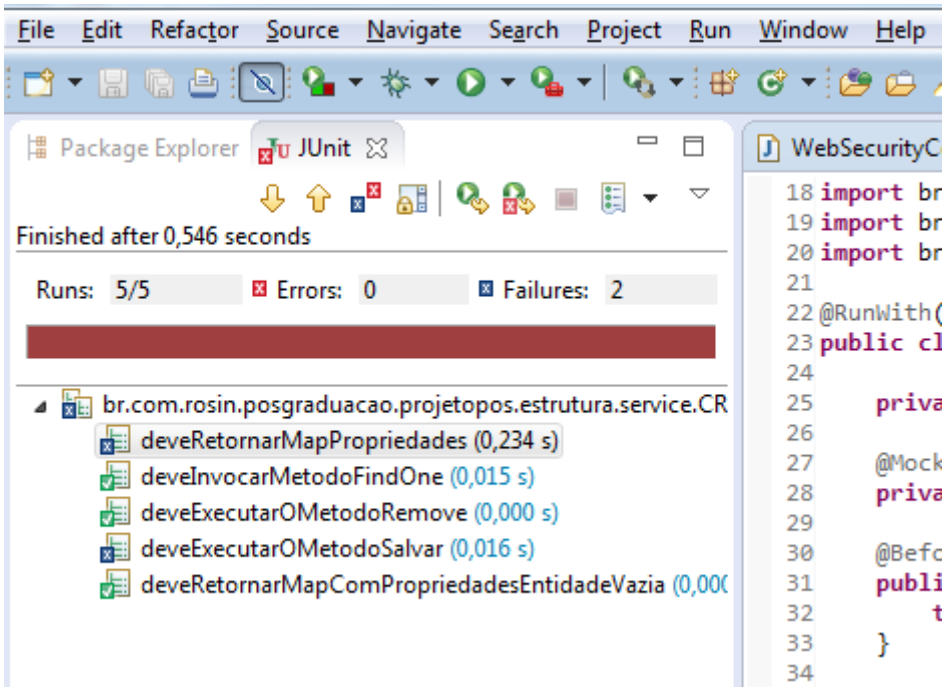

Figura 14 - Teste com falhas

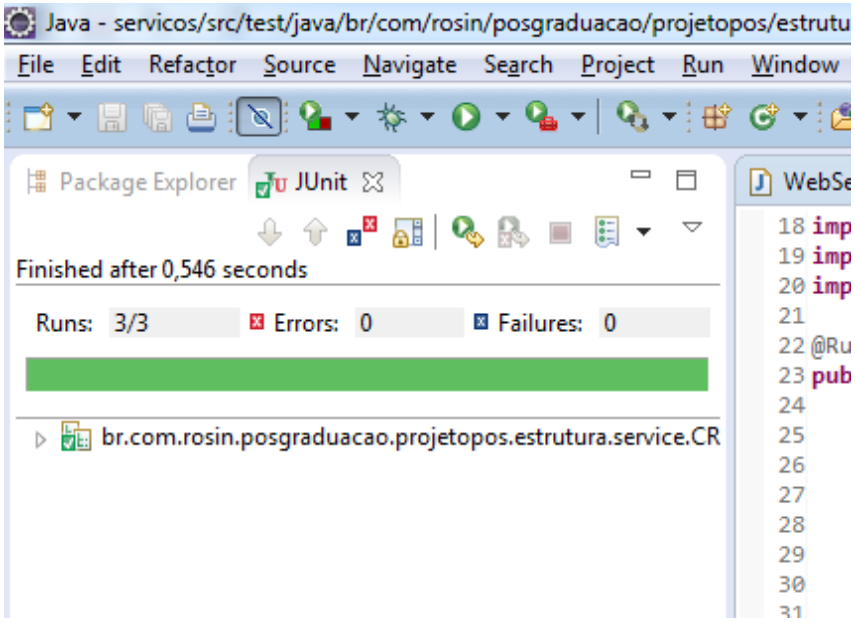

Figura 15 - Teste Passando.

Para a parte de *view* do sistema (páginas web) foi utilizada a tecnologia JSP (*Java Server Pages*). Para agilizar o processo de criação de páginas, foram desenvolvidas *tags*[6] JSP para o layout da aplicação e para campos de entrada, como texto, número, data entre outros. A Listagem 14 apresenta a *tag cnpj*, que cria um campo texto com uma

[6] A utilização de *tags* permite o reaproveitamento do código da criação de um componente.

máscara de CNPJ. Nas linhas 3, 4, 5 e 6 são definidos os atributos que a *tag* pode receber. Nas linhas 12 a 16 é apresentado um código JavaScript que cria a máscara para o componente por meio da biblioteca JQuery, e a partir da linha 17 é apresentado o HTML que será renderizado.

template.tag | index.jsp | *cnpj.tag

```
1  <%@ tag language="java" pageEncoding="UTF-8"%>
2
3  <%@ attribute name="name" required="true" %>
4  <%@ attribute name="label" required="true" %>
5  <%@ attribute name="id" required="true" %>
6  <%@ attribute name="required" required="false" type="java.lang.Boolean" %>
7
8  <%@ tag body-content="empty" %>
9
10 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
11
12 <script type="text/javascript">
13     $(function() {
14         $( "#${id}" ).inputmask("99.999.999/9999-99");
15     });
16 </script>
17 <div class="form-group" id="_div${name}">
18     <label class="control-label col-sm-4" for="${name}"><c:if test="${not empty required && required}">* </c:if>${label}:</label>
19     <div class="col-sm-6">
20         <input type="text" name="${name}" id="${id}" class="form-control" data-parsley-class-handler="#_div${name}"
21             <c:if test="${not empty required && required}">required</c:if>>
22         <span id="_error${name}" class="help-block" style="display: none"></span>
23     </div>
24 </div>
```

Listagem 14 - Tag CNPJ

Na Listagem 15 é apresentado um exemplo de uso da *tag cnpj*, bem como das *tags texto* e *cep*.

```
33  <div id="my-tab-content" class="tab-content">
34  <div id="dadosGerais" class="tab-pane active">
35    <fieldset>
36        <form:texto label="Nome" name="nome" id="nome" required="true" requiredMessage="Nome obrigatório"/>
37        <form:texto label="Razão Social" name="razaoSocial" id="razaoSocial" required="true"/>
38        <form:cnpj label="CNPJ" name="cnpj" id="cnpj" required="true" />
39        <form:texto label="Inscrição Estadual" name="inscricaoEstadual" id="inscricaoEstadual" required="true"/>
40    </fieldset>
41  </div>
42  <div id="endereco" class="tab-pane">
43    <fieldset>
44        <form:texto label="Logradouro" name="logradouro" id="logradouro" required="true" />
45        <form:texto label="Número" name="numero" id="numero" required="true" />
46        <form:cep label="CEP" name="cep" id="cep" required="true" />
47        <form:texto label="Complemento" name="complemento" id="complemento" />
48        <form:texto label="Referência" name="referencia" id="referencia" />
```

Listagem 15 - Uso da tag cnpj

A Figura 16 apresenta o código gerado por meio da *tag cnpj* que é renderizado pelo navegador.

```
404
405 <script type="text/javascript">
406         $(function() {
407                 $( "#cnpj" ).inputmask("99.999.999/9999-99");
408         });
409 </script>
410 <div class="form-group" id="_divcnpj">
411         <label class="control-label col-sm-4" for="cnpj">* CNPJ:</label>
412         <div class="col-sm-6">
413                 <input type="text" name="cnpj" id="cnpj" class="form-control" data-parsley-class-handler="#_divcnpj" required>
414                 <span id="_errorcnpj" class="help-block" style="display: none"></span>
415         </div>
416 </div>
417
418
419
420
```

Figura 16 - Código renderizado pelo navegador

Nas telas que utilizam grid, é utilizada a *tag* grid. Esta *tag* contém o código para montar um grid com os dados cadastrados no banco de dados, utilizando a biblioteca DataTables. A *tag* recebe por parâmetro um id para identificar o componente dentro do HTML, as colunas que serão mostradas, qual será a URL utilizada para buscar os dados e um nome para a variável que será referenciada no JavaScript. Ao carregar a tela, o componente faz uma requisição para a URL, informando quais são as colunas que serão apresentadas e recebe os dados no formato JSON. Na Listagem 14 é apresentado o código da *tag* grid.

```
    $(function() {
        for (var i = 0; i < ${columns}.length; i++) {
            $("#${id} thead tr").append($('<td/>').text(${columns}[i].label));
            $("#${id} tfoot tr").append($('<td/>').text(${columns}[i].label));
        }

        var colunas = ${columns}.slice();
        colunas.pop();
        var paramColumns = "?columns=" + JSON.stringify(colunas);
        if ("${ajax}".indexOf("?") > 0) {
            paramColumns = "&columns=" + JSON.stringify(colunas);
        }

        ${varGrid} = $('#${id}').dataTable( {
            "ajax": "${ajax}" + paramColumns,
            "scrollX": true,
            "scrollY": 314,
            "pagingType": "full_numbers",
            "lengthMenu": [[10, 15, 20, 25, 30, 50, -1], [10, 15, 20, 25, 30, 50, "Todos"]],
            "language": {
                "url": "/static/datatables/json/brasil.json"
            },
            "columns": ${columns}
        } );

        //${varGrid}.fnSettings().oScroll.sY = $('#inner_west_panel').height()-125;
    });
</script>
<div style="border: 1px solid #DDDDDD; padding: 1px; -webkit-border-radius: 5px; -moz-border-radius: 5px; border-radius: 5px;">
    <table id="${id}" class="table table-striped table-bordered responsive">
         <thead>
            <tr></tr>
         </thead>

         <tfoot>
            <tr>
            </tr>
         </tfoot>
    </table>
</div>
```

Listagem 16 - Tag grid

A classe CRUDController possui um método chamado *dados()*, que responde para a URL que é chamada na *tag* grid. O qual chama o método *listarDadosGrid()* na classe CRUDService, que faz a leitura das colunas solicitadas e, com a utilização da API Reflection do Java, busca os dados no banco de dados e retorna para o grid. Esse código é apresentado na listagem 17.

```
24      public JSONObject listarDadosGrid(String columns) {
25          try {
26              CreateJsonGrid<T> c = new CreateJsonGrid<>(repository.findAll(), ColumnsFinder.getMapColunas(columns));
27              return c.montarJSON();
28          } catch (JSONException | NoSuchFieldException | SecurityException | IllegalAccessException
29                  | IllegalArgumentException | InvocationTargetException | NoSuchMethodException e) {
30              e.printStackTrace();
31              JSONObject resposta = new JSONObject();
32              try {
33                  resposta.put("SUCCESS", false).put("mensagem", e.getMessage());
34              } catch (JSONException je) {
35                  je.printStackTrace();
36              }
37              return resposta;
38          }
39      }
```

Listagem 17 - Código da classe CRUDService

O método *listarDadosGrid()* apresentado na Listagem 17, utiliza a classe ColumnsFinder, apresentada na Listagem 18, para montar uma estrutura do tipo *Map* com as colunas que foram passadas pelo *request* por meio do método *getMapColunas()*. O qual retorna um objeto do tipo ColumnGrid, que possui os dados das colunas que devem ser exibidas no grid, como, por exemplo, o nome do campo que corresponde à um atributo de uma entidade. Após isso, é utilizada a classe CreateJsonGrid, apresentada na Listagem 19, a qual é responsável por criar o JSON que será retornado para a tela. Após carregar as colunas, é chamado o método montarJson*()*, da classe CreateJsonGrid, que é responsável por montar o JSON que será retornado, o qual contém os dados que devem ser exibidos no grid. O método *montarJson()* utiliza a classe FieldValueFinder, apresentada na Listagem 20, que utiliza a API Reflection do Java para buscar os valores dos campos e retornar um *array* do tipo JSON.

```
10 public class ColumnsFinder {
11
12     public static Map<String, String> getMapColunas(String colunas) throws JSONException {
13         Map<String, String> retorno = new HashMap<String, String>();
14         if (! colunas.isEmpty()) {
15             JSONArray array = new JSONArray(colunas);
16             for (int i = 0; i < array.length(); i++) {
17                 ColumnGrid column = new ColumnGrid((JSONObject) array.get(i));
18                 retorno.put(column.getName(), column.getData());
19             }
20         }
21         return retorno;
22     }
23 }
```

Listagem 18 - Classe ColumnsFinder

```java
public class CreateJsonGrid <T> {

    private List<T> registros;
    private Map<String, String> colunas;

    public CreateJsonGrid(List<T> registros, Map<String, String> colunas) {
        this.registros = registros;
        this.colunas = colunas;
    }

    public JSONObject montarJSON() throws NoSuchFieldException, SecurityException,
            IllegalAccessException, IllegalArgumentException, InvocationTargetException,
            NoSuchMethodException, JSONException {
        JSONArray array = new JSONArray();
        for (T registro : registros) {
            JSONObject objeto = new JSONObject();
            FieldValueFinder ff = new FieldValueFinder(registro);
            for (String key : colunas.keySet()) {
                Optional<Object> value = Optional.ofNullable(ff.find(key));
                objeto.put(colunas.get(key), value.orElse(""));
            }
            array.put(objeto);
        }

        return new JSONObject().put("data", array);
    }
}
```

Listagem 19 - Classe CreateJsonGrid

```java
public class FieldValueFinder {

    private Class<?> clazz;
    private Object registro;

    public FieldValueFinder(Object registro) {
        if (! Objects.isNull(registro)) {
            this.clazz = registro.getClass();
            this.registro = registro;
        }
    }

    public Object find(String nome) throws NoSuchFieldException,
            SecurityException, IllegalAccessException,
            IllegalArgumentException, InvocationTargetException,
            NoSuchMethodException {
        GetFieldValue getValue = new GetFieldValue();

        if (nome.contains(".")) {
            String nome2 = nome.substring(0, nome.indexOf("."));
            Object value = getValue.get(FindUnderlying.findField(nome2, clazz), clazz, registro);

            FieldValueFinder fvf = new FieldValueFinder(value);
            return fvf.find(nome.substring(nome.indexOf(".") + 1));
        }

        return getValue.get(findField(nome, clazz), clazz, registro);
    }
}
```

Listagem 20 - Classe FieldValueFinder

5 CONCLUSÃO

Este trabalho foi desenvolvido visando criar uma solução para uma empresa prestadora de serviços, sendo que o objetivo foi alcançado.

O uso do *framework* Spring se mostrou bastante produtivo, possibilitando a criação de classes genéricas, que agilizam o desenvolvimento e evitam a duplicação de código, principalmente nos *controllers* e *services*. O *framework* Spring Data auxilia no acesso a dados, sendo que é necessário escrever pouco código para ter acesso a essa funcionalidade, dando muita agilidade ao desenvolvimento. Também o *framework* Spring Boot auxilia nos primeiros passos do projeto e na configuração de novas dependências para o *framework* Spring.

O uso da API JPA para realizar o Mapeamento Objeto Relacional se mostrou eficiente, juntamente com o uso dos *frameworks* Spring Data e Hibernate, tornando a tarefa de acesso a dados simples.

Assim como a criação de *controllers* auxiliou na produtividade da codificação do lado servidor da aplicação, a criação de *tags* JSP auxiliou dando agilidade na criação do lado cliente, tornando a tarefa de criar uma tela mais rápida, encapsulando os códigos exigidos pela biblioteca Bootstrap para criação de formulários, lógicas de validação e de criação de máscaras para os componentes.

Um erro cometido no momento do design da aplicação foi desenhar os formulários de cadastros em janelas do tipo modal. Essa decisão aumentou a complexidade do código e limitou algumas possibilidades de *layout*. Para o futuro, pretende-se ajustar o layout, fazendo com que cada cadastro seja realizado em uma nova tela. Além das alterações no *layout*, pretende-se ajustar as funcionalidades para que fiquem mais adequadas com o processo de uma empresa, pois atualmente o sistema não atende como deveria as reais necessidades do dia a dia de uma empresa, faltando melhorar as funcionalidades de Ordem de Serviço e de Contas a Pagar e a Receber, bem como repensar o fluxo dessas informações no sistema. Com os devidos ajustes, pretende-se também publicar o aplicativo em um servidor web e disponibilizar para o uso.

REFERÊNCIAS

ANICHE, Mauricio. **Test-Driven Development:** Teste e Design no Mundo Real. São Paulo: Casa do Código, 2014.

BECK, Kent. TDD Desenvolvimento Guiado por Testes/ Kent Beck; tradução: Jean Felipe Patikowski Cheiran; revisão técnica: Marcelo Soares Pimenta; Porto Alegre; Bookman, 2010.

DEITEL, Paul; DEITEL, Harvey. **Java Como Programar.** 8. ed. São Paulo: Pearson, 2009.

FOWLER, Martin. **Inversion Of Control.** 2005. Disponível em: <http://martinfowler.com/bliki/InversionOfControl.html>. Acesso em: 17 set. 2015.

FOWLER, Martin. **Inversion of Control Containers and the Dependency Injection pattern.** 2004. Disponível em: <http://www.martinfowler.com/articles/injection.html>. Acesso em: 17 set. 2015.

ORACLE. **Java Platform, Enterprise Edition:** The Java EE Tutorial. 2014. Disponível em: <https://docs.oracle.com/javaee/7/tutorial/overview.htm#BNAAW>. Acesso em: 29 set. 2015.

ORACLE. **The History of Java Technology.** Disponível em: <http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html>. Acesso em: 24 ago. 2015.

POSTGRE. **About**. 2015. Disponível em: <http://www.postgresql.org/about/>. Acesso em: 12 out. 2015.

SPRING. **Spring Framework Reference Documentation.** 2014. Disponível em: <http://docs.spring.io/spring/docs/4.2.2.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#core-convert-programmatic-usage>. Acesso em: 17 set. 2015.

VISUAL-PARADIGM. 2015. Disponível em: <http://www.visual-paradigm.com/features/>. Acesso em: 12 out. 2015.