UNIVERSIDADE ESTADUAL DO NORTE DO PARANÁ

***CAMPUS* LUIZ MENEGHEL**

RICARDO MEDEIROS DA COSTA JUNIOR

IMPLEMENTAÇÃO DA PRÁTICA DE DESENVOLVIMENTO GUIADO POR TESTES EM PROTÓTIPO DE SISTEMA DE VENDAS

Bandeirantes

2013

RICARDO MEDEIROS DA COSTA JUNIOR

IMPLEMENTAÇÃO DA PRÁTICA DE DESENVOLVIMENTO GUIADO POR TESTES EM PROTÓTIPO DE SISTEMA DE VENDAS.

Trabalho de Conclusão de Curso apresentado à Universidade Estadual do Norte do Paraná-UENP, *campus* Luiz Meneghel – como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Me. Glauco Carlos Silva

Bandeirantes

2013

RICARDO MEDEIROS DA COSTA JUNIOR

IMPLEMENTAÇÃO DA PRÁTICA DE DESENVOLVIMENTO GUIADO POR TESTES EM PROTÓTIPO DE SISTEMA DE VENDAS.

Trabalho de Conclusão de Curso apresentado à Universidade Estadual do Norte do Paraná-UENP, *campus* Luiz Meneghel – como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

COMISSÃO EXAMINADORA

_____
Prof. Me. Glauco Carlos Silva
UENP - Campus Luiz Meneghel

_____
Prof. Wellington Della Mura
UENP - Campus Luiz Meneghel

_____
Prof. Me Luiz Fernando Legore do Nascimento
UENP - Campus Luiz Meneghel

Bandeirantes, __ de_____ de 2013

RESUMO

Este trabalho aplica a técnica *Test-Driven Development* (TDD - Desenvolvimento Guiado por Testes) em um protótipo de sistema de vendas no varejo. Em particular é demonstrado como um desenvolvimento incremental e interativo com TDD pode auxiliar a equipe de desenvolvimento a identificar erros em testes de regressão e como tornar seu código simples; mantendo o fácil entendimento e mais flexível, deixando acessível para qualquer possível modificação nos requisitos, comumente de um projeto desenvolvido utilizando uma metodologia ágil. No término do trabalho foi constatado que o emprego da técnica trouxe melhorias para o desenvolvimento do protótipo em questão. Pode-se constatar isto após utilizar como métricas alguns conceitos do guia *Project Management Body of Knowledge* (PMBOK).

Palavras-chave: TDD, Refatoração, Vermelho-Verde-Refatore.

ABSTRACT

This work applies the technique Test-Driven Development (TDD - Test-Driven Development) on a prototype system for retail sales. In particular it is shown as an incremental and iterative development with TDD can help the development team to identify errors in regression tests and making your code simpler, keeping the easy to understand and accessible to more flexible, leave any possible changes in requirements, commonly a project developed using an agile methodology. At the end of the work it was found that the use of technical improvements brought to the development of the prototype in question. It can be seen that after using metrics as some concepts of the *Project Management Body of Knowledge* (PMBOK) ® Guide.

Keywords: TDD, Refatoring, Red-Green-Refactor.

LISTA DE FIGURAS

LISTA DE SIGLAS

CNPJ	Cadastro Nacional de Pessoas Jurídicas
CPF	Cadastro de Pessoas Físicas
CRUD	*Create*, *Read*, *Update* e *Delete*
CVS	*Concurrent Version System*
IDE	*Integrated Development Environment*
IE	Inscrição Estadual
NIST	*National Institute of Standards and Technology*
ORM	*Object-relational mapping*
PMBOK	*Project Management Body of Knowledge*
RG	Registro Geral
SGBD	Sistema de Gerenciamento de Banco de Dados
SQL	*Structured Query Language*
SVN	*Subversion*
TDD	*Test Driven Development*
TFS	*Team Foundation Server*
UI	*User Interface*

SUMÁRIO

1. INTRODUÇÃO

Na visão de Evans (2003), a maior complexidade dos aplicativos está no modelo de negócios. Se essa complexidade não for tratada no *design* do domínio, compreender o *software* se torna uma tarefa árdua e os desenvolvedores têm dificuldades para alterar ou adicionar novas funcionalidades no sistema.

Freeman e Pryce (2010) argumentam que o *Test-driven Development* (TDD), tem como objetivo principal garantir um sistema confiável, mantendo o código constantemente testado. Garantindo que a cada modificação ou alteração em determinado requisito, não terão os recursos já existentes rompidos. *Test-driven Development* (TDD) facilita a vida dos desenvolvedores, torna o teste uma parte do processo de desenvolvimento e aumenta a produtividade da equipe, que não irá precisar realizar a testagem manual. Além disso, esta prática ainda torna o *design* mais intuitivo, flexível e de fácil compreensão, por meio do processo de refatoramento.

Este trabalho apresenta uma proposta de implementação da prática de TDD em um sistema pouco ambicioso, com a finalidade de demonstrar as vantagens de utilizar essa técnica em sistemas pequenos.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

O objetivo deste trabalho é desenvolver um sistema gerencial de vendas no varejo aplicando a prática do TDD, demonstrando por meio de métricas quais foram às vantagens do desenvolvimento do protótipo utilizando TDD; sua utilidade em identificar erros de regressão e como essa técnica auxilia o desenvolvimento mantendo o código simples, fácil de compreender e modificar.

1.1.2 Objetivos Específicos

- Definir as tecnologias utilizadas para tornar o processo de desenvolvimento mais produtivo;
- Definir as funcionalidades do sistema proposto;
- Implementar testes de unidade que falha, para descrever o que a funcionalidade necessita;
- Fazer cada teste de unidade passar, garantindo que a regra de negócio tenha sido alcançada;
- Refatorar cada solução encontrada a fim de simplificar o modelo de domínio tornando o mais intuitivo;
- Implementar testes de integração com a camada de infraestrutura e *User Interface* (UI);
- Validar os testes de sistema, assegurando a qualidade externa do *software*;
- Repetir o ciclo até o término do desenvolvimento do protótipo;
- Verificar se todos os requisitos iniciais do protótipo foram alcançados;
- Definir métricas para avaliar se o projeto foi bem sucedido; e,
- Avaliar o protótipo desenvolvido com a técnica do desenvolvimento guiado por testes, conforme as métricas definidas anteriormente.

1.2 JUSTIFICATIVA

Foi constatado por meio de um estudo realizado em 2002 pelo *National Institute of Standards and Technology* (NIST)[1], que a economia dos Estados Unidos da América sofre um prejuízo estimado em U$59.5 bilhões por ano com erros em *software*.

De acordo com o estudo, os desenvolvedores concordaram que o ideal seria descobrir o ponto onde o erro foi introduzido no código e quais consequências que ele traz ao processo de produção, em vez de exigir que o produto seja testado somente no término do desenvolvimento. Soma-se a isso, Begel e Simon (2008) quando afirmam

[1] NIST é a agência federal de tecnologia dos Estados Unidos da América, que trabalha com a indústria para desenvolver e aplicar tecnologias, padrões e medições.

que recém-formados na Microsoft® gastam mais tempo lendo o código do que desenvolvendo.

Portando é imprescindível que um projeto de *software* bem sucedido deve ser constantemente testado e de fácil entendimento, facilitando o trabalho dos desenvolvedores e tornando-os mais produtivos.

1.3 METODOLOGIA

Este trabalho foi desenvolvido, por meio de pesquisa bibliográfica sobre a prática do TDD. Neste, foi pesquisado qual ambiente de desenvolvimento foi utilizado para aplicar TDD na linguagem de programação C# (CSharp) e quais *frameworks* foram escolhidos para tornar o processo de desenvolvimento mais produtivo.

As melhores tecnologias foram selecionadas a fim de se obter um desenvolvimento produtivo com objetivo de criar um sistema com qualidade que atenda integralmente aos requisitos definidos no início do trabalho.

Foram definidos os requisitos funcionais do sistema de vendas, definindo as regras de negócio do protótipo proposto. Tais como cadastros, atendimentos, fluxo de caixa com controle financeiro.

1.4 ORGANIZAÇÃO DO TRABALHO

O trabalho esta dividido em seis capítulos: Introdução, Fundamentação Teórica, Métricas para Avaliar o Protótipo, Desenvolvimento, Conclusão e Trabalhos Futuros.

No capítulo 2 são mostrados conceitos sobre o Desenvolvimento Guiado Por Testes, Diferenças da Maneira Tradicional, Ciclo Guiado por Teste e Refatoração.

No capítulo 3 são exibidas as métricas que foram utilizadas para avaliar o processo de desenvolvimento usando TDD. Logo em seguida, no capítulo 4 é detalhada como foi feita a implementação do protótipo, desde o levantamento dos requisitos à finalização do ciclo.

No capítulo 5 consta a conclusão do trabalho realizado e no capítulo 6 sugestões para trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

No presente capítulo são abordados, na seção 2.1, Desenvolvimento Guiado Por Testes; na seção 2.2, Diferenças da Maneira Tradicional; na seção 2.3, Ciclo Guiado por Teste; na seção 2.4, Refatoração.

1.5 DESENVOLVIMENTO GUIADO POR TESTES

Segundo Freeman e Pryce (2010), muitos desenvolvedores acreditam que a tarefa de criar testes automatizados para garantir a qualidade externa do código é um trabalho desgastante e perca de tempo. Estes profissionais da área argumentam que escrever os testes após codificar uma funcionalidade compromete a produtividade da equipe. Os autores também afirmam que vários programadores preferem o método "tradicional", no qual é feito a testagem manual da aplicação, devido a estes motivos supracitados.

Ao utilizar TDD, este processo é invertido. A ideia não é complexa, porém foge completamente do método tradicional. O teste é escrito antes da implementação, toda implementação visa fazer o determinado teste passar.

De acordo com Freeman e Pryce (2010) essa abordagem proporciona diversos benefícios que são:

> - Esclarecer os critérios de aceitação para a próxima etapa de trabalho - temos que nos perguntar quando sabemos que acabamos (projeto);
> - Incentivar a escrever componentes livremente acoplados, de modo que eles possam ser facilmente testados individualmente e, em níveis mais altos, combinados (projeto);
> - Acrescentar uma descrição executável do que o código faz (projeto);
> - Adicionar a uma suíte completa de regressão (implementação);
> - Detectar erros enquanto o contexto está nítido em nossa mente (implementação); e,
> - Permitir saber quando fizemos o bastante, desestimulando "a medalha de ouro" e recursos desnecessários (projeto).

Portanto, apesar do nome sugerir, TDD não é está apenas relacionado a testes. O TDD é uma técnica de projeto de *software*. (LIU, 2010).

Essa afirmação tem base no auxílio que o uso da prática do TDD faz para se obter a qualidade externa e interna do código.

> [...] a qualidade externa está relacionada a quão bem o sistema preenche as necessidades de seus clientes e usuários (é funcional, confiável disponível, sensível e etc.), e qualidade interna está relacionada a quão bem ele satisfaz as necessidades de seus desenvolvedores e administradores (é fácil de entender, de alterar e etc.) (FREEMAN e PRYCE, 2010, p. 10)

A qualidade interna é tão importante quanto à qualidade externa, que por sua vez é mais difícil de alcançar. A equipe que utiliza TDD consegue atingir a qualidade interna do *software* com mais facilidade.

Como o procedimento do ciclo do TDD se baseia em testes de unidade, as classes precisam estar com sua responsabilidade clara e bem definida – facilitando as chamadas e as verificações – e com as dependências devidamente explícitas, facilitando uma possível substituição. Em resumo, precisam ser coesas e fracamente acopladas[2], respectivamente (FREEMAN; PRYCE, 2010).

No próximo tópico serão explicadas as diferenças de TDD para a abordagem convencional.

[2] Acoplamento e coesão são princípios da engenharia de software que devem ser seguidos para o bom desenvolvimento orientado a objetos.

1.6 DIFERENÇAS DA MANEIRA TRADICIONAL

Corrigir um problema no *design*, que foi identificado na etapa final de projeto sai caro. Caso uma alteração tenha como custo “1x” na etapa de desenvolvimento ela terá um custo de “60x a 100x” quando feito na fase de implantação. (Pressman 2001).

Além de que, estudos comparando um grupo que utilizou o TDD e outro que realizava os testes após a implementação, mostrou que os erros encontrados no desenvolvimento eram corrigidos mais rapidamente pelo grupo que utilizou TDD. (ANICHE, 2012 apud Lui e Chan, 2004).

Segundo ANICHE (2012), escrever o teste após a implementação retarda o retorno do *feedback* . Recebendo o *feedback* mais cedo, o desenvolvedor pode melhorar o código e corrigir os possíveis erros com um custo menor e com maior facilidade. Ele enfatiza que uma recente implementação é mais simples que corrigir um código de dias atrás.

Isso fica claro na Figura 1.

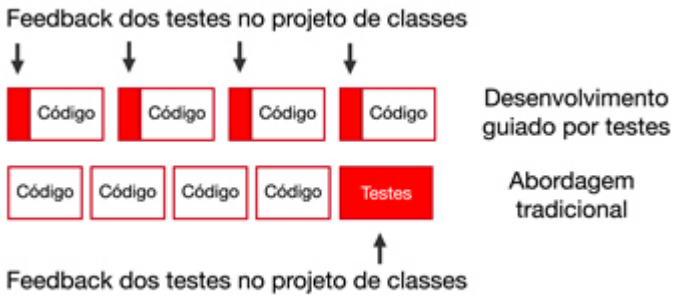

Figura 1 - Comparação de retorno entre TDD e abordagem tradicional
Fonte: ANICHE (2012, p. 29)

No entanto, a utilização de TDD por uma equipe com pouca experiência e falta de conhecimento do paradigma de orientação a objetos pode afetar a produtividade e aumentar consideravelmente as linhas de código de teste. (ANICHE, 2012)

1.7 CICLO GUIADO POR TESTE

Na visão de ANICHE (2012), o ciclo guiado por teste, também conhecido como *red-green-refactor*[3], é resumido da seguinte maneira:

- Escreve-se um teste de unidade para uma nova funcionalidade;
- Observa-se o teste falhar;
- Implementa-se a solução mais simples para resolver o problema;
- Veja o teste passar; e,
- Melhora-se (refatorando) o código se houver necessidade.

Na Figura 2 é exibido como o ambiente de desenvolvimento Eclipse[4] executa a suíte de testes.

Na Figura 3 é mostrado o ciclo guiado por teste.

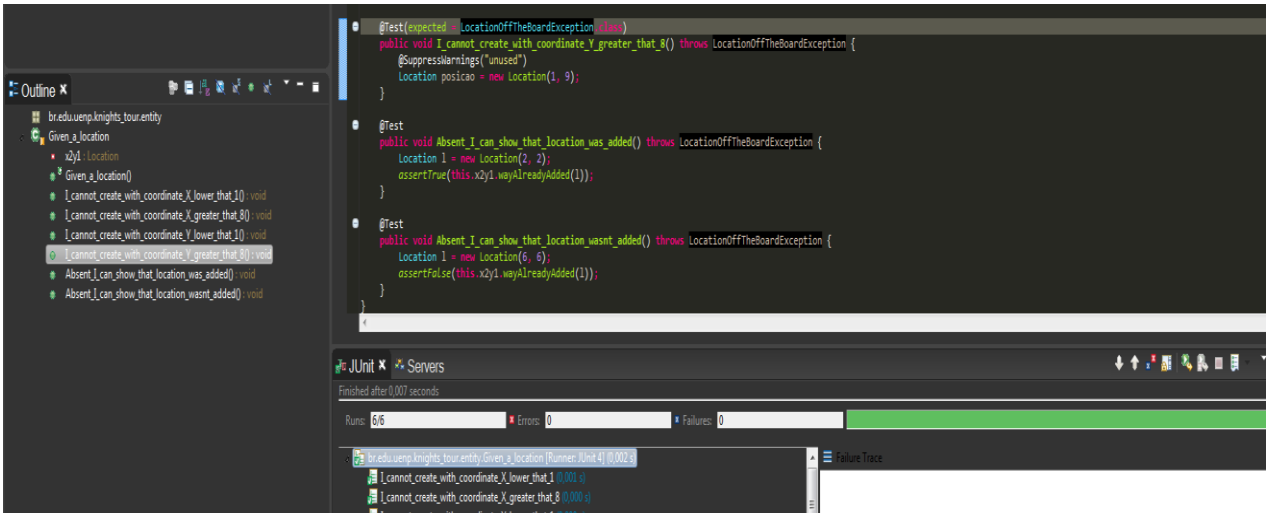

Figura 2 - Exemplo de Testes jUnit no *Integrated Development Environment* (IDE) Eclipse
Fonte: Elaborado pelo autor.

[3] Uma alusão ao *framework* xUnit de testes de unidades automatizados. No qual indica vermelho para um teste que falha e verde para um teste que passa.

[4] Eclipse é uma popular ferramenta de desenvolvimento.

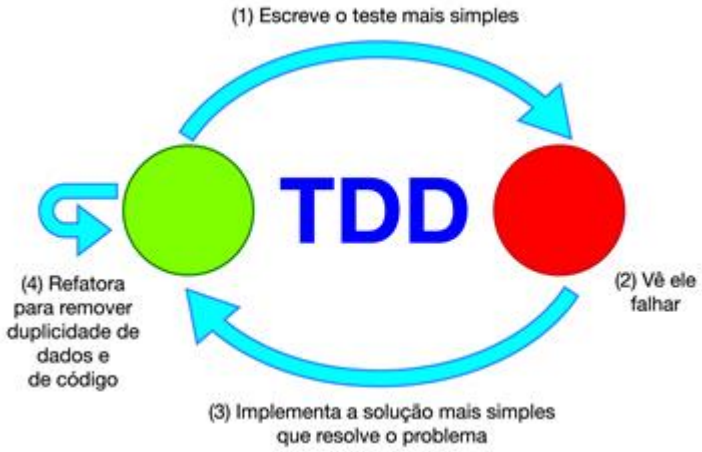

Figura 3 - Ciclo Guiado por Teste
Fonte: ANICHE (2012, p. 27)

Aniche argumenta que realizando este ciclo são obtidas as seguintes vantagens:

> - Foco no teste e não na implementação. Ao começar pelo teste, o programador consegue pensar somente no que a classe deve fazer, e esquece por um momento da implementação. Isso o ajuda a pensar em melhores cenários de teste para a classe sob desenvolvimento.
> - Código nasce testado. Se o programador pratica o ciclo corretamente, isso então implica em que todo o código de produção escrito possui ao menos um teste de unidade verificando que ele funciona corretamente.
> - Simplicidade. Ao buscar pelo código mais simples constantemente, o desenvolvedor acaba por fugir de soluções complexas, comuns em todos os sistemas.
> - O praticante de TDD escreve código que apenas resolve os problemas que estão representados por um teste de unidade. Quantas vezes o desenvolvedor não escreve código desnecessariamente complexo?
>   - Melhor reflexão sobre o design da classe. No cenário tradicional, muitas vezes a falta de coesão ou o excesso de acoplamento é causado muitas vezes pelo desenvolvedor que só pensa na implementação e acaba esquecendo como a classe vai funcionar perante o todo. Ao começar pelo teste, o desenvolvedor pensa sobre como sua classe deverá se comportar perante as outras classes do sistema. O teste atua como o primeiro cliente da classe que esta sendo escrita. Nele, o desenvolvedor toma decisões como o nome da classe, os seus métodos, parâmetros, tipos de retorno, e etc. No fim, todas elas são decisões de design e, quando o desenvolvedor consegue observar com atenção o código do teste, seu design de classes pode crescer muito em qualidade.

No próximo tópico será abordado o conceito de refatoração e como essa técnica auxilia a prática do TDD a melhorar a qualidade interna do *software*.

1.8 REFATORAÇÃO

Fowler (2004) define refatoração como: “[...] o ato de alterar a estrutura interna de um código existente sem modificar seu funcionamento.”

Essa micro técnica trás excelentes benefícios para o projeto de *software*, visto que ela é utilizada como forma de melhorar a qualidade interna da aplicação. Se um método está com difícil compreensão, o desenvolvedor pode refatorá-lo para aperfeiçoar o código. O mesmo pode ser dividido em sub-métodos, no qual é possível ter determinada funcionalidade encapsulada para um futuro reaproveitamento.

Essa constante busca pelo aprimoramento do código é uma etapa do ciclo guiado por testes. É por meio da refatoração que é possível obter o *feedback* do código fonte. Assim melhorando o modelo de negócios, simplificando o código – essencial para manutenibilidade, sustentabilidade e trabalho em equipe dos desenvolvedores que estão em um mesmo projeto de um sistema complexo. (FREEMAN; PRYCE, 2010).

Na Figura 4 é mostrado como a refatoração influencia diretamente no *design* da aplicação.

```
[TestMethod]
[ExpectedException(typeof(ExcecaoEstadoRepetido))]
public void Nao_Posso_Validar_Um_Estado_Novo_Com_UF_De_Outro_Estado_Ja_Cadastrado()
{
    Estado estadoCadastradoComMesmaUF = new Estado("PR");
    Estado estadoNovo = new Estado("PR");

    ValidadorEstadoJaExistente validacao = new ValidadorEstadoJaExistente(estadoCadastradoComMesmaUF);
    validacao.ValidaEstadoNovo(estadoNovo);
}
```

Figura 4 - Regra de negócio da classe Estado
Fonte: Elaborada pelo autor

Nesse projeto, é possível observar que havia uma regra de negócio na qual não era permitida a inserção de um novo estado com o atributo unidade federativa igual de outro estado já cadastrado no banco de dados.

A ideia inicial de implementação seria uma consulta ao repositório[5] de estado e verificar pela unidade federativa passando o estado a ser verificado pelo parâmetro, caso houvesse um estado que estivesse cadastrado no sistema com a mesma unidade federativa do objeto a ser verificado, uma exceção seria lançada.

Sendo assim, foi escrito um teste de unidade para essa funcionalidade e logo após foi realizada a codificação para esse teste passar. O teste passou como esperado, no entanto, essa implementação se mostrou falha. A fachada[6] estava acoplada em demasia com outros elementos do projeto.

Logo após, o código foi refatorado e uma nova classe surgiu, ela continha a responsabilidade de fazer essa verificação. Como é mostrada na Figura 5.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Eimbo.Dominio.Cadastro.Excecao;
using Eimbo.Dominio.Comum.Entidade;

namespace Eimbo.Dominio.Cadastro.Servico
{
    public class ValidadorEstadoJaExistente
    {
        private Estado _estadoCadastradoComMesmaUF;

        public ValidadorEstadoJaExistente(Estado estadoCadastradoComMesmaUF)
        {
            this._estadoCadastradoComMesmaUF = estadoCadastradoComMesmaUF;
        }

        private Boolean NaoFoiEncontradoEstadoComMesmaUF { get { return (this._estadoCadastradoComMesmaUF == null); } }

        public void ValidaEstadoNovo(Estado estadoNovo)
        {
            if (this.NaoFoiEncontradoEstadoComMesmaUF) return;

            if (this._estadoCadastradoComMesmaUF.UF.Equals(estadoNovo.UF))
                throw new ExcecaoEstadoRepetido();
        }

        public void ValidaEstadoAlterado(Estado estadoAlterado)
        {
            if (this.NaoFoiEncontradoEstadoComMesmaUF) return;

            if ((!this._estadoCadastradoComMesmaUF.Equals(estadoAlterado)) && (this._estadoCadastradoComMesmaUF.UF.Equals(estadoAlterado.UF)))
                throw new ExcecaoEstadoRepetido();
        }
    }
}
```

Figura 5 - Código aperfeiçoado
Fonte: Elaborada pelo autor

Há sistemas onde a complexidade é muito grande, as dependências estão implícitas e os componentes estão fortemente acoplados. Para estes casos, a

[5] Repositório um padrão de desenvolvimento exposto por Evans (2003).

[6] Fachada é um padrão de projeto mostrado por Gamma et al. (1994)

refatoração é uma tarefa complicada e arriscada, se o projeto não possui testes automatizados com o intuito de identificar erros de regressão.

3. MÉTRICAS PARA AVALIAR O PROTÓTIPO

Neste capítulo serão definidas as métricas para avaliar o protótipo no término do trabalho. Estas métricas foram apresentadas por Junqueira (2012) no qual se baseou no PMBOK®[7]. As métricas são:

- Escopo;
- Qualidade;
- Tempo; e,
- Riscos.

No fechamento deste trabalho, o protótipo será avaliado conforme as métricas mostradas acima. De acordo com estas métricas, serão definidos se o desenvolvimento utilizando TDD obteve vantagens em relação ao método de fabricar *software* tradicional.

As métricas segundo o guia PMBOK®, possuem as seguintes definições:

> **Escopo do projeto**. O trabalho que precisa ser realizado para entregar um produto, serviço ou resultado com as características e funções especificadas. (2004, p. 104)
> **Realizar o controle da qualidade**
> Este é o processo necessário para monitorar resultados específicos do projeto a fim de determinar se eles estão de acordo com os padrões relevantes de qualidade e identificar maneiras de eliminar as causas de um desempenho insatisfatório. (2004, p.63)
> **Tempos.** Define quando e com que frequência o processo de gerenciamento de riscos será executado durante todo o ciclo de vida do projeto e estabelece as atividades de gerenciamento de riscos que serão incluídas no cronograma do projeto. (2004, p. 243)
> O **risco do projeto** é um evento ou condição incerta que, se ocorrer, terá um efeito positivo ou negativo sobre pelo menos um objetivo do projeto, como tempo, custo, escopo ou qualidade (ou seja, em que o objetivo de tempo do projeto é a entrega de acordo com o cronograma acordado; em que o objetivo de custo do projeto é a entrega de acordo com o custo acordado, etc.) (2004, p. 238)

[7] PMBOK® é um guia para gerenciamento de projetos.

4. DESENVOLVIMENTO

Esse capítulo é a implementação da prática, nele serão descritos o ambiente criado para o desenvolvimento do protótipo, as funcionalidades do sistema proposto, a aplicação do ciclo do TDD e a avaliação conforme as métricas definidas anteriormente.

1.9 AMBIENTE DE DESENVOLVIMENTO

Como citado na seção 1.3 Metodologia, o *software* foi desenvolvido na linguagem de programação C# (CSharp). Foi escolhido o *Visual Studio Express for Windows Desktop* 2012 como ambiente de desenvolvimento integrado por causa adaptação com a linguagem, pelo ganho produtivo que essa ferramenta traz para o desenvolvimento e também por se tratar de uma versão *Express* (denominação que a Microsoft® utiliza para sua ferramentas de desenvolvimentos grátis).

Para injeção de dependências[8] foi utilizado um *framework* chamado *Structure Map.* Este *framework* foi selecionado porque seu uso é de extrema facilidade, como é mostrado na imagem (Figura 6) a seguir:

[8] Injeção de dependências é um padrão de desenvolvimento que visa diminuir o acoplamento do código.

```
public static void Inicializar()
{
    ObjectFactory.Initialize(e =>
    {
        e.For<ISession>().Use(GetSessao());

        #region Repositórios
        e.For<IEstadoRepositorio>().Use<EstadoRepositorioNHibernate>();
        e.For<ICidadeRepositorio>().Use<CidadeRepositorioNHibernate>();
        e.For<IPessoaRepositorio>().Use<PessoaRepositorioNHibernate>();
        e.For<IServicoRepositorio>().Use<ServicoRepositorioNHibernate>();
        e.For<IFormaPagamentoRepositorio>().Use<FormaPagamentoRepositorioNHibernate>();
        e.For<ICaixaRepositorio>().Use<CaixaRepositorioNHibernate>();
        e.For<IAtendimentoRepositorio>().Use<AtendimentoRepositorioNHibernate>();
        #endregion

        #region Fachadas
        e.For<IFachadaEstado>().Use<FachadaEstado>();
        e.For<IFachadaCidade>().Use<FachadaCidade>();
        e.For<IFachadaPessoa>().Use<FachadaPessoa>();
        e.For<IFachadaEmpresa>().Use<FachadaEmpresa>();
        e.For<IFachadaCliente>().Use<FachadaCliente>();
        e.For<IFachadaServico>().Use<FachadaServico>();
        e.For<IFachadaFormaPagamento>().Use<FachadaFormaPagamento>();
        e.For<IFachadaCaixa>().Use<FachadaCaixa>();
        e.For<IFachadaAtendimento>().Use<FachadaAtendimento>();
        #endregion
    });
}
```

Figura 6 - Container de Inicialização do Structure Map
Fonte: Elaborada pelo autor

Na Figura 6 é mostrado o método Inicializar(), que como o próprio nome sugere, inicializa o contêiner de injeção de dependências. É nítido a facilidade, para cada interface é definido sua implementação concreta. Para utilizar uma abstração estabelecida no contêiner, basta chamar o método GetInstace<>() da classe estática ObjectFactory. Essa chamada fica de fácil compreensão após observar a Figura 7.

```
public ControladorAtendimento(IVisaoAtendimento visao)
    : base(visao)
{
    this._fachada                = ObjectFactory.GetInstance<IFachadaAtendimento>();
    this._fachadaEmpresa         = ObjectFactory.GetInstance<IFachadaEmpresa>();
    this._fachadaCliente         = ObjectFactory.GetInstance<IFachadaCliente>();
    this._fachadaFormaPagamento = ObjectFactory.GetInstance<IFachadaFormaPagamento>();
    this._fachadaServico         = ObjectFactory.GetInstance<IFachadaServico>();
```

Figura 7 - Obtendo uma abstração
Fonte: Elaborada pelo autor

O *framework Structure Map* injeta as dependências automaticamente, a velocidade no desenvolvimento aumenta consideravelmente, sem contar que ao utilizar essa abordagem programa-se para uma abstração e não para uma implementação. Isto é, aplica-se uma boa prática de programação.

A persistência de dados irá ser feita por meio de um banco de dados relacional. O Sistema de Gerenciamento de Banco de Dados (SGBD) selecionado foi o PostgreSQL. A escolha se dá pelo fato deste SGBD ser gratuito. Acrescenta-se a isso, que este SGBD possui uma comunidade ativa que periodicamente libera atualizações com correções e novos recursos. Além disso, é disponibilizado na página oficial do PostgreSQL a documentação completa para o emprego desta tecnologia no ambiente de produção e acadêmico.

Como a técnica do TDD é voltada para o desenvolvimento com o paradigma de programação orientado a objetos e o banco de dados será construído no modelo relacional, é necessário utilizar uma técnica que reduz a impedância da programação orientada a objetos com o modelo relacional. Essa técnica é conhecida como *Object Relational Mapper* (ORM). Esta técnica se baseia em mapear as classes que serão persistidas no banco de dados de acordo com as tabelas correspondentes. Aplicando essa técnica, o desenvolvedor não se preocupa com a *Structured Query Language* (SQL)[9], a persistência é feita semelhante ao um banco de dados orientado a objetos.

Na Figura 8 é mostrada a persistência em um banco de dados orientado a objetos e a persistência utilizando um *framework* ORM em um banco de dados relacional, respectivamente:

[9] SQL É uma linguagem para de dados relacionais.

GenericoDAO.java

```
1  package dao;
2
3
4  import javax.swing.JOptionPane;
14
15 public class GenericoDAO<T> {
16
17     Banco bd = new Banco();
18     ObjectContainer db4o = bd.db();
19
20     public void inserir(T obj){
21         db4o = bd.db();
22         db4o.store(obj);
23             JOptionPane.showMessageDialog(null, "Cadastro Realizado com Sucesso!");
24     }
```

*GenericoDAO.java

```
1  package dao;
2
3  import java.util.ArrayList;
20
21 public class GenericoDAO<T> {
22
23     private Session session;
24     private Transaction transacao;
25
26     public void inserir(T obj){
27         session = HibernateUtil.getSession();
28         transacao = session.beginTransaction();
29         session.save(obj);
30         transacao.commit();
31     }
32
```

Figura 8 - Comparação da persistência ORM com OO
Elaborada pelo autor

O mapeamento é uma espécie de configuração, que pode ser realizada de várias formas, isso depende do *framework* e da linguagem de programação utilizada. Para este trabalho, foi realizado o mapeamento através de expressões *lambdas*[10], um recurso disponibilizado pelo .NET[11].

Para executar essa estruturação, foi optado utilizar o *framework Fluent NHibernate*, que segundo o próprio desenvolvedor, é uma alternativa simples, rápida e

[10] Uma expressão lambda é uma função anônima que você pode usar para criar tipos de árvore de expressão. Usando expressões lambda, você pode gravar funções locais que podem ser passadas como argumentos ou serem retornadas como o valor de chamadas de função.

[11] .NET é uma iniciativa da Microsoft® que visa uma plataforma única para desenvolvimento. C# e VB.NET são exemplos de linguagens que rodam no .NET.

automatizada para o mapeamento. O *Fluent NHibernate* é baseado no consagrado *framework* ORM de persistência de dados *NHibernate*.

Na Figura 9 é exibido um exemplo de mapeamento por intermédio desse *framework*:

Eimbo.Infraestrutura.Repositorio.NHibernate.Cadastro.Mapeamento.CidadeMap

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Eimbo.Dominio.Comum.Entidade;
using FluentNHibernate.Mapping;

namespace Eimbo.Infraestrutura.Repositorio.NHibernate.Cadastro.Mapeamento
{
    public class CidadeMap :ClassMap<Cidade>
    {
        public CidadeMap()
        {
            this.Table("cidades");

            this.Id(c => c.Id)
                 .Column("id")
                 .Not.Nullable()
                 .Index("idx_cidade_pkey")
                 .GeneratedBy
                 .SequenceIdentity("seq_cidades");

            this.Map(c => c.Nome)
                .Column("nome")
                .Not.Nullable()
                .Length(300);

            this.Map(c => c.Status)
                .Column("status")
                .CustomType(typeof(int))
                .Not.Nullable()
                .Length(1);

            this.References(c => c.Estado)
                            .Column("estado_id")
                            .Not.Nullable()
                            .Index("idx_estado_fkey")
                            .Cascade.Delete()
                            .ForeignKey("cidade_estado_fkey");

        }
    }
}
```

Figura 9 - Mapeamento pelo Fluent NHibernate
Fonte: Elaborada pelo autor

Pode-se ver na Figura 9, a clareza do uso desse *framework.* O desenvolvedor simplesmente deve deixar as propriedades que serão mapeadas da sua classe com a palavra reservada *virtual* e herdar a classe ClassMap na classe de mapeamento

definindo a classe a ser mapeada no objeto genérico. Simplificando, CidadeMap é o mapeamento, ClassMap é a classe herdada do *Fluent NHibernate* e Cidade é a classe a ser mapeada.

O mapeamento de cada propriedade é feito pelo método construtor, efetuando chamadas dos métodos de mapeamento e passando expressões *lambda* por parâmetro. O *Fluent NHibernate* possui uma ótima documentação e suporte no seu repositório oficial no *GitHub*[12]*.*

Os testes automatizados foram feitos por meio do *framework Visual Studio Unit Testing.* Um *framework* que está embutido nas versões mais recentes do *Visual Studio.* Esta ferramenta é bastante semelhante ao *NUnit*[13]*.*

Basta adicionar o *namespace* Microsoft.VisualStudio.TestTools.UnitTesting com a cláusula *using* para ter acesso aos d*ata annotations,* classes e métodos estáticos.

Na Figura 10 é apresentado como é feito a escrita de uma classe de teste utilizando o *Unit Testing*:

[12] *GitHub* é uma plataforma de versionamento de código na nuvem. Ele contém planos comerciais e planos grátis.

[13] *NUnit* é um *framework* conhecido para escrita de testes automatizados na plataforma .NET.

```
namespace Eimbo.Teste.Dominio.ModuloCaixa
{
    [TestClass]
    public class Dado_Uma_FachadaCaixa
    {
        [TestMethod]
        public void Devo_Conseguir_Abrir_Novo_Caixa()
        {
            Decimal saldoAbertura = 150m;

            var caixa = new Mock<Caixa>();
            caixa.Setup(c => c.DataAbertura).Returns(DateTime.Now);
            caixa.Setup(c => c.SaldoAbertura).Returns(150m);
            caixa.Setup(c => c.EstaAberto).Returns(true);

            var repositorioCaixa = new Mock<ICaixaRepositorio>();

            IFachadaCaixa fachada = new FachadaCaixa(repositorioCaixa.Object);

            Assert.IsTrue(fachada.AbrirNovoCaixa(saldoAbertura));

            repositorioCaixa.Verify(r => r.Salvar(It.IsAny<Caixa>()));
        }
```

Figura 10 - Classe de teste com Unit Testing
Fonte: Elaborada pelo autor

Para identificar uma classe de teste, usa-se o *data annotation* [TestClass]. Os métodos de teste são identificados através do [TestMethod] e as asserções são feitas chamando a classe estática Assert. O *framework* possui outros recursos como ExpectedException, que indica que o método deve esperar uma exceção do tipo passado no parâmetro, caso o método não lance essa exceção, ele falhará.

Segundo Freeman e Pryce (2010) deve-se criar instâncias de objetos simulados para auxiliar a escrita do teste. Estes objetos simulados contém expectativas de como eles são chamados e simulam comportamento desconhecido. Segundos os autores, essa forma de implementar TDD ajuda a testar o objeto isoladamente tornando explícita a relação entre o objeto alvo e seu ambiente.

A fim de tornar possível o desenvolvimento fazendo uso de objetos simulados, foi definida a biblioteca Moq. Igualmente ao *Fluent NHibernate*, o Moq adota expressões *lambdas* para configuração de comportamentos, expectativas e verificações (se dado método/propriedade foi chamado durante a execução do objeto simulado).

Na Figura 11 contém um pequeno exemplo do uso dessa biblioteca e como é feito a utilização das expressões *lambdas* por meio do método Setup():

```
[TestMethod]
public void Devo_Conseguir_Efetuar_Uma_Sangria()
{
    var caixa = new Mock<Caixa>();
    caixa.Setup(c => c.EstaAberto).Returns(true);
    caixa.Setup(c => c.EfetuarSangria(It.IsAny<Decimal>()));

    var repositorioCaixa = new Mock<ICaixaRepositorio>();
    repositorioCaixa.Setup(r => r.ObterUltimoCaixaAberto())
                    .Returns(caixa.Object);


    IFachadaCaixa fachada = new FachadaCaixa(repositorioCaixa.Object);

    Assert.IsTrue(fachada.EfetuarSangria(100));
    caixa.Verify(c => c.EfetuarSangria(It.IsAny<Decimal>()));
    repositorioCaixa.Verify(r => r.Salvar(caixa.Object));
}
```

Figura 11 - Aplicando a biblioteca Moq
Fonte: Elaborada pelo autor

Percebe-se pelo nome do método de teste, que dado o objeto alvo do tipo FachadaCaixa deve ser possível realizar uma sangria. FachadaCaixa neste caso é um serviço do domínio, ele faz chamadas em um objeto do tipo Caixa e em um objeto do tipo ICaixaRepositorio. Caixa e ICaixaRepositorio são os objetos que precisam ter os comportamentos simulados.

Então é criado uma instância Mock<Caixa> que ao solicitado a propriedade booleana EstaAberto retornará verdadeiro e que invocará o método EfetuarSangria(), no qual será passado qualquer valor do tipo Decimal. Além disso, é criado uma instância Mock<ICaixaRepositorio>, este que retornará um objeto do tipo Caixa, com os comportamentos definidos anteriormente no Mock<Caixa>, ao fazer a chamada do método ObterUltimoCaixaAberto().

O objeto simulado do tipo ICaixaRepositorio é injetado por construtor na fachada. No final do método de teste é feito uma asserção para ter certeza que a sangria foi realizada, do mesmo modo que são feitas as verificações pela biblioteca Moq para confirmar que os métodos esperados foram utilizados.

Analisando esse exemplo, fica claro abstrair a ideia que os autores londrinos (Freeman e Pryce) passam. O teste especifica o que é esperado do objeto alvo, como é sua relação com os objetos vizinhos e a ferramenta auxilia a implementar o comportamento desconhecido dos objetos vizinhos de maneira descomplicada.

Finalizando a preparação do ambiente, foi escolhida uma ferramenta de versionamento. Dentre tantas opções no mercado como: *Concurrent Version System* (CVS), *Subversion* (SVN), *Mercurial,* Git e etc. Foi preferido usar o *Team Foundation Server* (TFS), dado a excelente integração com o *Visual Studio,* seu gerenciamento pela nuvem (basta ter um cadastro de desenvolvedor na *Microsoft)* e por ser uma ferramenta gratuita. (Para equipes de até cinco desenvolvedores).

O TFS oferece diversos recursos: controle de código, integração com a equipe, testes programados, *feedbacks* e até a aplicação de uma metodologia ágil no projeto como Scrum[14]. O emprego do TFS não vai ser abordado a fundo, pois foi usado apenas o básico para um projeto com apenas um desenvolvedor. Na Figura 12 é mostrada a tela inicial do TFS:

[14] Segundo Junqueira (2012) Scrum é uma metodologia ágil desenvolvida por Jeff Sutherland e sua equipe na década de 1990.

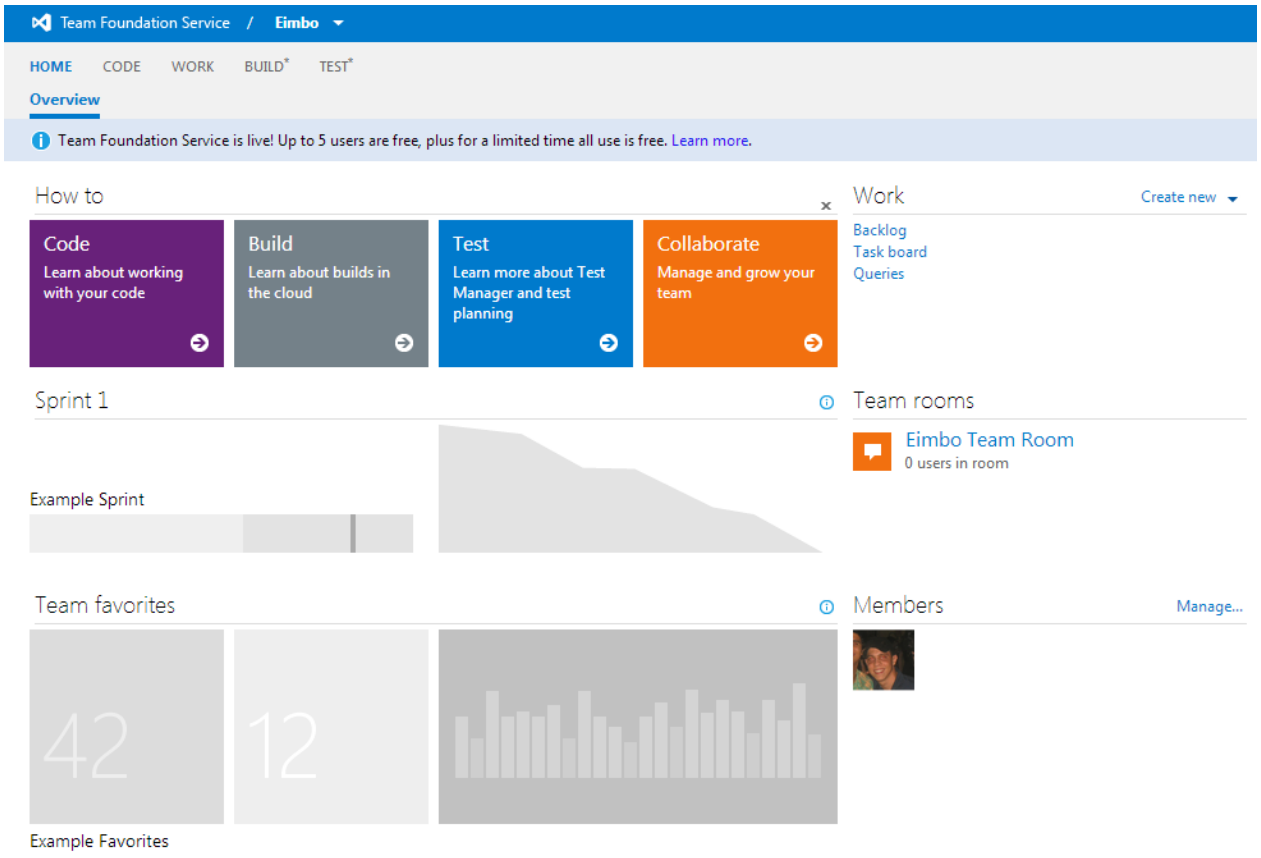

Figura 12 - Tela inicial do TFS
Fonte: Elaborada pelo autor

Observa-se na Figura 12 as ferramentas do TFS supracitadas. A lista de membros no projeto, as salas de conversação do projeto, o código fonte, os testes e os artefatos da metodologia, que neste caso é a Scrum.

1.10 FUNCIONALIDADES DO SOFTWARE PRETENDIDO

O sistema desenvolvido é um sistema para controle gerencial para atendimentos (vendas) para consumidor final denominado Eimbo.[15] Eimbo é um sistema pouco ambicioso, tal qual irá fornecer para o usuário um modesto controle

[15] Nome escolhido pelo autor justamente por não ter significado nenhum e não ter sido encontrado nada referente a este nome nos motores de busca da internet, assim evitando qualquer problema relacionado a plágio ou direitos autorais.

interno e agilidade nos processos de atendimento, controle do fluxo de caixa da empresa e os cadastros necessários para realizar estes processos.

O sistema irá controlar operações básicas de saída e entrada de caixa tais como: abertura e fechamento de caixa, reforços e sangrias em dinheiro, recebimentos de atendimentos em dinheiro, cartão de crédito e cartão de débito. O usuário irá se beneficiar com uma melhoria no processo de recebimento, o troco será calculado automaticamente, poderá ser completado o recebimento com determinado tipo de pagamento. Para cada recebimento será gravado um histórico de recebimentos e saída, caso houver troco.

Para controle de saldos há três opções de relatório com o saldo do caixa. Um chamado saldo do caixa, que mostra sinteticamente apenas o saldo dos três tipos de pagamentos: dinheiro, cartão de crédito e cartão de débito. Outro relatório é intitulado de saldo detalhado, este exibe a movimentação de cada tipo de pagamento, tanto as entradas quanto as saídas. O último relatório de caixa é o extrato do caixa, cujo terá por finalidade informar todas as movimentações do caixa em ordem cronológica, idêntico a um extrato bancário.

Nas próximas três figuras serão mostrados os relatórios citado acima:

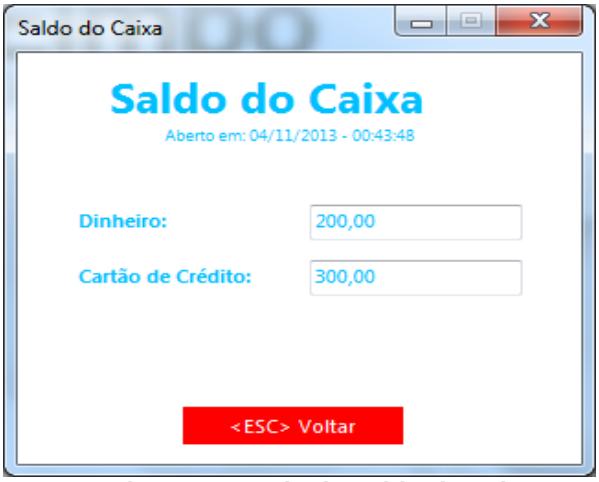

Figura 13 - Tela do saldo do caixa
Fonte: Elaborada pelo autor

Figura 14 - Tela do saldo detalhado do caixa
Fonte: Elaborada pelo autor

Figura 15 - Tela do extrato do caixa
Fonte: Elaborada pelo autor

O caixa contém mais algumas regras de negócio. Estas são:

- Não deve ser possível abrir um caixa com saldo negativo;
- Não pode ser admitido efetuar uma sangria se o valor da sangria for maior que o saldo em dinheiro do caixa;
- Não é permitido abrir um novo caixa, caso exista um caixa anterior aberto;
- Deve ser possível reabrir o último caixa aberto;
- O usuário deverá ser informado se o caixa atual aberto for de um dia diferente do dia atual; e,
- Não é permissível retornar um troco que seja maior que o saldo em dinheiro do caixa.

O módulo de atendimentos é responsável por agilizar o processo de emissão de comandas de vendas. Nele há um campo para o usuário informar a empresa (quem está emitindo o atendimento), o cliente (para quem o atendimento é emitido) e a forma de pagamento que fornece informações para cálculos de descontos, acréscimos e geração de duplicatas. No atendimento o usuário também informa os serviços que foram prestados para o cliente, a quantidade de cada serviço e o seu valor unitário. Todas essas informações: empresa, cliente, forma de pagamento e serviços devem ser previamente cadastrados no módulo de cadastros.

Acrescenta-se no atendimento campos para o usuário digitar o desconto e o acréscimo manualmente, paro o caso do atendimento ter um desconto/acréscimo além do desconto/acréscimo calculado pela forma de pagamento.

Após a entrada destes dados pelo usuário, o sistema calculará os valores totais de cada item e mostrará no rodapé da lista de itens. No qual tem uma somatória da quantidade, valor unitário e valor do item. A cada operação realizada pelo usuário (como adicionar desconto, remover item, alterar acréscimo, incluir serviço) o valor total líquido do atendimento será atualizado e o valor da entrada também (quando a natureza da forma de pagamento permitir).

É exibido na Figura 16 como será o módulo de atendimento para maior compreensão do leitor.

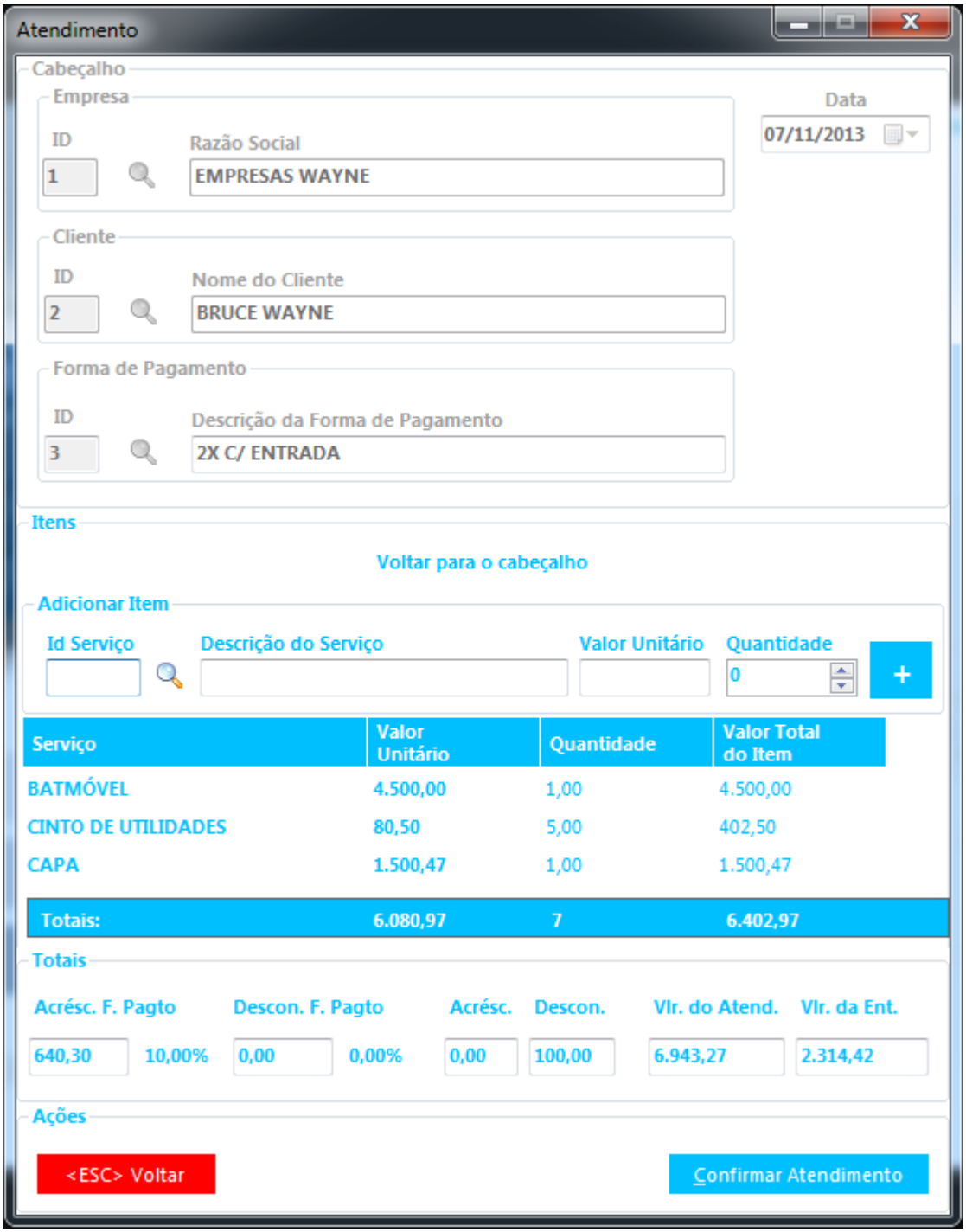

Figura 16 - Tela do Atendimento
Fonte: Elaborada pelo autor

Nota-se pela Figura 16 que o usuário já informou a empresa, o cliente e a forma de pagamento. É necessário informar estes três campos e a data do atendimento para ir para inclusão dos itens. Caso o usuário não informe qualquer um destes itens é exibido

um balão no respectivo campo solicitando o preenchimento do mesmo. Depois de realizado este passo é necessário informar os itens adicionando-os um a um. O valor unitário é obtido por meio do campo preço previamente incluído no cadastro de serviços, porém é possível alterá-lo e adequá-lo conforme a necessidade. Depois é preciso informar a quantidade daquele serviço e clicar no operador mais para adicioná-lo na lista de serviços. Na hipótese do usuário não informar qualquer um destes três itens, o sistema realiza a mesma validação dos campos do cabeçalho, exibindo o balão no controle não informado.

Na parte inferior da tela é apresentado o grupo dos totais. O percentual de acréscimo de forma de pagamento e o percentual de desconto de forma de pagamento são adicionados no cadastro de formas de pagamento. Estes percentuais são recuperados pelo sistema automaticamente e utilizados para realizar o cálculo de porcentagem em cima do valor bruto dos itens (total de quantidades x valores unitários). O desconto e o acréscimo de forma de pagamento não são liberados para o usuário alterar, são apenas informativos.

À direita destes campos, tem os campos de acréscimo e desconto. Os dois podem ser alterados manualmente quando houver necessidade. Para finalizar o módulo de atendimento, há os campos: valor do atendimento e valor da entrada. O valor do atendimento nada mais é que o valor líquido do atendimento (total bruto dos itens - descontos + acréscimos). O valor da entrada é calculado de acordo com a forma de pagamento.

No cadastro de forma de pagamentos há uma opção para selecionar o tipo da forma de pagamento, à vista e a prazo. À vista é o pagamento total do atendimento no recebimento. A prazo é calculado de acordo com as parcelas e se há entrada ou não. Se houver entrada, então é calculado o valor do atendimento dividido pelas parcelas, a primeira parcela será a entrada que será recebida no ato de finalização do recebimento.

A seguir serão listadas as regras de negócio do módulo de atendimento para proporcionar a montagem do cenário para a escrita dos testes:

- Deve ser possível criar um atendimento válido (com data, empresa, cliente e forma de pagamento válidos);

- Deve ser possível adicionar um item válido (com serviço, valor unitário e quantidade válidos);
- Deve ser possível remover um item adicionado;
- Deve ser possível obter os valores do atendimento. valor total, valor de entrada, totais da lista de itens;
- Ao adicionar um item que já foi adicionado, o mesmo deve ser sobrescrito;
- Ao movimentar um item, isto é, adicionar ou remover; os valores do atendimento devem ser recalculados e atualizados;
- Deve ser possível dar desconto e acréscimo manualmente e atualizar os valores do atendimento;
- Não deve ser possível validar um atendimento com valor zerado;
- Não deve ser possível validar um atendimento sem itens;

Dado a circunstância do atendimento ter valor de entrada, no momento que o usuário confirmar o atendimento, é mostrado uma tela para o recebimento em caixa desta entrada.

Nessa tela é exibido o valor a receber - este é o valor da entrada calculado anteriormente – e contém três campos para digitar a quantidade que será recebida em cada tipo de pagamento. Ao alterar cada um dos três campos, o sistema informa em um quarto campo a soma dos valores incluídos nos três tipos de pagamento. O restante a ser recebido também é modificado automaticamente, pois o restante é o saldo de: valor a receber menos a soma dos valores informados para cada tipo de pagamento. Se a soma dos valores for maior que o valor a ser recebido, então o restante é alterado para troco, para facilitar o entendimento do usuário do que precisa ser devolvido para o cliente.

Adiante, será exibida na Figura 17 a tela do recebimento do atendimento com o valor restante não informado integralmente e a seguir na Figura 18, a tela do recebimento com o valor da soma maior que o valor a receber:

Figura 17 - Tela de Recebimento com valor restante
Fonte: Elaborada pelo autor

Figura 18 - Tela de Recebimento com troco
Fonte: Elaborada pelo autor

Para finalizar as funcionalidades do recebimento, o usuário poderá informar o valor restante no tipo de pagamento que ele deseja para completar o valor restante com esse respectivo tipo de pagamento. Isso é realizado por meio do botão R, situado à direita do respectivo campo de tipo de pagamento (Figura 17e Figura 18).

Como foi feito nas etapas anteriores, serão elencadas as funcionalidades do recebimento de atendimento:

- Deve ser possível obter o valor restante ao calcular o recebimento;
- Deve obter valor restante zerado, se a soma for maior que o valor a receber;
- Deve obter o valor do troco, se a soma for maior que o valor a receber;
- Deve ser possível receber o valor restante em dinheiro;
- Deve ser possível receber o valor restante com cartão de crédito;
- Deve ser possível receber o valor restante com cartão de débito;
- Não deve ser possível receber se o valor restante for maior que zero;
- Não deve ser possível receber se o valor do troco for maior que o saldo em dinheiro do caixa;
- Não deve ser possível receber se não houver nenhum caixa aberto; e,
- Deve ser possível se a transação for válida (se não satisfez todas as funcionalidades do recebimento)

Para fornecedor os dados cadastrais necessários para efetuar os processos dos módulos supracitados, o sistema precisa conter as informações previamente cadastradas. O módulo de cadastros tem os cadastros precedentemente citados e mais dois adicionais: estado e cidade.

O endereço do cadastro de cliente e empresa é composto por uma cidade e cada cidade é composta por um estado. Podem ser cadastrados diversos endereços, basta informar seu tipo. Os tipos do endereço são: residencial, comercial e cobrança. Somam-se a isso, a inclusão dinâmica de documentos e telefones. Documentos serão adicionados da mesma maneira que o endereço basta informar o valor do documento e seu tipo. Os tipos de documento são: Registro Geral (RG), Cadastro de Pessoas Físicas (CPF), Inscrição Estadual (IE), Cadastro Nacional de Pessoas Jurídicas (CNPJ).

Os telefones também serão adicionados dessa forma, os tipos do telefone são: residencial, comercial, celular, fax.

O nome do cliente e razão social da empresa juntamente com data de aniversário e data de abertura da empresa são dados necessários para o cadastro de cliente e empresa.

Na Figura 19 é mostrada como será o cadastro de clientes, o cadastro de empresas é semelhante, ao não ser pela nomenclatura de alguns campos.

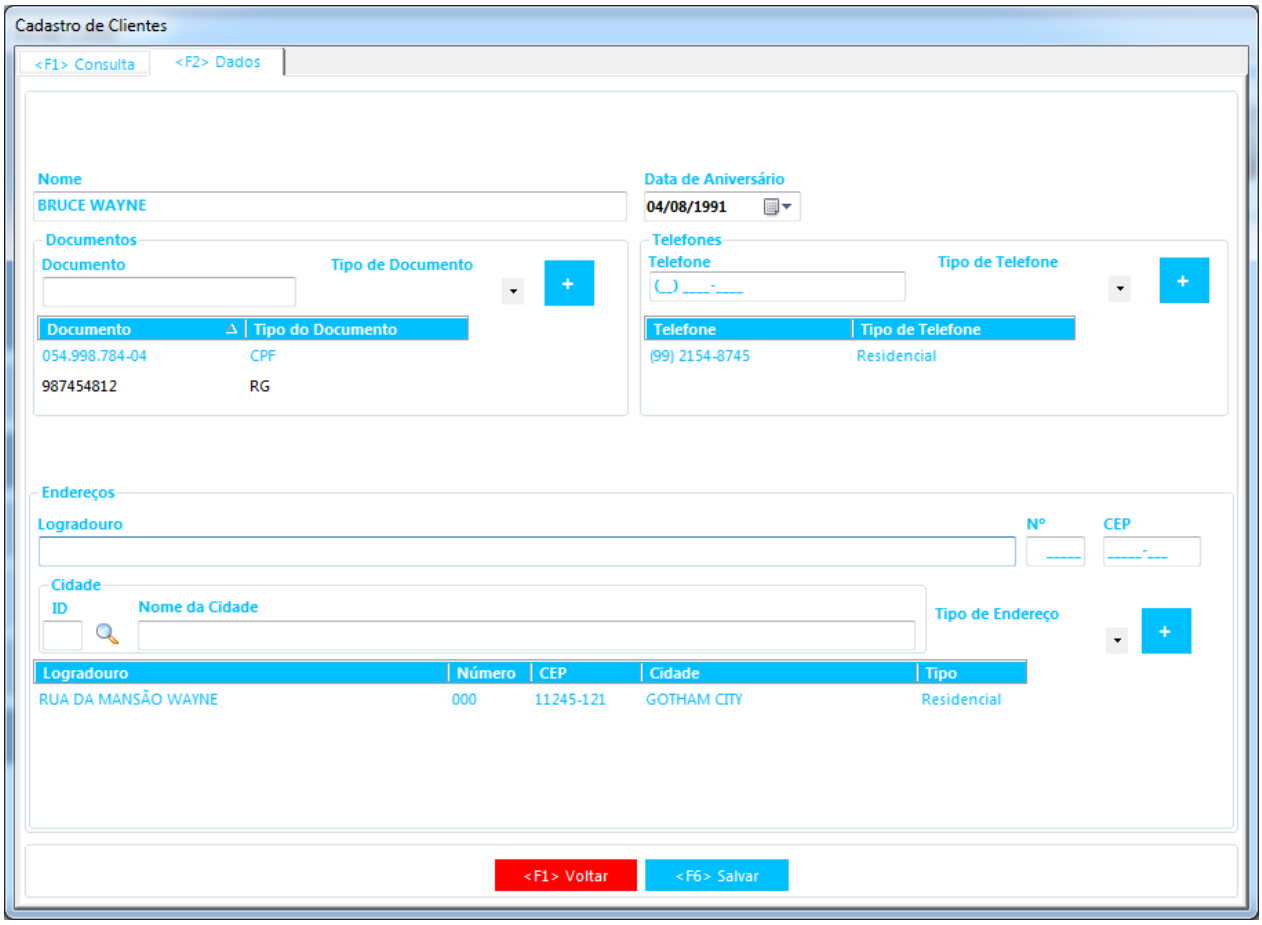

Figura 19 - Tela de cadastro de clientes
Fonte: Elaborada pelo autor

Abaixo será exibida a lista com as funcionalidades do cadastro de cliente:

- É obrigatório informar o nome, a data de aniversário, o CPF, um telefone e um endereço;

- Não deve ser possível cadastrar um cliente com o mesmo CPF de outro cliente cadastrado no banco de dados; e,
- Deve ser possível remover os documentos, endereços e telefones.

  E a lista com as funcionalidades do cadastro de empresa:
- É obrigatório informar o nome, a data de abertura da empresa, o CNPJ, um telefone e um endereço;
- Não deve ser possível adicionar um telefone residencial;
- Não deve ser possível adicionar um endereço residencial;
- Não deve ser possível cadastrar uma empresa com o mesmo CNPJ de outra empresa cadastrada no banco de dados; e,
- Deve ser possível remover os documentos, endereços e telefones.

No cadastro de serviço há poucas regras de negócio. O usuário apenas inclui o nome do serviço e o valor padrão que é cobrado para esse serviço. O sistema verifica automaticamente se o nome do serviço já está cadastrado para outro serviço, caso esteja, impedirá o processo de ser finalizado.

As funcionalidades do cadastro de serviço são:

- É obrigatório informar o nome e um valor, e,
- Não deve ser possível cadastrar um serviço com o mesmo nome de outro serviço previamente cadastrado no banco de dados;

No cadastro de formas de pagamentos existe uma peculiaridade. Neste cadastro tem o campo de descrição, o percentual de acréscimo e desconto – que é usado no cálculo do valor líquido do atendimento – e o tipo da forma de pagamento. Caso a forma de pagamento seja a prazo, então é exibido um painel para o para o usuário informar as opções de parcelamento.

O usuário seleciona a forma de pagamento, a quantidade de parcelas e o intervalo (em dias) entre as parcelas. Como podemos observar na Figura 20:

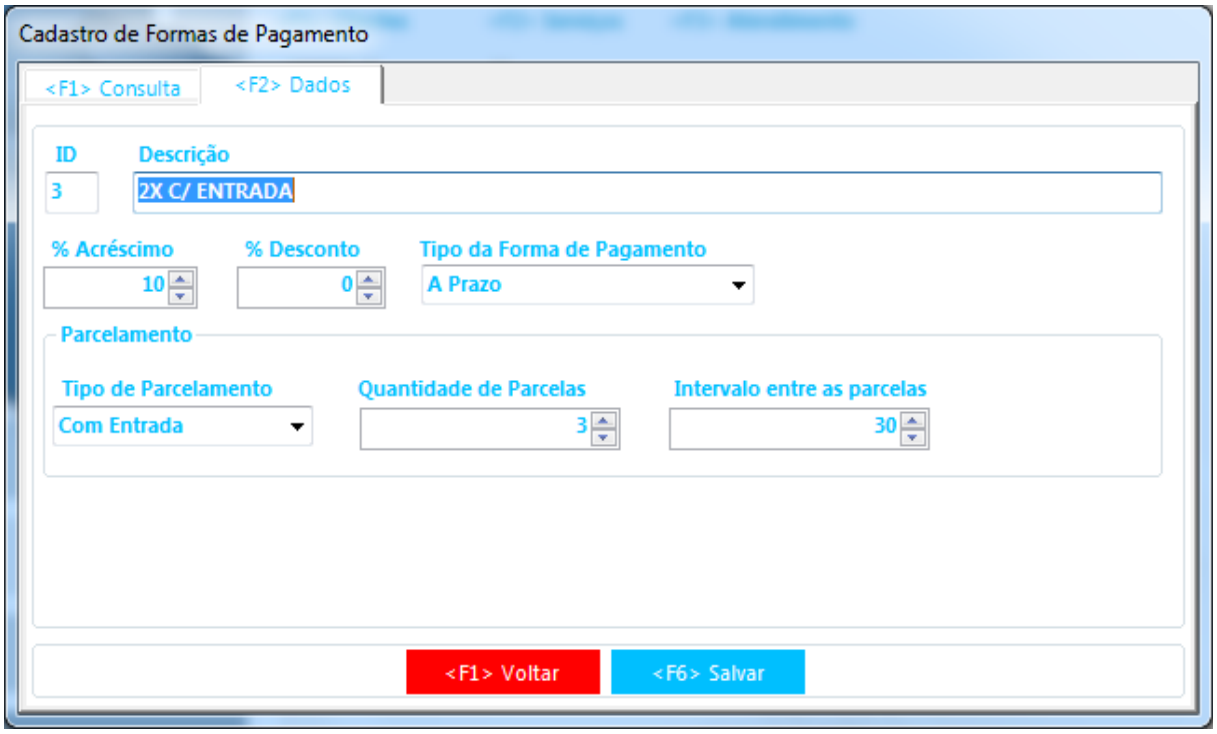

Figura 20 - Tela de cadastro de forma de pagamento
Fonte: Elaborada pelo autor

As funcionalidades deste cadastro são:

- É obrigatório informar a descrição e o tipo da forma de pagamento;
- Não deve ser possível incluir percentual de acréscimo ou desconto negativo;
- Se a forma de pagamento for a prazo, o tipo do parcelamento e a quantidade de parcela são obrigatórios;
- Não deve ser possível a quantidade de parcelas e o intervalo entre as parcelas ser negativo (para forma de pagamento a prazo); e,
- Se a forma de pagamento for a prazo e tiver entrada, então a quantidade de parcelas deve ser maior que 1 (para esse caso, o certo é cadastrar a forma de pagamento com tipo à vista).

Todos os cadastros tem a tela do mesmo formato por questão de padronização e reaproveitamento de leiaute. Eles são separados por duas abas principais, a segunda contendo as informações do registro – também utilizada para incluir um novo – foi exibida em várias figuras anteriores. Na primeira há uma listagem dos registros do

cadastro em questão, um campo para se buscar por qualquer valor dos campos da listagem, botões para incluir, alterar e uma opção para bloquear o registro, como é mostrada por intermédio da Figura 21:

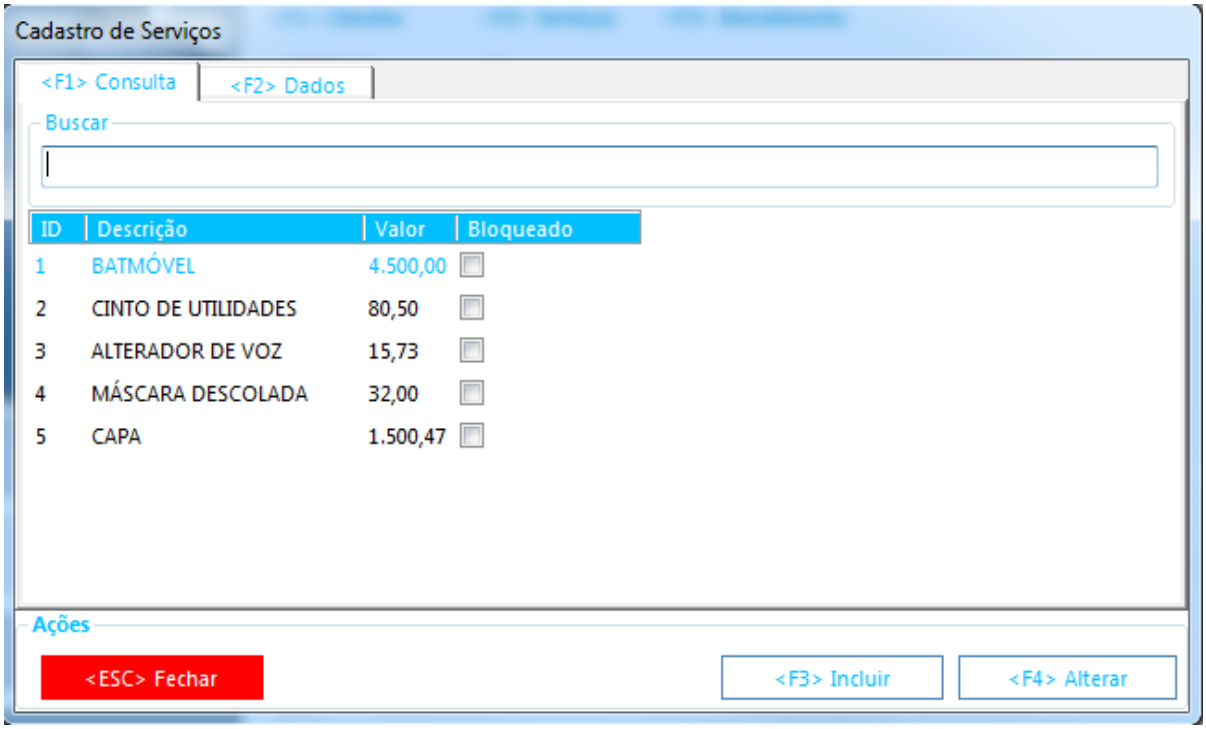

Figura 21 - Aba de consulta dos cadastros
Fonte: Elaborada pelo autor

Para bloquear um registro, o usuário só precisa dar um clique duplo na respectiva caixa de verificação do registro na coluna bloqueado. Para efetuar o desbloqueio o processo é o mesmo.

Em algumas ocasiões, o usuário necessita buscar algum registro cadastrado no banco de dados. Observa-se isso, analisando a tela de atendimento (Figura 16), na qual o usuário precisa informar o cliente, forma de pagamento e etc. Para estes casos, as telas de cadastros se comportam de uma maneira diferente. É permitido para usuário apenas visualizar os registros, buscar, olhar as informações detalhadas na aba de dados e selecionar aquele que necessita. Portanto, nas situações de busca, as telas de cadastros não exibem para o usuário as opções de persistência de dados (alterar, incluir, bloquear). Pode-se ver esse comportamento por meio da Figura 22:

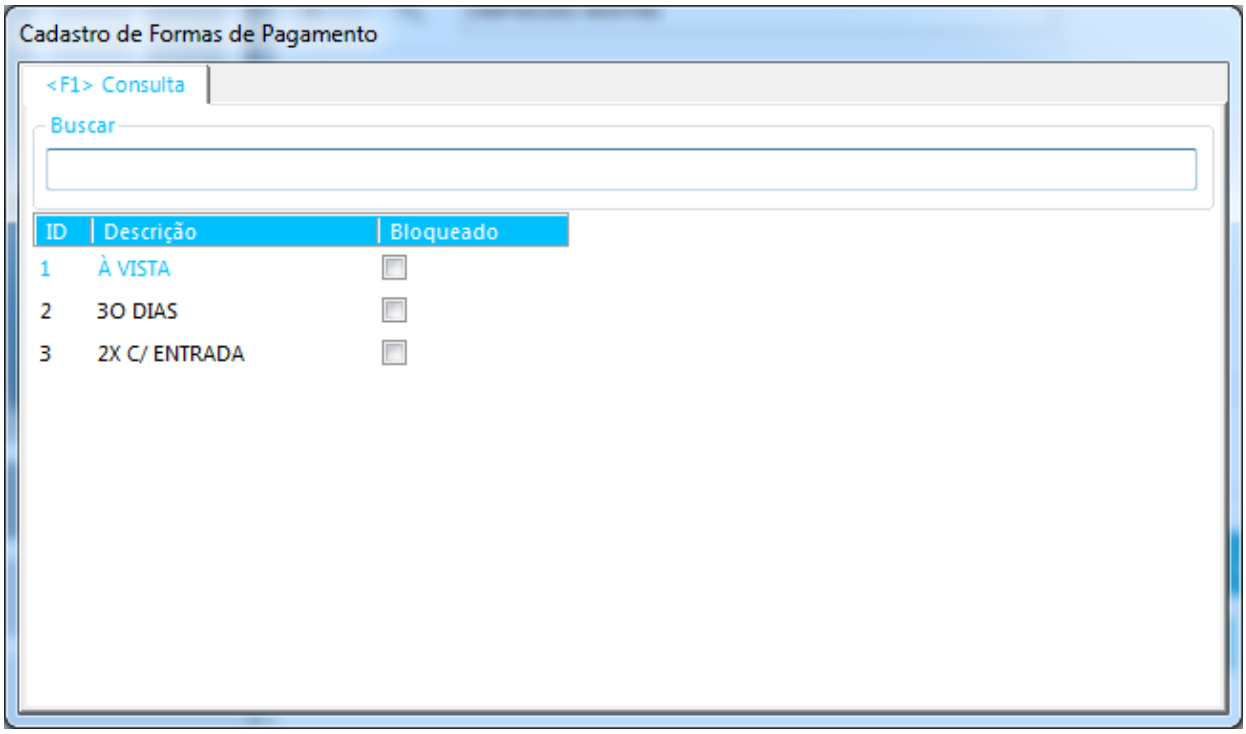

Figura 22 - Tela de cadastro no modo de busca
Fonte: Elaborada pelo autor

1.11 IMPLEMENTAÇÃO

Foi decidido implementar o sistema utilizando uma arquitetura com quatro camadas lógicas: aplicativo, domínio, infraestrutura e UI. De acordo com Evans (2003), a utilização dessa arquitetura, garante que cada camada se especialize em determinado aspecto do programa. Cada camada depende apenas dos elementos “abaixo” dela. Essa implementação permite sistemas mais coesos e facilita a interpretação do design do software.

A técnica do TDD foi aplicada apenas na camada de domínio, ela foi utilizada para guiar o desenvolvimento e obter um modelo de domínio consistente que atenda os requisitos do sistema, em adição, uma vez que os testes estão implementados e validados, os testes de regressão garantem que qualquer alteração no código existente não quebre recursos que estão funcionais. Em relação às outras camadas, será descrito uma breve explicação sobre cada uma, qual a finalidade de cada e como elas interagem entre si.

A camada de aplicativo recebe as requisições interpretadas pela camada de interface com o usuário e direcionar para determinada área do domínio, para que o domínio resolva o problema. Os controladores e o contêiner de injeção de dependências ficam situados nessa camada.

UI é a camada responsável por interpretar as entradas de dados do usuário e mostrar as informações de saída para o usuário. É nesta camada que estão às telas do sistema.

Os repositórios do *NHibernate* e os mapeamentos do *Fluent NHibernate* pertencem a camada de infraestrutura. Esta camada é responsável por conter relações com bibliotecas e *frameworks* de terceiros.

Finalmente a camada de domínio, na qual, neste trabalho, foi aplicada a prática do TDD. Esta camada é responsável por toda a regra de negócio do software. Evans (2003), afirma que essa camada é o “coração do software”.

A implementação do ciclo do TDD foi realizada para guiar todo o desenvolvimento da camada de domínio do sistema. O código fonte completo da solução é encontrada nos anexos deste trabalho, por causa de seu imenso tamanho, fica inviável a exibição e explanação detalhadamente.

Foi seguido o ciclo do TDD para implementar cada funcionalidade, abaixo (na Figura 23) segue um exemplo da escrita de um teste para um objeto do tipo FormaPagamento.

```
[TestMethod]
[ExpectedException(typeof(ExcecaoQuantidadeParcelasInvalidaParaFormaPagamentoComEntrada))]
public void Nao_Devo_Conseguir_Criar_Uma_Forma_Pagamento_A_Prazo_Com_Entrada_E_Apenas_Uma_Parcela()
{
    Int16 quantidadeDeParcelas = 1;
    Int16 intervaloEntreParcelas = 30;

    ParcelamentoFormaPagamento parcelamento = new ParcelamentoFormaPagamento(TipoParcelamentoFormaPagamento.ComEntrada,
                                                                             quantidadeDeParcelas,
                                                                             intervaloEntreParcelas);
    FormaPagamento formaPagamento = new FormaPagamento("30 dias",
                                                       TipoFormaPagamento.Prazo,
                                                       0,
                                                       0,
                                                       parcelamento);
}
```

Figura 23 - Teste para um objeto FormaPagamento
Fonte: Elaborada pelo autor

Observa-se por este teste, a aplicação do ciclo do TDD. O nome do teste é parecido com uma das funcionalidades do cadastro de forma de pagamento. Quando o teste foi escrito, precisava-se alcançar aquela funcionalidade. Então com o *ExpectedException* foi definido que era esperado a exceção ExcecaoQuantidadeParcelasInvalidaParaFormaPagamentoComEntrada. No corpo do teste foi escrito em qual circunstância essa exceção deveria ser lançada.

Neste processo foi escrito o primeiro passo do ciclo do TDD: escreva um teste que falha. O primeiro erro foi o de compilação, porque ainda não havia sido implementado a classe da exceção esperada e a classe FormaPagamento. Foram implementadas as classes para executar a compilação, porém o teste falhou, devido ao fato dessa regra de negócio ainda não ter sido implementada na classe FormaPagamento.

Então foi feito o segundo passo do ciclo: implemente o mais simples possível para o teste passar. Na classe FormaPagamento foi criado uma verificação, no caso do tipo da forma de pagamento ser a prazo, com entrada e apenas uma parcela; essa exceção ExcecaoQuantidadeParcelasInvalidaParaFormaPagamentoComEntrada seria lançada. O teste foi executado com sucesso. Apesar desta funcionalidade estar garantida, era necessário deixar o código na classe FormaPagamento mais simples, para melhorar a legibilidade do código fonte. Baseado nesse objetivo, foi efetuado o último passo do ciclo: a refatoração.

A regra de negócio foi adicionada no método privado set da propriedade Parcela, tornando o código mais amigável e organizado. Na Figura 24 pode-se observar esse fato:

```
public virtual ParcelamentoFormaPagamento Parcelamento
{
    get
    {
        try
        {
            return this._parcelamentos.First();
        }
        catch (Exception)
        {
            return null;
        }
    }
    set
    {
        if (value == null)
            throw new ExcecaoParametroInvalido("Parcelamento");

        if (TipoFormaPagamento.Vista.Equals(this.Tipo))
            throw new ExcecaoCampoNaoPermitido("parcelamento", "forma de pagamento à vista");

        if ((!value.Tipo.Equals(TipoParcelamentoFormaPagamento.ComEntrada)) && (!value.Tipo.Equals(TipoParcelamentoFormaPagamento.SemEntrada)))
            throw new ExcecaoParametroInvalido("TipoParcelamento");

        if (value.IntervaloEntreParcelas <= 0)
            throw new ExcecaoParametroInvalido("IntervaloParcelas");

        if (value.QuantidadeParcelas <= 0)
            throw new ExcecaoParametroInvalido("QuantidadeParcelas");

        if ((value.QuantidadeParcelas < 2) && (TipoParcelamentoFormaPagamento.ComEntrada.Equals(value.Tipo)))
            throw new ExcecaoQuantidadeParcelasInvalidaParaFormaPagamentoComEntrada();

        if (this._parcelamentos == null)
            this._parcelamentos = new List<ParcelamentoFormaPagamento>();

        this._parcelamentos.Clear();

        this._parcelamentos.Add(value);
    }
}
```

Figura 24 - Regra de negócio refatorada em FormaPagamento
Fonte: Elaborada pelo autor

Nessa perspectiva do exemplo da FormaPagamento, nota-se que TDD guia o *design* do modelo. Foi criada uma exceção que deixa explícita uma regra de negócio. Ou seja, a implementação e os requisitos “falam uma só língua”.

Em seguida, será mostrado como a escrita do teste antes da implementação torna claro as relações do objeto. Ao escrever o método de teste Devo_Conseguir_Adicionar_Itens_Validos(), fica nítido que dado um objeto do tipo Atendimento; pode ser feito a inclusão de um serviço, com valor unitário e quantidade. No entanto, a classe Servico possuía validações que dificultavam sua instanciação. Neste momento é que os objetos simulados são úteis. Observe a Figura 25:

```
[TestMethod]
public void Devo_Conseguir_Adicionar_Itens_Validos()
{
    var servico1 = new Mock<Servico>();
    servico1.Setup(s => s.Id)
            .Returns(1);

    var servico2 = new Mock<Servico>();
    servico2.Setup(s => s.Id)
            .Returns(2);

    Atendimento atendimento = new Atendimento(DateTime.Now, this._empresaValida, this._clienteValido, this._fpagtoValida);

    atendimento.AdicionarItem(servico1.Object, 1m, 1);
    atendimento.AdicionarItem(servico2.Object, 2m, 2);

    int contador = 1;

    foreach(ItemAtendimento i in atendimento.Itens)
    {
        switch (contador)
        {
            // Servico 1
            case 1: Assert.AreEqual(1, i.Servico.Id);
                    Assert.AreEqual(1m, i.ValorUnitario);
                    Assert.AreEqual(1, i.Quantidade);
                    break;
            // Servico 2
            case 2: Assert.AreEqual(2, i.Servico.Id);
                    Assert.AreEqual(2m, i.ValorUnitario);
                    Assert.AreEqual(2, i.Quantidade);
                    break;
        }

        contador++;
    }
}
```

Figura 25 - Teste que torna as relações do objeto explícitas
Fonte: Elaborada pelo autor

Foi utilizado o *framework* Moq para criar dois objetos simulados do tipo serviço. Foi passado na declaração do objeto genérico do método construtor Mock<>() qual tipo de objeto era preciso simular. Logo após, foi configurado as propriedades necessárias para realizações das asserções que validarão o teste. Foi finalizado o ciclo do TDD para esse teste, garantindo a funcionalidade.

O ciclo do TDD foi efetuado para todas as regras de negócio do sistema, para cada módulo com os testes finalizados foram implementados os mapeamentos e realizados os testes de sistema, rodando a aplicação e testando manualmente.

Não foram utilizados testes automatizados para a infraestrutura de persistência, pois como o *NHibernate* é um *framework* com uma grande comunidade ativa – que garantem a qualidade do *framework* - e as operações *Create Restore Update Delete* (CRUD) eram relativamente simples, não se aplica garantir tal cobertura de testes.

Ao final da implementação do sistema, a suíte de testes contava com 160 testes (Figura 26). Além do processo de TDD ter garantido uma modelagem de domínio fracamente acoplada e bastante coesa, o sistema contava com uma completa suíte de testes de regressão.

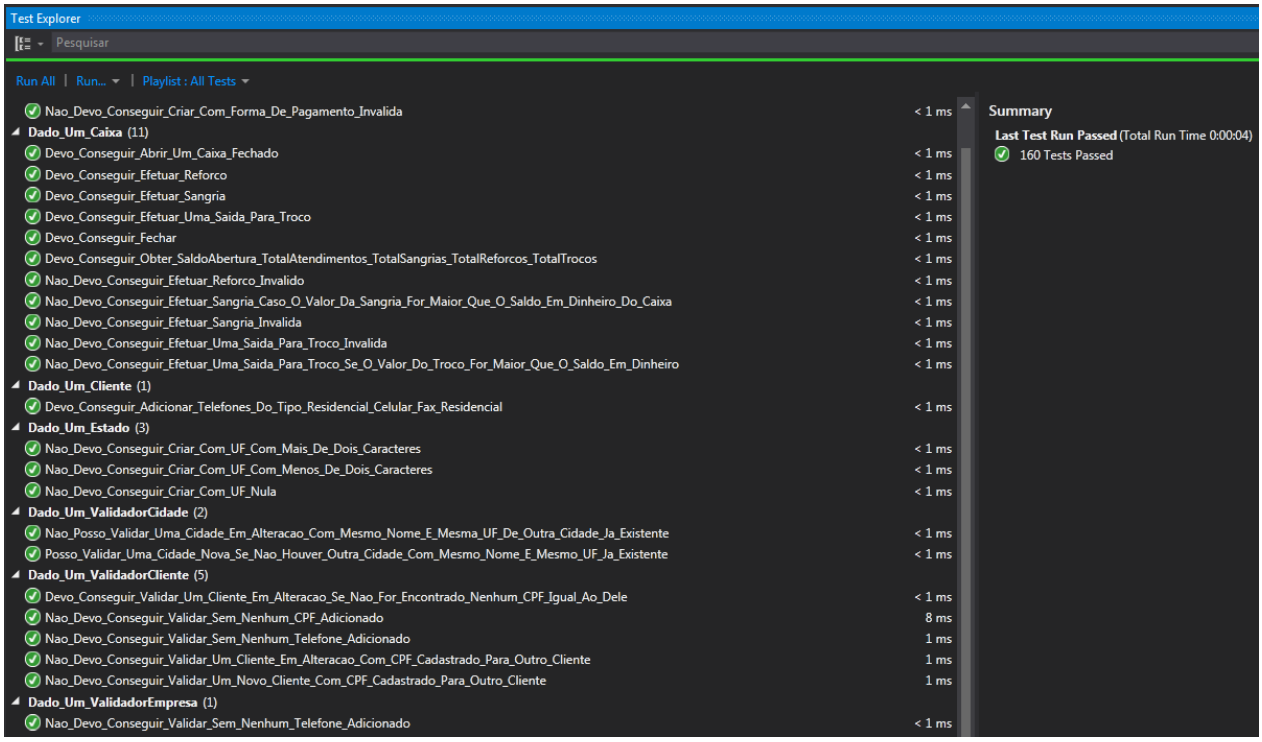

Figura 26 - Suíte de testes finalizada
Fonte: Elaborada pelo autor

1.12 AVALIAÇÃO

Nesse capítulo será avaliado o emprego da técnica TDD no sistema desenvolvido. A avaliação vai ser realizada utilizando o conceito das métricas definidas no início do trabalho. O autor dará um breve *feedback* de como foi utilizar a prática, se na visão dele, obteve vantagens do método tradicional de desenvolver software.

1.12.1 Escopo Do Projeto

Ao utilizar a prática do TDD não foi notada grande diferença em relação ao método tradicional para se definir o escopo do projeto. Isso é devido ao fato do cliente ser o próprio desenvolvedor, portanto os requerimentos dos projetos foram bem definidos e não sofreram grandes alterações. Outrora nos casos que as alterações nos requerimentos ocorreram – como na implementação do recebimento de atendimento, cujo tipo do pagamento era apenas dinheiro e depois foram incluídos cartão de crédito e débito – a técnica mostrou algumas vantagens.

Com a suíte de testes de regressão bem definida, a escrita de novos testes e implementação dos mesmos com novas funcionalidades, fizeram testes anteriores que estavam passando, não passarem. Ou seja, foi antecipado um erro no *software* no momento do desenvolvimento, antes de ser colocado em produção.

1.12.2 Realizar o Controle de Qualidade

Segundo o guia PMBOK®, a identificação dos padrões de qualidade e como realizar a garantia da qualidade, é definida por quem desenvolve o produto. Pode-se basear em um modelo de normas técnicas mundialmente conhecidas para estabelecer estes padrões ou defini-los. A mesma regra vale para aplicar o controle de qualidade. Pode-se utilizar de ferramentas e métodos globalmente reconhecidos e indicados pelo guia, ou aplicar as próprias técnicas definidas.

Dado a baixa complexidade do *software* desenvolvido, o método definido para garantir a qualidade do sistema foi realizar testes de sistema, verificando se as funcionalidades propostas foram alcançadas. Logo após os testes serem realizados e a qualidade pretendida ter sido alcançada, o autor argumenta que para essa métrica o emprego da prática se tornou eficaz.

Como em um projeto que foi utilizado TDD toda implementação é feita para realizar um teste de aceitação passar e ao momento que o teste está validado, é garantido que a funcionalidade foi alcançada com sucesso, aplicar TDD evita o trabalho desnecessário realizado com frequência por desenvolvedores, aumentando a complexidade em uma implementação. Isto é, aplicando o ciclo do TDD corretamente, o alcance dos requisitos é feito de forma direta e objetiva.

1.12.3 Tempo

O gerenciamento de tempo para esse projeto foi feito da mesma forma que seria em um projeto tradicional. Como o TDD é uma prática de desenvolvimento recomendada pela comunidade do desenvolvimento ágil, para avaliar melhor o gerenciamento do tempo, o correto seria avaliar esta forma de desenvolvimento em conjunto com alguma metodologia ágil. Portanto, não se aplica utilizar essa métrica para avaliar a técnica do TDD de forma isolada.

1.12.4 Riscos

Partindo da perspectiva que um risco de projeto é um evento que causa uma alteração incerta em pelo menos um objetivo do projeto. Por meio da experiência obtida durante o desenvolvimento do trabalho, o autor alega que TDD diminui os riscos do projeto.

Seja garantindo a qualidade externa do *software*, no qual os requisitos definidos no escopo foram alcançados; quanto na qualidade interna, cujo modelo de *design* ficou de fácil entendimento - permitindo a manutenibilidade com facilidade em caso de

alguma alteração de requisitos - evitando longos períodos de análise do código, período que pode ser utilizado para entregar o projeto no prazo. Sem contar a suíte de testes de regressão que diminuem o risco de uma versão ir para a produção com erros.

5. CONCLUSÃO

O objetivo do trabalho foi aplicar a técnica do desenvolvimento guiado por testes em um hipotético sistema de vendas. Para concluir esse objetivo, foi estudada a técnica profundamente, analisando pontos de vista de diversos autores e procurado assimilar como a técnica deveria ser empregada. Além disso, foram analisados diversos exemplos de códigos fonte disponíveis na internet. Foi constatado que TDD é uma técnica que guia desenvolvimento e não uma forma de aplicar os testes.

Identificou-se a necessidade de estudar as ferramentas para colocar o conceito de TDD em prática, o que era necessário para realizar o ciclo guiado por teste de maneira eficiente e eficaz. Percebeu-se que era preciso utilizar os *frameworks e* bibliotecas como o Moq, para auxiliar na escrita de objetos simulados.

Após os conceitos terem sido abstraídos e as ferramentas necessárias selecionadas, foram definidos as funcionalidades pretendidas do *software* proposto. Foi uma tarefa simples de realizar, dado que o próprio usuário também era o desenvolvedor.

Decidiu-se então, colocar os conceitos em prática aplicando o ciclo para obter as funcionalidades especificadas. Foi uma tarefa complicada, soava estranho escrever um teste antes de uma implementação. Criar o cenário contendo a funcionalidade pretendida era a maior dificuldade, porém depois de bastante persistência a implementação começou a fluir. Em seguida, adquirida certa experiência por parte do autor, as vantagens do emprego da técnica começaram a surgir.

A simplicidade obtida com o código implementado aplicando o ciclo do TDD era algo surpreendente para o autor, visto que não era mais necessário utilizar rotinas complexas para implementar algo simples. A produtividade aplicando TDD se mostrou igualmente surpreendente. Ao contrário das expectativas, o tempo gasto para implementar algo novo foi drasticamente reduzido, pois acrescentava-se mais tempo escrevendo o teste, porém gastava muito menos tempo depurando o código. Depois que o emprego do ciclo começou a ser feita de maneira correta, a depuração do código foi praticamente nula.

A técnica do TDD comprovou realmente sua utilidade no domínio do sistema desenvolvido. Precauções devem ser tomadas na escrita do teste; afinal, se o teste estiver errado a implementação também estará.

6. TRABALHOS FUTUROS

Ao final do trabalho, são identificadas propostas de trabalhos futuros para aprimorar a pesquisa. Entre elas estão:

- Implementar o mesmo domínio do protótipo proposto com a maneira de desenvolvimento tradicional e comparar as abordagens, verificando se TDD obtém vantagens em relação a abordagem tradicional;
- Verificar se é possível a implementação de TDD em uma plataforma *mobile* e realizá-la, caso for possível; e,
- Implementar TDD um modelo de negócio complexo para verificar sua aplicabilidade e eficácia, isto poderia ser feito em um sistema legado, migrando-o aos poucos.

REFERÊNCIAS

ANICHE, Maurício. **Test-Driven Development:** Teste e Design no Mundo Real. São Paulo: Casa do Código, 2012. 160 p.

Begel, Andrew e Beth Simon, **Struggles of New College Graduates in Their First Software Development Job.** In: SIGCSE Bulletin, 40, nº1 (Março de 2008); 226-230, ACM, ISSN 0097-8418.

EVANS, Eric. **Domain-Driven Design:** Atacando as Complexidades no Coração do Software. 2. ed. Rio de Janeiro: Alta Books, 2003. 493 p.

FOWLER, Martin. **Refatoração:** Aperfeiçoando o Projeto de Código Existente. São Paulo: Bookman, 2004. 366 p.

FREEMAN, Steve; PRYCE, Nat. **Desenvolvimento de Software Orientado a Objetos, Guiado por Testes**. Rio de Janeiro: Alta Books, 2010. 384 p.

GAMMA, Erich et al. **Design Patterns:** Elements of Reusable Object-Oriented Software: Addison-wesley, 1994. 350 p.

JUNQUEIRA, Bruno. **Estudo Comparativo entre Métodos Ágeis e Tradicionais de Fabricação de Software**. 2012. 41p. Trabalho de Conclusão de Curso (Graduação em Engenharia da Computação) - UPE - Universidade de Pernambuco, Recife, 2012.

Management Body of Knowledge – **PMBOK Guide** – 2000 Edition. PMI, 2003.

PMBOK- **Construction. Construction**, Extension to A Guide to the Project

PMBOK®: **Um Guia do Conjunto de Conhecimentos em Gerenciamento de Projetos**, Terceira edição, 2004. 405p.Uma Norma Nacional Americana ANSI/PMI 99-001-2004.

PRESSMAN, R. **Engenharia de Software** McGraw-Hill, (2001)

SANTOS, Rafael. **Emprego do Test Driven Development no desenvolvimento de aplicações**. 2010. 123p. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - UnB - Universidade de Brasília, Brasília, 2010.

THIBODEAU, Patrick Thibodeau. **Study: Buggy software costs users, vendors nearly 60b annually. Computer World**, jan. 2002. Disponível em: <http://www.computerworld.com/s/article/72245/Study_Buggy_software_costs_users_vendors_nearly_60B_annually>. Acesso em: 11 jun. 2013