JULIO CÉSAR MARQUES FARAH

INFLUÊNCIAS DA ABORDAGEM TDD NA QUALIDADE DE CÓDIGO-FONTE

LAVRAS – MG

2013

JULIO CÉSAR MARQUES FARAH

INFLUÊNCIAS DA ABORDAGEM TDD NA QUALIDADE DE CÓDIGO-FONTE

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Sistemas de Informação, área de Engenharia de *Software*, para obtenção do título de Bacharel em "Sistemas de Informação"

Orientador
Prof. Dr. Cristiano Leite Castro

LAVRAS – MG

2013

JULIO CÉSAR MARQUES FARAH

INFLUÊNCIAS DA ABORDAGEM TDD NA QUALIDADE DE CÓDIGO-FONTE

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Sistemas de Informação, área de Engenharia de *Software*, para obtenção do título de Bacharel em "Sistemas de Informação"

APROVADA em 19/08/2013.

Prof. Dr. André Pimenta Freire	UFLA
Prof. M. Sc. André Grützmann	UFLA

Prof. Dr. Cristiano Leite Castro

(Orientador)

LAVRAS – MG

2013

AGRADECIMENTOS

Dentre todas as fases do trabalho de conclusão de curso (pesquisas, implementações, levantamentos, etc.), no final, percebemos que a parte mais difícil é escrever os agradecimentos.

Devo agradecer a Deus pelas oportunidades que me foram dadas. Sem Ele, nada do que fiz até hoje teria sentido, e todas as pessoas que agradecerei são instrumentos dEle em minha vida.

Vou repetir um grande clichê, coisa que vários amigos já fizeram: citar nomes é injustiça, e agradecer a todos os envolvidos é impossível, mas vamos tentar. Primeiramente, gostaria de agradecer à minha família: Ivayr, Rosângela, Netto, e Letícia. Sem a ajuda de vocês, nada disso teria sido possível. Obrigado a vocês pelas palavras de amor, de apoio, de carinho, pelos puxões de orelha e pela paciência (que eu sei que foi muito grande).

Agradeço também aos amigos, que, mesmo que não tenham influência direta em alguns processos, moldam nosso caráter e nos fazem quem somos, (aí a injustiça de não citar todos!) Asafe Costa e André Monster Costa, dois irmãos que tenho desde pequeno; Cristiano, que além de orientador, se mostrou um grande amigo; Sarah e Álvaro, brothers pra todas as horas; Moacir, Lessandro, Bruno, Fabrício, Marzon, colegas com os quais aprendi muito e, além das relações de trabalho, se tornaram amigos; Pedro, Ramon, Daniel, Diego e Thiago, amigos sem comentários; e, por fim, Luís e Matheus, duas das criaturas mais malucas que já encontrei, sem as maluquices e gritaria de vocês, eu teria conseguido dormir, e não teria terminado a monografia a tempo.

Como podia o mundo voltar a ser o que era depois de tudo isso? Mas, no fim, é só uma coisa passageira, até tudo passar. Um novo dia virá. E, quando o sol brilhar, brilhará ainda mais forte. Eram essas as histórias que ficavam nas lembranças, que significavam algo. Mesmo que você fosse pequeno demais para entender o por quê. Mas acho, que entendo, sim. Agora eu sei. As pessoas dessas histórias tinham várias oportunidades de voltar atrás mas não voltavam. Elas seguiam em frente, porque tinham no que se agarrar.

(J. R. R. Tolkien)

RESUMO

Existem diversos recursos, na literatura, que guiam o processo de construção de *software* visando a criação de produtos com maior qualidade. Dentre esses recursos, está o TDD - *Test-Driven Development*. Este trabalho se propõe a avaliar o impacto do uso desta abordagem na qualidade de código-fonte, através da comparação de métricas de *software* propostas pela norma padrão de qualidade do produto de *software* aplicadas sobre dois sistemas, desenvolvidos sobre um mesmo documento de requisitos, tendo sido um desses sistemas, construído sob o uso da abordagem citada. Os resultados dessa comparação são apresentados e discutidos, trazendo uma visão prática do impacto que esse tipo de recurso pode trazer sobre o produto de *software* em geral, principalmente sobre os aspectos ligados ao código-fonte.

Palavras-Chave: Teste de *Software*; TDD; Qualidade de Código-Fonte

ABSTRACT

There are several features, in the literature, that guide the process of building software focused at creating high quality software products. Among these features is TDD - Test-Driven Development. This study evaluates the impact of the technique at the source code by comparing software metrics proposed by the software quality standards applied in two systems, developed according to the same requirements. One of these systems was built applying the mentioned technique. The results of this comparison are presented and discussed, bringing a practical view of the impact that the technique can bring to the software product, especially on aspects related to the source code.

Keywords: Teste de *Software*; TDD; Qualidade de Código-Fonte

SUMÁRIO

LISTA DE FIGURAS

1 INTRODUÇÃO

A busca pela qualidade de *software*, nos últimos anos, tem se mostrado cada vez mais presente no desenvolvimento de sistemas (PRESSMAN, 2010). Tal busca impõe a necessidade do uso de técnicas e ferramentas que auxiliem no desenvolvimento, buscando produtos de alta qualidade.

Embora tais técnicas sejam oferecidas pela Engenharia de *Software*, defeitos podem ser inseridos, o que traz a necessidade de uma etapa no desenvolvimento do *software* que tenha como objetivo minimizar a ocorrência destes erros e defeitos. Assim, o Teste de *Software*, uma das atividades de Verificação e Validação, consiste na investigação do *software* a fim de fornecer informações sobre sua qualidade em relação ao contexto em que ele deve operar.

Quando se discute qualidade de *software* a busca da qualidade dos processos de *software* se torna indispensável. Dentro da qualidade dos processos, é importante a busca contínua da qualidade do código-fonte. Com a alta complexidade dos *softwares* produzidos atualmente, esta busca se torna cada vez mais difícil e demanda mais tempo no desenvolvimento, o que pode trazer frustrações e quebras de cronogramas (RIAZ; MENDES; TEMPERO, 2009). Neste cenário de dificuldade de desenvolvimento e manutenção de *software*, surge a preocupação com a legibilidade do código-fonte, de modo a facilitar a manipulação do mesmo (JANZEN; SAIEDIAN, 2008).

Proposta por Beck (2002), a técnica *Test Driven Development* surge como uma abordagem promissora para a obtenção de um código-fonte mais legível e de qualidade. Esta técnica tem como objetivo guiar o desenvolvimento a fim de que seja escrito um código-fonte simples, e que funcione conforme a necessidade de uso do *software* descrita pelo cliente, além de funcionar como parte da documentação do projeto.

O estudo feito neste trabalho é focado na influência do Teste de *Software* na qualidade do código-fonte e do produto de *software* como um todo, abordando a técnica de TDD como linha mestra a ser seguida no processo de desenvolvimento.

1.1 Motivação

Atualmente, questões como Garantia da Qualidade de *Software* já não representam um fator diferencial para as empresas, mas um fator competitivo. Produzir *software* de qualidade é um pré-requisito para qualquer empresa de desenvolvimento.

Devido à grande necessidade de se produzir *software* de qualidade, tendo foco em indicadores de qualidade, como funcionalidade, manutenibilidade, usabilidade, etc., propostos por normas de qualidade de produto de *software*, se faz necessário o uso de alguma abordagem que auxilie na garantia da qualidade.

O TDD tem como principal objetivo a criação de código-fonte limpo (ASTELS, 2003), tornando a manutenção uma tarefa menos árdua. Outra preocupação é o desenvolvimento do *software* com a funcionalidade desejada pelo cliente (BECK, 2002; ASTELS, 2003), tendo em vista que isso se torna um grande desafio, já que muitas vezes os conceitos se perdem no meio dos processos (FOWLER, 1999).

Como o TDD antecipa falhas, problemas no resultado final do *software* são evitados, devido à realização antecipada dos testes, permitindo reduzir custos e tempos. Estas reduções levam a um aumento na qualidade do código-fonte, que será refletida na qualidade do *software* e outros artefatos.

1.2 Objetivo

O objetivo geral do presente trabalho é a análise de métricas de qualidade obtidas na comparação entre o código-fonte produzido através do uso da aborda-

gem TDD e o código produzido utilizando-se metodologias tradicionais de desenvolvimento e, assim, avaliar o impacto da técnica TDD na qualidade do código-fonte no desenvolvimento do sistema orientado a objetos.

Como objetivos específicos, pode-se citar:

- Produzir código-fonte utilizando o TDD;
- Computação de métricas escolhidas com base na literatura (segundo o trabalho de Janzen e Saiedian (2008) e métricas propostas pela ferramenta *Code Pro Analytix*) sobre o código-fonte gerado;
- Avaliação e comparação dos resultados obtidos na computação das métricas feita nos dois *softwares* desenvolvidos;
- Discussão e conclusão sobre as vantagens ou desvantagens do uso da abordagem proposta e seus impactos na qualidade do código-fonte gerado.

1.3 Síntese da Metodologia

A metodologia de desenvolvimento proposta nesse trabalho envolve a codificação de um sistema orientado a objetos a partir de duas equipes distintas: a primeira equipe utilizou a abordagem *Test-Driven Development* e a segunda utilizou uma metodologia clássica de desenvolvimento. Neste cenário, as duas equipes desenvolveram os *softwares* separadamente, pois qualquer contato entre os desenvolvedores poderia influenciar nos resultados e torná-los tendenciosos.

A descrição, bem como os requisitos do *software* construído, foram retirados de um dos trabalhos práticos da disciplina Programação Orientada a Objetos, ministrada pelo Departamento de Ciência da Computação da Universidade Federal de Lavras.

Após a finalização da etapa de desenvolvimento, foi feita a análise do código-fonte produzido utilizando a ferramenta *Code Pro Analytix* (GOOGLE,

2012), que possibilitou o levantamento dos resultados da aplicação das métricas para a comparação.

1.4 Tipo de Pesquisa

A Figura 1.1 ilustra uma estrutura de classificação para uma pesquisa científica (JUNG, 2009)

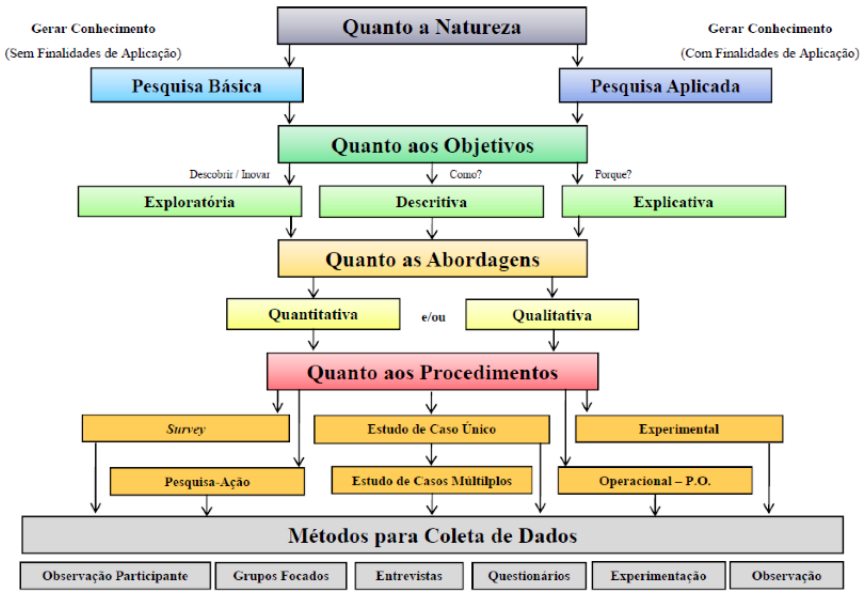

Figura 1.1: Tipos de pesquisa científica

Quanto ao tipo de pesquisa, este trabalho pode ser visto como uma pesquisa aplicada. Quanto aos seus objetivos, pode ser caracterizado como uma pesquisa exploratória, pois avalia a melhoria da qualidade no processo de desenvolvimento de um *software* através da técnica de TDD, com o intuito de descobrir possíveis impactos, sejam negativos ou positivos. Quanto à sua abordagem, este trabalho é uma pesquisa quantitativa, pois visa à medição e à análise de algumas métricas de código-fonte de dois *softwares* com mesma funcionalidade, porém desenvolvidos de maneiras diferentes. Quanto aos procedimentos, pode ser caracterizado como um estudo de caso, pois é apresentado um estudo da aplicação de

uma técnica e da análise de seus impactos no desenvolvimento de um *software*. A coleta de dados se dá pela observação do código-fonte de dois *softwares* em análise.

1.5 Estrutura do Trabalho

O restante do texto foi estruturado da seguinte forma: Capítulo 2 trata do referencial teórico necessário para o entendimento do presente trabalho, abordando conceitos como Qualidade de *Software*, Manutenção de *Software* e a abordagem "*Test-Driven Development*", além de alguns trabalhos correlatos.

O Capítulo 3 trata da metodologia utilizada para a confecção do *software* utilizado para análise, bem como as ferramentas utilizadas, as métricas selecionadas para comparação de resultados e o *software* base para comparação.

No Capítulo 4, são apresentados os resultados das métricas selecionadas, bem como uma discussão sobre estes resultados e considerações a respeito do código desenvolvido.

Por fim, o Capítulo 5 contém a conclusão do presente trabalho, juntamente com os trabalhos futuros propostos.

2 REFERENCIAL TEÓRICO

Este capítulo aborda conceitos que são fundamentais para o entendimento do presente trabalho. Aqui, são detalhados os conceitos de Qualidade de *Software*, sua definição segundo autores como Crosby (1979) e Humphrey (1989), e os conceitos de métricas de *software*, métricas de qualidade de *software* e métricas de código, segundo ISO/IEC (2001); o conceito de Manutenção de *Software*, sua definição segundo autores como Sommerville (2007), Pressman (2010), Hanna (1993) e Sneed e Brössler (2003); e, por fim, os conceitos fundamentais sobre a abordagem TDD, bem como um curto exemplo sobre o uso da mesma, utilizando como exemplo uma pequena funcionalidade do *software* construído.

2.1 Qualidade de *Software*

Com o passar dos anos, o termo "qualidade" vem sendo definido de formas diferentes por vários autores (ABRAN *et al.*, 2004). Segundo Crosby (1979), em *"Quality is free"*, o conceito de qualidade pode ser definido como conformidade dos requisitos do usuário, já Humphery (1989), em *"Managing the Software Process"*, refere-se a tal conceito como alcance de níveis excelentes de adequamento ao uso.

A qualidade de *software* melhorou significativamente nos últimos quinze anos. Uma razão para tal, é a adoção de novas técnicas e tecnologias pelas empresas, como o uso do desenvolvimento orientado a objetos e o suporte oferecido por ferramentas CASE. Além disso, nesse período, tem se dado também maior importância ao gerenciamento da qualidade de *software* e à adoção de técnicas para tal (SOMMERVILLE, 2007).

2.1.1 Norma ISO/IEC 9126

A norma ISO/IEC 9126 é um padrão definido pela IEEE, *International Organization of Standardization/International Electrotechnical Commission*, que define características relacionadas à qualidade de *software* (ZEISS *et al.*, 2007).

Desenvolver *software* com qualidade é um desafio, pois se trata de questões abstratas, sendo uma tarefa difícil encontrar parâmetros para medir pontos de qualidade (PRESSMAN, 2010). Para isso, a norma ISO/IEC 9126 descreve um modelo de qualidade composto em duas partes: qualidade interna e externa, e qualidade de uso. A primeira especifica seis características para qualidade interna e externa, que são subdivididas em subcaracterísticas, manifestadas externamente, quando o *software* é utilizado como parte de um sistema. Já a segunda, especifica quatro características de qualidade em uso, que é, para o usuário, o efeito combinado das seis características de qualidade do produto de *software*.

Segundo a norma ISO/IEC 9126, o modelo de qualidade para qualidade externa e interna categoriza os atributos de qualidade em seis características - funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade – que são representadas na Figura 2.1 e enumeradas a seguir.

1. **Confiabilidade** consiste na capacidade do produto de *software* de manter um nível de desempenho especificado, quando usado em condições especificadas, abordando subcaracterísicas como: maturidade, tolerância a falhas, recuperabilidade e conformidade. Como em *software* não ocorre desgaste ou envelhecimento, limitações em confiabilidade são decorrentes de defeitos na especificação de requisitos, projeto e implementação. Tais falhas decorrentes desses defeitos dependem de como o produto de *software* é usado e das opções de programa selecionadas e não do tempo decorrido.

2. **Eficiência** é a capacidade do produto de *software* de apresentar desempenho apropriado, relativo à quantidade de recursos usados, sob condições especi-

ficadas. Tais recursos podem incluir outros produtos de *software*, configurações de *hardware* e *software* do sistema e materiais como, papel, disquetes, etc.. Esta característica engloba, também, algumas subcaracterísticas: comportamento em relação ao tempo, utilização de recursos e conformidade.

3. **Funcionalidade** pode ser definida como a capacidade do produto de *software* de prover funções que atendam às necessidades explícitas e implícitas, quando o *software* estiver sendo utilizado sob condições específicas. Esta característica está relacionada com o que o *software* faz para atender a estas necessidades, e aborda subcaracterísticas como: adequação, acurácia, interoperabilidade e segurança de acesso e conformidade.

4. **Manutenibilidade** diz respeito à capacidade do produto de *software* de ser modificado. Tais modificações podem incluir correções, melhorias, adaptações do *software* devido a mudanças no ambiente e nos seus requisitos ou especificações funcionais. Envolve subcaracterísticas como: analisabilidade, modificabilidade, estabilidade, testabilidade e conformidade.

5. **Portabilidade** diz respeito à capacidade do produto de *software* de ser transferido de um ambiente para outro, abordando as subcaracterísticas: adaptabilidade, capacidade para ser instalado, coexistência, capacidade para substituir outros *softwares* especificados e conformidade.

6. **Usabilidade** diz respeito à capacidade do produto de *software* de ser compreendido, aprendido, operado e atraente ao usuário. Alguns aspectos como funcionalidade, confiabilidade e eficiência também afetarão a usabilidade. Tal conceito pode ser também dividido em algumas subcaracterísticas, tais como: inteligibilidade, apreensibilidade, operacionalidade, atratividade e conformidade.

Todas as características possuem a subcaracterística “conformidade”, que consiste em atributos do *software* que evidenciam o quanto o mesmo obedece aos

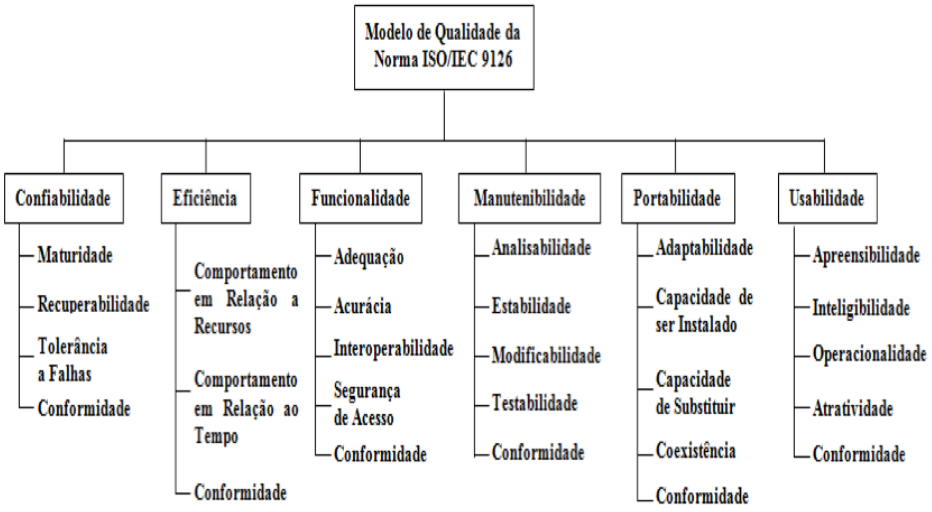

Figura 2.1: Características de Qualidade da Norma ISO/IEC 9126

requisitos de legislação e todo o tipo de padronização ou normalização aplicável ao contexto.

A segunda parte proposta pela norma é a qualidade em uso, que se trata da visão da qualidade sob a perspectiva do usuário. A qualidade de uso é dividida em quatro atributos de qualidade, dispostos na Figura 2.2.

Figura 2.2: Atributos de Qualidade em Uso da Norma ISO/IEC 9126

Os atributos referentes a qualidade em uso não serão detalhados aqui, pois este trabalho está focado na avaliação e análise dos aspectos de qualidade interna

e externa, mas estes podem ser encontrados com mais detalhes na norma ISO/IEC 9126.

2.1.2 Medição e Métricas de *Software*

Segundo Sommerville (2007), avaliações de qualidade são caras e demoradas e, inevitavelmente, atrasam a conclusão de um sistema de *software*. Idealmente, seria possível acelerar o processo de revisão usando ferramentas para processar o *design* de *software* ou programa e fazer algumas avaliações automatizadas da qualidade do *software*. Estas avaliações podem verificar se o *software* atingiu a qualidade exigida e, caso contrário, destacar áreas do *software* onde o reexame deve incidir.

Assim, para realizar estas avaliações e medir seus resultados, é necessário definir alguns conceitos, como medição, métricas e indicadores. Como tais conceitos são usados na engenharia de *software*, geralmente, em um mesmo contexto, é importante entender as pequenas diferenças entre cada um deles. Pressman (2010) faz uma breve descrição de cada um desses termos:

Métrica: Métrica relaciona medidas individuais de alguma maneira (por exemplo, o número médio de erros encontrado por revisão, ou o número médio de erros encontrado por teste unitário). A *IEEE Standard Glossary of Software Engineering Terminology* define métrica como uma medida quantitativa do grau que cada sistema, componente ou processo possui de um dado atributo.

Medição: Medição é o ato de determinar uma medida. A medição ocorre como o resultado da coleção de uma ou mais métricas (por exemplo, um número de revisões de componentes e testes unitários é investigado para coletar medidas de quantidades de erros para cada componente). Medição de *software* se preocupa em atribuir valores numéricos a alguns atributos do produto ou processo de *software*. Comparando esses valores uns com os outros e com

padrões que são aplicados na organização, é possível chegar a conclusões quanto à qualidade do produto e/ou processo de *software*.

Medida: Como na língua inglesa a palavra "*measure*" pode ser usada tanto como substantivo, quanto como verbo (*to measure*), sua definição pode se tornar um pouco confusa. No contexto da engenharia de *software*, uma medida provê um indicador quantitativo de extensão, quantidade, dimensão, capacidade ou tamanho de algum atributo do produto/processo de *software*. Uma medida é estabelecida quando se define um *data point*, ou seja, um componente ou processo a ser analisado (por exemplo, o numero de erros descobertos dentro de um único componente do *software*).

Indicadores: Um indicador é uma métrica ou combinação de métricas que provê *insights* (informações privilegiadas ou diagnósticos) no processo de *software*, projeto de *software*, ou no próprio produto. Um indicador provê informações que permitem ao gerente de projetos ou engenheiro de *software* ajustar os processos, o projeto, ou o produto para um melhor trabalho.

2.1.3 Métricas de Qualidade de *Software*

Ainda, como parte fundamental deste estudo, é importante detalhar o conceito de "métricas de *software*". Estas, auxiliam de forma efetiva a administração de um projeto de *software* com o intuito de controlar determinadas tarefas ou rotinas (PRESSMAN, 2010), garantindo o acompanhamento durante as etapas que compõem o ciclo de vida de um *software* (KOSCIANSKI; SOARES, 2007; LANZA; MARINESCU; DUCASSE, 2007).

Portanto, do ponto de vista de medição, Duarte e Falbo (2000) sugerem dois tipos de métricas (onde as métricas diretas podem ser vistas como custo e esforço; porém, a qualidade e a funcionalidade do *software*, ou a sua capacidade de manutenção, são mais difíceis de se avaliar e só podem ser medidas de forma indireta):

1. **Métricas Diretas:** métricas realizadas em função de atributos observados que, através de uma medição, se aplica uma medida. Exemplos de métricas diretas podem ser custo, esforço, número de linhas de código, número de erros, etc..

2. **Métricas Indiretas:** métricas que são descritas em termos de outras características, obtidas através de outras métricas (qualidade, complexidade, eficiência, etc.).

Assim, é possível enquadrar as métricas definidas pela norma ISO/IEC 9126 às propostas por Duarte e Falbo (2000), sendo, as da norma, classificadas em três categorias, citadas nas seções a seguir.

2.1.3.1 Métricas Externas

Métricas externas utilizam medidas de um produto de *software* derivadas de medidas do comportamento do sistema do qual o *software* faz parte, através do teste, operação e observação do *software* executável. Estas, oferecem aos usuários e avaliadores a possibilidade de se avaliar o produto de *software* durante seu teste ou operação (ISO/IEC, 2001).

2.1.3.2 Métricas Internas

Estas, são aplicadas à parte não executável do produto de *software*, ou seja, documentos, código-fonte, etc., durante o processo de codificação. Assim, elas têm como objetivo assegurar que a qualidade externa e a qualidade em uso sejam alcançadas. Além disso, elas oferecem aos usuários e avaliadores a possibilidade de avaliar o produto de *software* e considerar questões relativas à qualidade bem antes do *software* chegar a um estado operável (ISO/IEC, 2001).

2.1.4 Métricas de Código

Como um dos objetivos deste trabaho é explorar métricas pertinentes à fase de codificação do *software*, é de suma importância conceituar métricas de código, possuindo um importante papel no acompanhamento e controle dos processos de codificação e teste (SOMMERVILLE, 2007). Estas métricas são encarregadas de manter o *software* em funcionamento (BECK, 2000). Desta maneira, o código se torna um objeto que será inspecionado e verificado a todo momento, assim, garante-se o bom funcionamento do *software*, desde a menor unidade de código, até a integração de todo o sistema (DUVALL; MATYAS; GLOVER, 2007).

Segundo Poppendieck e Poppendieck (2003), métricas de código possuem como grande benefício a eliminação do retrabalho. Este problema é um dos principais problemas relatados pela literatura na engenharia de *software* (ABRAN *et al.*, 2004).

Exemplos de métricas de código são:

1. **Tamanho:** medida determinística para outras métricas. Entre as principais métricas deste grupo, pode-se encontrar: **LOC**, *Lines Of Code*, **KLOC**, *Kilo Lines Of Code* e **SLOC**, *Source Lines Of Code* (PRESSMAN, 2010).

2. **Complexidade:** mensura o quanto um determinado bloco de código é legível, ou o quão complexo pode se tornar um elevado número de aninhamentos de estruturas lógicas (KOSCIANSKI; SOARES, 2007). Neste grupo, pode-se encontrar métricas como **Complexidade Ciclomática**, **Profundidade Média de Bloco** e **Métodos Ponderados**.

3. **Cobertura de Código:** ou *Code Coverage* (GOOGLE, 2012), é associada ao desenvolvimento orientado a testes. Essa métrica mensura a abrangência dos testes em relação ao código desenvolvido, tornando o desenvolvimento coeso com o que é solicitado (BECK, 2000, 2002).

4. **Métricas Orientadas a Objetos:** visam avaliar fatores como acoplamento ou abstração de um determinado bloco de código e medir a profundidade de árvore de herança (LANZA; MARINESCU; DUCASSE, 2007; PRESSMAN, 2010; SOMMERVILLE, 2007). Métricas neste contexto são **WMC**, *Weighted Methos per Class*, **NOC**, *Number Of Children*, entre outras.

2.2 Manutenção

Manutenção de *software* é o processo geral de mudar um sistema depois que ele foi entregue ao cliente. As mudanças feitas no *software* podem ser simples mudanças para corrigir erros de código, mudanças mais extensas para corrigir erros de projeto ou melhorias significativas para corrigir erros de especificação ou adequar novos requisitos (SOMMERVILLE, 2007). Esta fase do ciclo de vida do *software* é considerada a atividade que demanda o maior volume de esforço dentre todas as atividades da engenharia de *software* (PRESSMAN, 2010; HANNA, 1993; SNEED; BRÖSSLER, 2003). A manutenção de *software* pode ser dividida em quatro tipos, segundo Pressman (2010):

1. **Adaptativa:** é realizada quando o produto de *software* precisa ser adaptado às novas tecnologias implantadas no ambiente operacional.

2. **Corretiva:** é realizada quando são corrigidos erros não identificados durante o Fluxo de Trabalho de Teste.

3. **Evolutiva:** é realizada quando o produto de *software* deve englobar novos requisitos ou melhorias decorrentes da evolução na tecnologia de implementação utilizada.

4. **Preventiva:** é realizada quando o produto de *software* é alterado para aumentar sua manutenibilidade ou confiabilidade. Este tipo de manutenção é

relativamente raro em ambientes de desenvolvimento, já que não alteram o comportamento fundamental do *software*.

Sommerville (2007) também propõe três tipos de manutenção de *software*, que complementam a classificação de Pressman (2010):

1. **Manutenção para reparar falhas de *software*:** erros de código são, geralmente, relativamente baratos de se corrigir; erros de design são mais caros, já que podem envolver a reescrita de grandes partes do código. Erros de requisitos de *software* são os mais caros de se corrigir, já que envolvem uma grande reestruturação no design do mesmo.

2. **Manutenção para adaptar o *software* a diferentes ambientes operacionais:**este tipo de manutenção é necessário quando alguns aspectos do ambiente do sistema, como *hardware*, sistema operacional, etc., mudam.

3. **Manutenção para adicionar ou modificar funcionalidades do sistema:** este tipo de manutenção é necessário quando os requisitos de sistema mudam em resposta às mudanças organizacionais ou mudanças de negócio.

Um processo de manutenção de *software* diz respeito a um conjunto de etapas bem definidas, que direcionam a atividade de manutenção de *software*, com o objetivo de satisfazer as necessidades do usuário de maneira planejada e controlada (PIGOSKI, 1997).

Segundo a norma NBR ISO/IEC 12207 (Associação Brasileira de Normas Técnicas, 1998), o objetivo do processo de manutenção é modificar um produto de *software* existente, preservando sua integridade. Além disso, o processo de manutenção começa a partir do momento em que é necessário efetuar modificações no código e na manutenção do sistema devido a um problema, adaptação a novos requisitos e/ou novos equipamentos ou necessidade de melhoria, e termina no momento em que o software não será mais utilizado.

2.2.1 Manutenibilidade

Grande parte dos sistemas dos quais as organizações dependem atualmente foram desenvolvidos há anos, e, desde então, estes sistemas são migrados para novas plataformas, ajustados devido a mudanças nos equipamentos de *hardware* e nos sistemas operacionais e melhorados para atender a novos requisitos do cliente. O resultado são aplicações mal estruturadas, mal codificadas e com documentação pobre (PRESSMAN, 2010).

Portanto, os *softwares* devem ser projetados tendo em vista a manutenibilidade e a sua documentação deve permitir a correção e o reuso de código-fonte durante seu aprimoramento. Manutenibilidade não é um processo do desenvolvimento do *software*, mas uma característica de qualidade utilizada para avaliar um *software*, sendo um fator que impacta significantemente os custos do mesmo (RIAZ; MENDES; TEMPERO, 2009).

Segundo Abran *et al.* (2004), a IEEE define manutenibilidade como a facilidade com que o *software* pode ser mantido, aumentado, adaptado ou corrigido para satisfazer os requisitos especificados, e, segundo a Norma ISO/IEC 9126, pode ser dividida em quatro subcategorias de qualidade:

1. **Analisabilidade:** capacidade de permitir o diagnóstico de deficiências ou causas de falhas no *software*, ou a identificação de partes a serem modificadas.

2. **Modificabilidade:** capacidade de permitir que uma modificação seja implementada.

3. **Estabilidade:** capacidade de evitar efeitos inesperados decorrentes de modificações no *software*.

4. **Testabilidade:** capacidade de permitir que o *software*, quando modificado, seja validado.

Ainda, a ISO/IEC 9126-2:2002, ISO/IEC 9126-3:2002 e ISO/IEC 9126-4:2002 propõe, consecutivamente, métricas para qualidade externa e qualidade interna de *software*, e métricas para qualidade em uso. Nos tópicos a seguir, são detalhadas as métricas externas e internas para manutenibilidade de *software*.

2.2.2 Métricas Externas para Manutenibilidade

Uma métrica externa de manutenibilidade deve ser capaz de medir atributos tais como o comportamento do mantenedor, utilizador ou sistema, incluindo o *software*, quando o *software* é modificado durante os testes ou manutenção.

1. **Métricas para analisabilidade:** deve ser capaz de medir atributos como o esforço do mantenedor ou utilizador, ou gasto de recursos na tentativa de diagnosticar deficiências ou causas de falhas, ou para a identificação de partes a serem modificadas.

2. **Métricas para modificabilidade:** deve ser capaz de medir atributos como o esforço do mantenedor ou usuário, medindo o comportamento do mantenedor, usuário ou sistema, quando se implementa uma modificação.

3. **Métricas para estabilidade:** deve ser capaz de medir atributos relacionados a um comportamento inesperado do sistema quando o *software* é testado ou operado após a modificação.

4. **Métricas para testabilidade:** deve ser capaz de medir atributos como esforço do mantenedor ou utilizador através da medição do comportamento do mantenedor, utilizador ou sistema, quando testar o *software* modificado ou não modificado.

2.2.3 Métricas Internas para Manutenibilidade

Métricas internas de manutenibilidade são usadas para prever o nível de esforço necessário para modificar o produto de *software*.

1. **Métricas para analisabilidade:** indicam um conjunto de atributos para prever o esforço do mantenedor ou usuário, ou gasto de recursos na tentativa de diagnosticar deficiências ou causas de fracasso, ou para identificação de partes a serem modificados no produto de *software*.

2. **Métricas para modificabilidade:** indicam um conjunto de atributos para prever o esforço do mantenedor ou usuário esforço despendido ao tentar implementar uma modificação no produto de *software*.

3. **Métricas para estabilidade:** indicam um conjunto de atributos para prever o quão estável o produto de *software* seria depois de qualquer modificação.

4. **Métricas para testabilidade:** indicam um conjunto de atributos para a previsão da quantidade de funções autônomas de teste projetadas e implementadas presentes no produto de *software*.

2.3 *Test-Driven Development*

TDD (*Test-Driven Development* - desenvolvimento orientado por testes) (BECK, 2002) é uma das práticas centrais da metodologia ágil de desenvolvimento *Extreme Programming* (WASMUS; GROSS, 2007) (para saber mais sobre *Extreme Programming*, é indicada a leitura de (BECK, 2000)). O TDD afirma que o ciclo clássico de desenvolvimento, onde primeiro se escreve o código e depois se faz os testes para ter certeza de que tudo funciona como proposto, pode não ser a melhor abordagem. A técnica reverte a ordem de implementação e teste quando comparado à abordagem tradicional "*test-last*" (BECK, 2002): onde, em uma abordagem "*test-last*", a funcionalidade é implementada e depois testada. A abordagem "*test-first*" faz com que o programador primeiro escreva os testes para a funcionalidade, depois implemente o código à medida que esta implementação se faz necessária aos testes.

Sua essência de desenvolvimento é descrita como uma cadeia de tarefas iterativas e conectadas entre si (MüLLER; HöFER, 2007), que consistem em: escrever os testes para um pequeno incremento de funcionalidade, escrever o código para que tal teste seja executado corretamente, e, após esse incremento, o código é refatorado para manter a qualidade (BECK, 2002). Enquanto o código é desenvolvido, o desenvolvedor deve focar apenas no desenvolvimento da funcionalidade que está sendo feita, evitando qualquer código adicional que não será utilizado (TORCHIANO; SILLITTI, 2009). O ciclo composto por estas três tarefas é ilustrado na Figura 2.3.

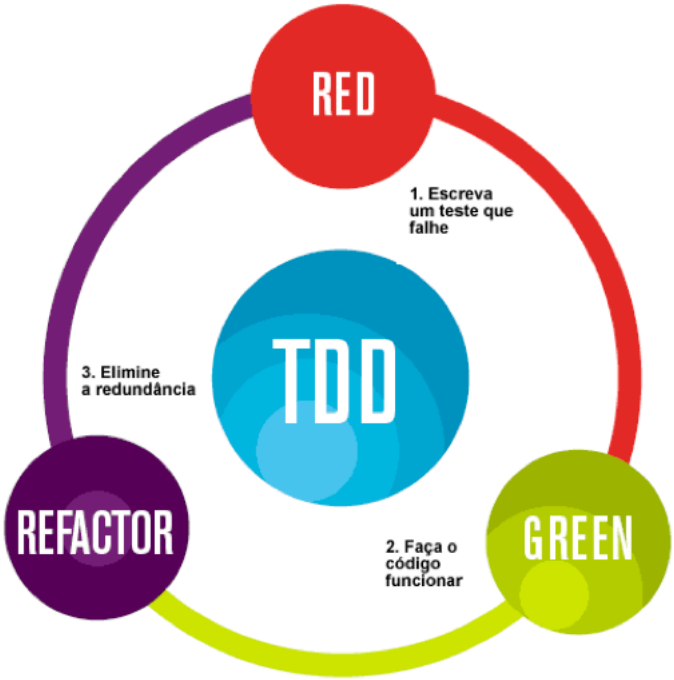

Figura 2.3: Ciclo de TDD (Adaptado de Brief, 2001)

Juntamente com a programação em par e *refactoring*, técnicas de grande importância na metodologia ágil *Extreme Programming*, propostas por Kent Beck em (BECK, 2000), o TDD tem recebido atenção individual desde a introdução do *XP*. Desenvolvedores têm criado ferramentas específicas para o suporte de TDD

para várias linguagens de programação e escrito vários livros explicando como aplicar os conceitos do TDD (JANZEN; SAIEDIAN, 2005).

Apesar de seu crescimento, esta técnica ainda não tem seu uso frequente. As empresas têm adotado a técnica apenas recentemente (WASMUS; GROSS, 2007), já que o *Extreme Programming* ainda não se solidificou no mercado, sendo, as metodologias tradicionais, as mais utilizadas (JANZEN; SAIEDIAN, 2005).

2.3.1 Funcionamento

Como explicado anteriormente, o funcionamento do TDD pode ser resumido em três passos (BECK, 2002): a tarefa de criar o teste, criar o código-fonte e refatorá-lo. Assim, segundo Astels (2003), os passos do TDD devem ser vistos como uma iteração.

As fases do TDD, assim como seu funcionamento, serão detalhados a seguir. O exemplo utilizado é uma pequena funcionalidade do *software* desenvolvido no presente estudo, a ser detalhado no Capítulo 3.

2.3.1.1 Roteiro

A funcionalidade que se deseja implementar, segundo a especificação do sistema proposto para o estudo, se baseia na habilitação de um celular pela operadora: “Habilitar (criar um novo) um celular. O usuário escolhe o tipo de celular (*Smartphone* ou *RegularPhone*), o tipo de plano (cartão ou assinatura), promoções (entre as disponíveis para o modelo de telefone e plano) e fornece o dia de vencimento da fatura (para celular habilitado em plano de assinatura). O número do celular é gerado automaticamente pelo sistema”.

O primeiro passo para o desenvolvimento da funcionalidade proposta é traçar um roteiro de testes (ANICHE, 2012). Para criar este roteiro, é necessário interpretar cada uma das possíveis realidades para a funcionalidade. Assim, pode-se chegar à seguinte situação:

1. **Habilitar um celular qualquer:** consiste em habilitar um celular qualquer, sem nenhuma especificação.

2. **Habilitar um *smartphone* qualquer:** consiste em habilitar um celular do tipo *smartphone*, sem outras restrições.

3. **Habilitar um *regularphone* qualquer:** consiste em habilitar um celular do tipo *regularphone*, sem outras restrições.

4. **Habilitar um *smartphone* do plano cartão:** consiste em habilitar um celular do tipo *smartphone* no plano cartão (pré-pago).

5. **Habilitar um *regularphone* do plano cartão:** consiste em habilitar um celular do tipo *regularphone* no plano cartão (pré-pago).

6. **Habilitar um *smartphone* do plano assinatura:** consiste em habilitar um celular do tipo *smartphone* no plano assinatura (pós-pago).

7. **Habilitar um *regularphone* do plano assinatura:** consiste em habilitar um celular do tipo *regularphone* no plano assinatura (pós-pago).

2.3.1.2 *Test*

Traçado o roteiro dos testes que a funcionalidade deve satisfazer, inicia-se a fase do teste. Não é necessário criar um teste para cada um dos itens citados no roteiro, mas é fundamental que todos eles sejam satisfeitos.

A fase de teste é a primeira parte do ciclo *Test-Green-Refactor* (BECK, 2002). Nela, é escrito o primeiro teste, que deve falhar, ilustrado na Figura 2.4

```
1. @Test
2. public void deveHabilitarUmCelularQualquer(){
3.     Celular celular = new Celular();
4.     assertTrue(celular.isHabilitado());
5. }
```

Figura 2.4: Primeiro teste: “deveHabilitarUmCelularQualquer()”

Explicando detalhadamente o código, na primeira linha, "@Test" é uma anotação que é utilizada para que o compilador Java reconheça que o método precedido pela anotação é um teste unitário. A segunda linha é a declaração do método, contendo sua visibilidade, retorno, nome e parâmetros e a terceira linha, contém a instanciação do objeto foco do teste. Para o teste unitário, a quarta linha é a mais importante, que é onde assegura-se de que o valor de retorno do método chamado é um determinado valor (valor esperado).

Obviamente, e como desejado, o teste falha, já que não existe o método "isHabilitado()" na classe "Celular". Desta maneira, a próxima etapa é implementar a funcionalidade de maneira que o teste seja bem sucedido, independente de como a funcionalidade seja implementada. Esta implementação será mostrada na próxima sub-seção.

É importante perceber que apenas o primeiro item do roteiro foi satisfeito pelo teste. Ainda é necessário cobrir todos os itens restantes.

2.3.1.3 *Green*

Uma vez que o teste foi escrito e falhou, é hora de implementar o método que o faz funcionar (BECK, 2002). No exemplo, o desejado é que a linha "assertTrue(celular.isHabilitado())" seja verdadeira. Desta maneira, o método "isHabilitado()" deve retornar um booleano verdadeiro.

O passo "Green" do ciclo do TDD diz que, independente de como seja feita a implementação do método que o teste valida, o teste deve funcionar (ANICHE, 2012). Então, a maneira mais simples e rápida, é implementar o método "isHabilitado()" para retornar o booleano verdadeiro, como ilustrado na Figura 2.5

```
1. public boolean isHabilitado(){
2.         return true;
3. }
```

Figura 2.5: Implementação do método "isHabilitado()"

Obviamente, esta não é a melhor opção para a implementação do método, e não retrata a realidade da funcionalidade. Mas o importante, neste passo, é fazer o teste funcionar, e isto foi feito.

2.3.1.4 *Refactor*

Feito o roteiro de testes, feito o teste unitário, e implementada a funcionalidade, é hora de reescrever o código da funcionalidade excluindo qualquer tipo de duplicação, redundância, ou código que force algum tipo de resultado (BECK, 2002) (por exemplo, o código ilustrado na Figura 2.5).

Uma vez que o teste já foi escrito, é necessário que ele funcione para todas as possíveis realidades enumeradas no roteiro de testes. Então, começa-se a implementar uma solução que satisfaça cada um dos cenários.

O passo de *refactor* pode ser ilustrado em Figura 2.6 e Figura 2.7, com evoluções incrementais no código-fonte.

```
1. public boolean isHabilitado(){
2.         return ((!this.tipo.equals("") &&
3.                 (this.plano != null))
4. }
```

Figura 2.6: Implementação do método “isHabilitado()” - primeiro passo do refactor

```
1. public boolean isHabilitado(){
2.         return ((((!this.tipo.equals("") &&
3.                 this.plano != null)
4.                 && this.numero != 0))
5. }
```

Figura 2.7: Implementação do método “isHabilitado()” - segundo passo do refactor

É importante ressaltar que o desenvolvimento da funcionalidade é uma tarefa que deve ser feita paralelamente aos testes, ou seja, deve ser feita implementando uma pequena parte da funcionalidade, forçando o resultado para que o

teste funcione, refatorando o código para eliminar redundâncias, e assim sucessivamente. Assim, a funcionalidade é desenvolvida incrementalmente, até que se chegue a uma solução final.

2.4 Trabalhos Relacionados

Esta seção trata de trabalhos de profissionais e pesquisadores envolvidos no desenvolvimento de *softwares* que têm investido esforços para encontrar maneiras de diminuir custos empregados na realização de atividade de manutenção de *softwares*. Três trabalhos podem ser citados neste capítulo: Canfora e Visaggio (2009), Janzen e Saiedian (2008) e Vu *et al.* (2009).

No primeiro trabalho, de Canfora e Visaggio (2009), foram utilizadas duas equipes independentes para avaliar a aplicação da abordagem TDD. Uma equipe realizou testes antes, como proposto pela abordagem, e a outra realizou os testes depois. Após a conclusão do trabalho, medições foram feitas utilizando métricas para analisar o código fonte, tais como complexidade ciclomática, número de ramificações de dependências, métodos ponderados, quantidade de linhas por classe e por método, e número de métodos por classe. Assim, foram avaliadas a influência entre as classes e o quanto o código-fonte e seu acoplamento impactavam na qualidade interna. A conclusão foi que a técnica de TDD é uma alternativa à forma tradicional de desenvolvimento e favorece a construção de código fonte manutenível.

O segundo trabalho, de Janzen e Saiedian (2008), apresenta um estudo com alunos de graduação e pós-graduação organizados em duas equipes. No trabalho, estas equipes desenvolveram um *software* utilizando o paradigma de orientação a objetos, sendo que uma equipe desenvovleu utilizando a abordagem TDD e a outra equipe desenvolveu utilizando uma metodologia "*test-last*" de desenvolvimento. Este trabalho teve como objetivo concluir se o uso do TDD melhora o *design* do código-fonte, realizando comparações dos resultados obtidos nos dois

softwares desenvolvidos. Após o desenvolvimento, medições foram feitas utilizando métricas semelhantes às selecionadas no trabalho de Canfora e Visaggio (2009). Como resultado, o *software* foi construido com classes menores e com menos responsabilidades, proporcionando simplicidade ao código-fonte e facilidade de compreensão, permitindo perceber que a abordagem de TDD guiou o desenvolvimento para a construção de código-fonte mais coeso e com reaproveitamento maior.

Por fim, o trabalho de Vu *et al.* (2009) identifica resultados diferentes relacionados à abordagem proposta. Semelhantes aos métodos aplicados nos dois trabalhos anteriormente citados, os métodos aplicados a este trabalho mostra que, em algumas equipes, a técnica foi responsável por uma queda de produtividade. O grande problema destacado por Vu *et al.* (2009) na abordagem TDD é depender da capacidade e da preferência da equipe que a utiliza.

O presente estudo busca permear os conceitos levantados pelos autores dos três trabalhos citados e corroborar os resultados encontrados nos mesmos, buscando um maior apoio no trabalho de Janzen e Saiedian (2008).

3 METODOLOGIA

Este capítulo trata da metodologia utilizada para o desenvolvimento do presente trabalho, abordando aspectos como o detalhamento das ferramentas (IDE - *Integrated Development Environment*, *frameworks*, ferramenta para metrificação), o detalhamento dos requisitos do *software* implementado e as métricas utilizadas para comparação.

3.1 *Software* base

O sistema utilizado para o desenvolvimento trata-se de um dos trabalhos práticos da disciplina Programação Orientada a Objetos, ministrada pelo Departamento de Ciência da Computação da Universidade Federal de Lavras. O *software* simula um sistema de operadora de telefonia móvel, onde é possível manter dados sobre clientes, planos, promoções e telefones.

Para o funcionamento do *software*, foram definidas algumas operações, como cadastro, inserção de créditos, realização de ligações, assim como um conjunto de restrições para cada uma dessas operações, que são regras para o funcionamento do sistema. Mais informações sobre o conjunto de regras e operações podem ser encontradas no Apêndice A.

Dentre os trabalhos desenvolvidos pelos alunos, um foi selecionado pelo professor da disciplina para a comparação. Este trabalho é referenciado como "*software* base" ao longo do texto.

O desenvolvimento do *software* base foi feito sob uma perspectiva clássica de desenvolvimento, sem o uso de *frameworks* ou metodologias que pudessem influenciar de alguma maneira o código-fonte produzido, sendo baseado apenas nos conhecimentos do desenvolvedor, exceto pela presença do padrão de projeto *Factory Method*, que foi utilizado em casos onde a utilização de herança seria uma alternativa de implementação.

3.2 *Software* desenvolvido

No desenvolvimento do *software* proposto, como citado anteriormente, não houve contato com o código-fonte do *software* base, tendo apenas o documento de requisistos guiar o desenvolvimento. Assim, foi possível conseguir um maior nível de independência entre os códigos estudados.

A Figura 3.1 traz os resultados das métricas para o sistema implementado. A análise e discussão desses valores é apresentada no Capítulo 4.

Metric	Value
⊞ Average Block Depth	0.90
⊞ Average Cyclomatic Complexity	1.23
⊞ Average Lines Of Code Per Method	4.14
⊞ Average Number of Methods Per Type	3.00
Efferent Couplings	0
⊞ Lines of Code	513
⊞ Number of Types	8
⊞ Weighted Methods	132

Figura 3.1: Resultado das métricas para o *Software* TDD

A abordagem TDD estudada permitiu ao produto de *software* construído apresentar, desde o início de seu desenvolvimento, um comportamento indicativo da melhora esperada. Este fato é explicado no Capítulo 5.

Um exemplo da aplicação da técnica pode ser visualizado nas Figuras 3.2 e 3.3. Nelas, são ilustrados, respectivamente, o teste desenvolvido para a funcionalidade, e a funcionalidade implementada ao fim das iterações do ciclo de TDD.

```
1. @Test
2. public void deveHabilitarQualquerCelularDeQualquerPlano() {
3.     assertTrue(celular.isHabilitado());
4. }
```

Figura 3.2: Exemplo de teste de TDD

```
1. public boolean isHabilitado(){
2.         return ((((!this.tipo.equals("") &&
3.                 this.plano != null)
4.                 && this.numero != 0))
5. }
```

Figura 3.3: Exemplo de funcionalidade após o fim das iterações do TDD

3.3 Ferramentas Utilizadas

Neste trabalho, tanto o *software base*, quanto o *software* desenvolvido utilizando TDD, foram escritos na a linguagem Java (ORACLE, 1995) e baseados nos princípios do paradigma de orientação a objetos.

No *software* desenvolvido com o auxílio do TDD (BECK, 2002), foi utilizada a IDE Eclipse (ECLIPSE FOUNDATION, 2012), onde foi instalado o *plugin* da ferramenta *Code Pro Analytix* (GOOGLE, 2012), utilizada para a metrificação e levantamento dos resultados. Além disso, é importante citar que foi utilizado o *framework* JUnit 4.11 (JUNIT TEAM, 2012), que possibilitou a confecção dos testes unitários necessários para o desenvolvimento do trabalho.

3.3.1 *Code Pro Analytix*

Para o desenvolvimento do presente trabalho, foi necessário o uso de uma ferramenta que possibilitasse a metrificação do código fonte e a criação de testes automatizados. A escolha dessa ferramenta se deu devido à sua fácil integração com a IDE escolhida e por sua facilidade de manuseio, fornecendo um conjunto de métricas que atendiam às necessidades do projeto.

Dentre as funcionalidades disponíveis nessa ferramenta, este trabalho destaca duas com maior importância. A primeira é a Tabela de Métricas, recurso essencial para a execução do objetivo analítico do projeto. Este recurso, de acordo com o documento disponível em Google (2012), fornece informações de grande

importância sobre o código implementado. A segunda, é o *"Code Coverage"*, ou "Cobertura de Código", que é responsável por mostrar o nível de cobertura dos testes sobre o código implementado.

3.3.2 Tabela de Métricas

A tabela de métricas, ilustrada na Figura 3.4 contém uma lista de métricas e seus correspondentes valores que foram computados com base em um arquivo. O nome da métrica é mostrado na primeira coluna e seu valor mostrado na segunda.

Para que sejam mostradas as métricas para cada escopo específico (projeto, pacote, etc.), pode-se expandir cada item do menu. Qualquer métrica que tenha excedido os valores definidos previamente, será destacada, por padrão, em vermelho.

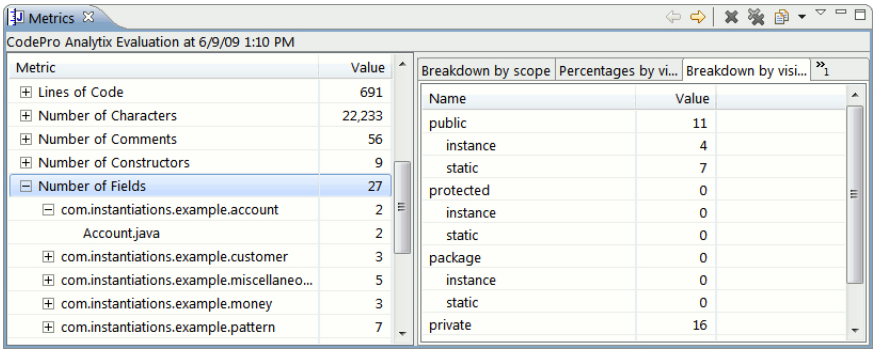

Figura 3.4: Tabela de Métricas segundo Google (2012)

Além disso, o *Code Pro Analytix* permite exportar relatórios em formato HTML (*HyperText Markup Language*) e gráficos, facilitando a visualização dos resultados, ilustrados, respectivamente, nas Figuras 3.5, 3.6 e 3.7.

3.3.3 *Code Coverage*

A ferramenta *Code Coverage* (cobertura de código) permite medir o quanto do código está sendo coberto pelos testes automatizados. Provavelmente, a razão mais comum para medir cobertura de código é avaliar a efetividade dos testes ao

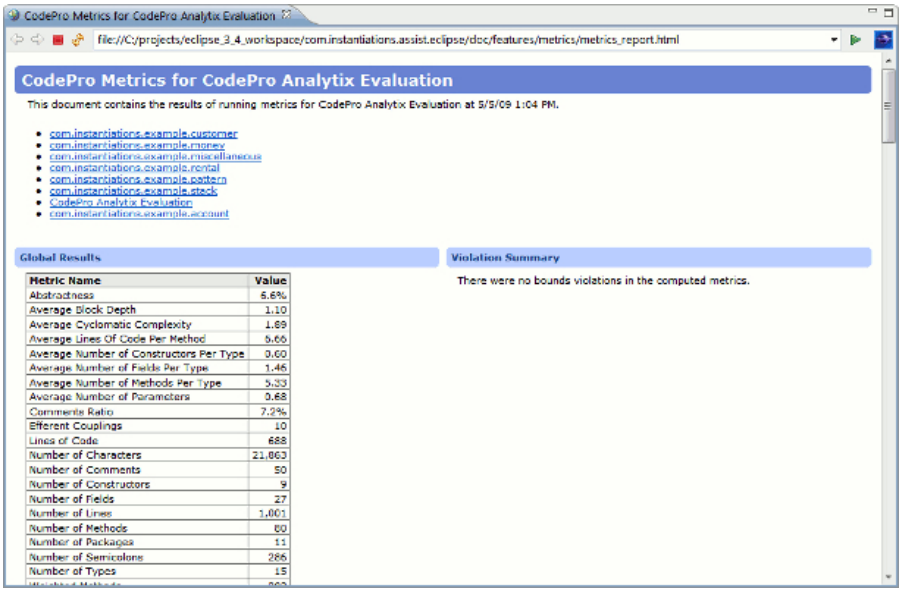

Figura 3.5: Relatório HTML segundo Google (2012)

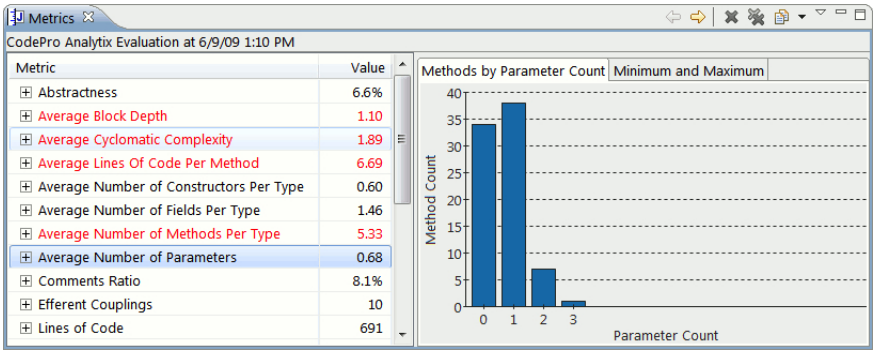

Figura 3.6: Exemplo de Gráfico Exportado por Google (2012)

longo de todos os possíveis caminhos através do código (GOOGLE, 2012). Cada vez que se computa métricas em uma determinada classe, é gerado um relatório de cobertura, ilustrado na Figura 3.8

3.4 Métricas

Como um dos objetivos do presente trabalho é a avaliação e análise das métricas de código, detalhadas no Capítulo 2, foram avaliadas as métricas relati-

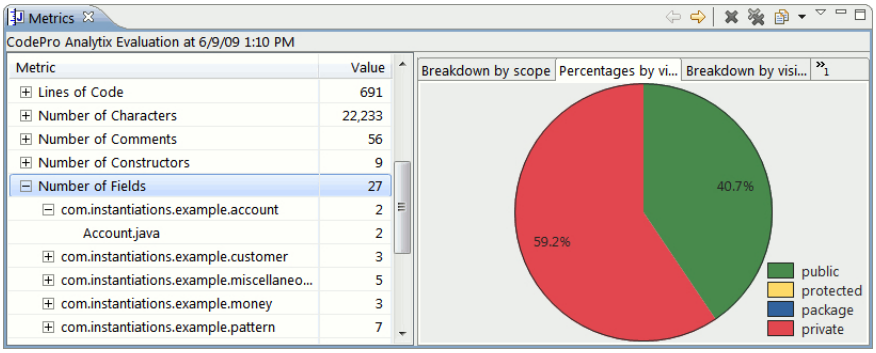

Figura 3.7: Exemplo 2 de Gráfico Exportado por Google (2012)

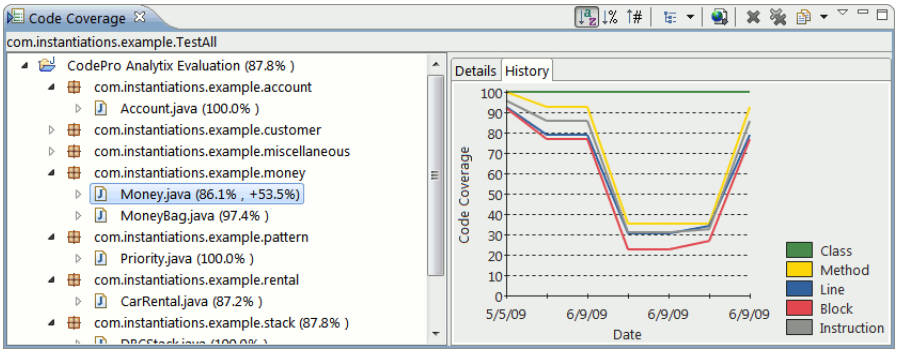

Figura 3.8: Exemplo de Relatório de Cobertura de Código por Google (2012)

vas ao tamanho do código, à coesão e acoplamento, uma vez que esses parâmetros estão diretamente relacionados às definições de código de qualidade (ABRAN *et al.*, 2004; JANZEN; SAIEDIAN, 2008, 2005) e têm um grande impacto nas características de manutenibilidade do produto de *software* (ABRAN *et al.*, 2004; ISO/IEC, 2001), atributo de qualidade abordado neste trabalho.

Estas, foram computadas sobre o código desenvolvido e o código base selecionado para comparação, selecionadas conforme recomendações dos trabalhos de Janzen e Saiedian (2008), Beck (2002), Fowler (1999) e também com base na documentação do *Google Code Pro Analytix* (GOOGLE, 2012).

As métricas selecionadas, foram divididas em três grupos, conforme proposto em (GOOGLE, 2012):

1. **Básicas** (*Basics*)

   (a) **Número Médio de Linhas de Código por Método** (*Average Lines of Code Per Method*)

   (b) **Número Médio de Métodos por Tipo** (*Average Number of Methods Per Type*)

   (c) **Número de Linhas de Código** (*Lines of Code*)

   (d) **Número de Tipos** (*Number of Types*)

2. **Complexidade** (*Complexity*)

   (a) **Profundidade Média de Bloco** (*Average Block Depth*)

   (b) **Complexidade Ciclomática Média** (*Average Cyclomatic Complexity*)

   (c) **Métodos Ponderados** (*Weighted Methods*)

3. **Dependência** (*Dependency*)

   (a) **Acoplamentos Eferentes** (*Efferent Couplings*)

Estas métricas são descritas com mais detalhes nas seções a seguir.

3.4.1 Básicas

3.4.1.1 Número médio de Linhas de Código por Método

Esta métrica reflete o número médio de linhas de código (LOC) em cada método definido na classe que foi foco da metrificação. Esta métrica contém detalhes dos números máximo e mínimo de linhas de código em cada um dos métodos analisados. Valores elevados para essas métricas impactam negativamente no quesito legibilidade do código fonte.

1. Aplicabilidade: esta métrica pode ser aplicada a qualquer classe ou conjunto de classes.

2. Parâmetros:

   (a) Valor Médio Máximo: a métrica destaca qualquer classe ou pacote com uma média de LOC/método além do determinado pelo usuário.

3.4.1.2 Número Médio de Métodos por Tipo

Esta métrica mede o número médio de métodos definidos para cada classe. Assim como a métrica detalhada anteriormente, esta contém detalhes dos números máximo e mínimo de métodos definidos para a classe.

1. Aplicabilidade: esta métrica pode ser aplicada a qualquer classe ou conjunto de classes.

2. Parâmetros:

   (a) Valor Médio Máximo: a métrica destaca qualquer classe ou pacote com uma média de métodos/classe além do limite máximo fornecido pelo usuário.

3.4.1.3 Linhas de Código

Conta o número de linhas de código em uma determianda classe ou counjunto de classes. É importante destacar que esta métrica é diferente da primeira métrica detalhada (“número médio de linhas de código por método”), onde a primeimra mede o número médio de linhas de código por método, e esta, conta o número de linhas total de uma determinada classe ou conjunto de classes.

1. Aplicabilidade: esta métrica pode ser aplicada a qualquer arquivo baseado em linhas de código.

2. Parâmetros:

(a) Máximo de Linhas de Código pro Unidade de Compilação: a métrica destaca qualquer classe ou pacote com LOC além do limite máximo fornecido pelo usuário.

3.4.1.4 Número de Tipos

Esta métrica mede o número de tipos definidos em um grupo de arquivos.

1. Aplicabilidade: esta métrica pode ser computada para qualquer conjunto de arquivos.

2. Parâmetros:

   (a) Número Máximo de Tipos por Pacote: a métrica destaca qualquer pacote com número de tipos além do limite máximo fornecido pelo usuário.

Valores elevados para esta métrica impactam negativamente não só no tamanho e legibilidade do código, mas também na sua complexidade, já que um número maior de tipos implica um número maior de possíveis dependências.

3.4.2 Complexidade

3.4.2.1 Profundidade Média de Bloco

Esta métrica encontra métodos e construtores que têm muitos níveis de blocos aninhados. Este aninhamento de bloco pode dificultar o entendimento e legibilidade do código (GOOGLE, 2012).

1. Aplicabilidade: esta métrica pode ser computada para qualquer método ou conjunto de métodos.

2. Parâmetros:

(a) Número de Sentenças *Try*/*Catch*: determina quando uma sentença "*try* / *catch* / *finally*" é contada como um fator que influencia na profundidade de bloco.

(b) Sentenças sincronizadas: determina quando uma sentença sincronizada é um fator que influencia na profundidade de bloco. Estas sentenças sincronizadas podem ser vistas como uma situação de exclusão mútua, usada para aplicações *multithread*.

(c) Valor Máximo Médio: a métrica destaca qualquer projeto, pacote ou classe com valor de profundidade de bloco além do limite máximo especificado pelo usuário.

3.4.2.2 Complexidade Ciclomática

Esta medida calcula a complexidade ciclomática média de cada método definido na classe, ou conjunto de classes. A complexidade ciclomática de um único método é a medição do número de caminhos distintos de execução dentro do método. É mensurada ao se adicionar um caminho ao método, passando por cada um dos caminhos criados por sentenças condicionais (como "*if*" ou "*for*") e operadores (como "?:").

1. Aplicabilidade: essa métrica pode ser computada para qualquer método ou conjunto de métodos.

2. Parâmetros:

   (a) Número de instâncias de objetos na computação de complexidade: determina quando uma instância de um objeto é contada na computação da métrica.

   (b) Número de instâncias de objetos sem construtores na computação de complexidade: determina quando uma instância de um objeto sem construtores definidos é contado na computação da métrica.

(c) Valor Máximo Médio: a métrica destaca qualquer projeto, pacote ou classe com valor de profundidade de bloco muito alto.

(d) Número de construtores na computação de complexidade: determina quando um construtor é contado na computação da métrica.

(e) Número de Sentenças "*catch*": determina quando uma cláusula "*catch*" é contada na computação da métrica.

(f) Uso de operadores condicionais: determina quando o uso de operadores condicionais é contado na computação da métrica.

(g) Uso de operadores condicionais "&" e "||": determina quando o uso de operadores condicionais "e" e "ou" são contados na computação da métrica.

(h) Valor Máximo Médio: a métrica destaca qualquer projeto, pacote ou classe com valor de complexidade ciclomática muito alta.

3.4.2.3 Métodos Ponderados

Esta é a soma da complexidade ciclomática de cada um dos métodos definidos no objeto de metrificação.

1. Aplicabilidade: essa métrica pode ser computada para qualquer método ou conjunto de métodos

3.4.3 Dependência

3.4.3.1 Acoplamentos eferentes

Esta métrica mede o número de tipos de dentro do objeto que dependem dos tipos de fora.

1. Aplicabilidade: essa métrica pode ser computada para qualquer classe ou conjunto classes

2. Parâmetros:

   (a) Número máximo de tipos com referências externas: destaca qualquer projeto ou pacote com número de referências externas acima do limite definido pelo usuário.

4 ANÁLISE E DISCUSSÃO

Neste capítulo são discutidos os resultados obtidos na metrificação do *software* desenvolvido e do *software* base, bem como a comparação dos mesmos, além as dificuldades encontradas na aplicação da técnica TDD e algumas observações pertinentes aos resultados.

A Figura 4.1 mostra os resultados da computação das métricas, que são discutidos nas seções subsequentes.

Computação das Métricas		
	TDD	Base
Número médio de linhas de código por método	4.14	7.05
Número médio de métodos por tipo	3	4.44
Número de linhas de código	513	1117
Número de tipos	8	18
Profundidade Média de Bloco	0.9	0.88
Complexidade Ciclomática	1.23	1.73
Métodos Ponderados	132	240
Acoplamentos Eferentes	8	17

Figura 4.1: Computação das Métricas

4.1 Metrificação

As métricas escolhidas nesse trabalho foram selecionadas dentre várias outras métricas disponibilizadas pela ferramenta *Code Pro Analytix* para que fosse possível realizar observações sobre três pontos fundamentais para a análise de qualidade de código: tamanho, complexidade e acoplamento.

4.1.1 Impacto no tamanho do código

Estas são as métricas básicas propostas pela divisão feita na documentação da ferramenta *Code Pro Analytix* (GOOGLE, 2012) e apresentadas na Seção 3.4.1. Segundo Janzen e Saiedian (2008), as métricas mais simples são as relativas ao tamanho do código fonte. Os resultados são ilustrados na Figura 4.2, e detalhados nas seções a seguir.

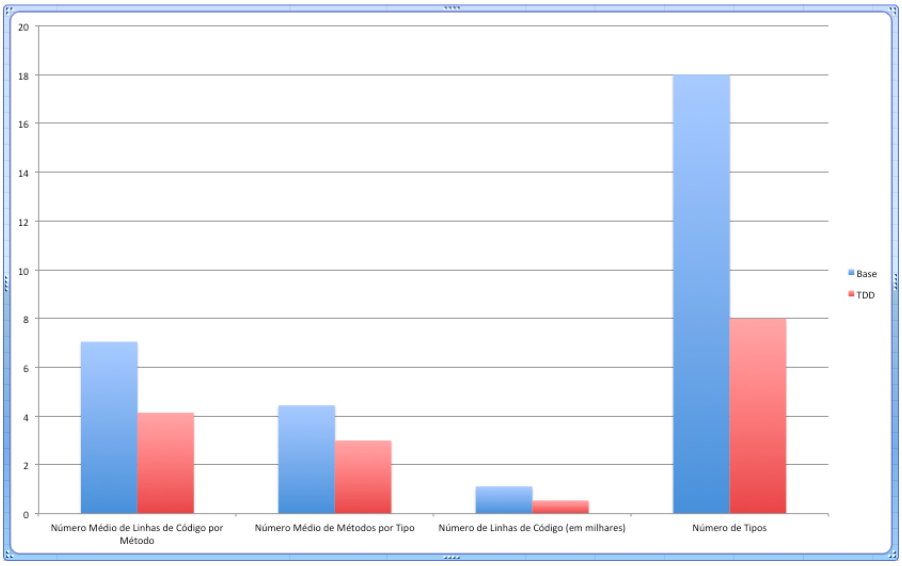

Figura 4.2: Métricas Básicas

Valores para métricas de tamanho de código são relativos, visto que *software* de porte grande terá muitas linhas de código e isso não influenciará necessariamente na sua qualidade. Mas quando se trata de uma comparação, convém utilizá-la.

4.1.1.1 Número médio de linhas de código por método

Para o *software* desenvolvido usando TDD, o número médio de linhas por método encontrado foi de 4.14 linhas por método, onde, quando computada a mesma métrica para o *software* base, o resultado foi de 7.05. Esta diferença se dá devido ao *refactoring* que acontece no ciclo do TDD (BECK, 2002). O *refactoring* tem o objetivo de simplificar o código fonte e excluir qualquer tipo de duplicidade ou redundância encontrados no mesmo. O resultado pode ser visualizado na Figura 4.3, para os valores encontrados no *software* base e, para o método desenvolvido utilizando TDD, na Figura 4.4.

Um exemplo pode ser visualizado nas Figuras 4.5 e 4.6, que representam, respectivamente, os métodos “randomizarNumero( )”, do *software* desenvolvido

Metric	Value
Average Lines Of Code Per Method	7.05
Bonus.java	8.44
Celular.java	6.40
CelularFactory.java	9.00
Cliente.java	5.70
Internet.java	5.45
Ligacao.java	7.77
MinhaException.java	4.50
Minutos.java	6.12
Operadora.java	9.90
Plano.java	3.38
PlanoFactory.java	9.00
PosPago.java	4.87
PrePago.java	5.22
Promocao.java	4.14
PromocaoFactory.java	23.00
RegularPhone.java	10.80
SaldoException.java	4.50
SmartPhone.java	15.00

Figura 4.3: Número Médio de Linhas por Métodos (*Software* base)

Metric	Value
Average Lines Of Code Per Method	4.14
Celular.java	5.64
Cliente.java	3.55
Endereco.java	3.00
Ligacao.java	3.30
PessoaFisica.java	3.00
PessoaJuridica.java	3.00
Plano.java	3.50
Promocao.java	3.65

Figura 4.4: Número Médio de Linhas por Métodos (*Software* TDD)

utilizando TDD, e "gerarNumeroCelular( )", do *software* base. Neste exemplo, apesar das escolhas dos desenvolvedores em relação aos tipos das variáveis, é fácil notar a diferença entre as linhas de código dentro do método.

```
1. private void randomizarNumero() {
2.     Random gerador = new Random();
3.     this.setNumero(gerador.nextInt(100000000));
4. }
```

Figura 4.5: Método "randomizarNumero( )" do *Software* TDD

```
protected static String gerarNumerosCelular(){
    int aux;
    StringBuilder construirNumero = new StringBuilder();
    do{
        aux = 66000000 + (int)(Math.random() * 900000 + 100000);
        construirNumero.replace(0, 10, Integer.toString(aux));
        construirNumero.insert(4, '-');

    }while(Celular.todosNumeros.contains(construirNumero.toString()));
    Celular.todosNumeros.add(construirNumero.toString());
    return construirNumero.toString();
}
```

Figura 4.6: Método “gerarNumeroCelular( )” do *Software* Base

4.1.2 Número médio de métodos por tipo

Na computação desta métrica, os seguintes resultados foram encontrados: para o *software* implementado utilizando TDD foi encontrado o valor médio de 3 métodos por tipo, enquanto que para o *software* base, foi encontrado um valor médio de 4.44 métodos por tipo.

É importante notar que, para o *software* desenvolvido com TDD, apesar de a média ter sido menor no número de métodos, em uma das classes implementadas (Celular.java), encontrou-se o valor de 18 métodos, e em outras foram encontrados valores como 2, ou 3 métodos. Isso pode ser explicado relacionando a presente métrica com a primeira métrica apresentada (número médio de linhas de código por método). O uso do TDD influenciou na criação de métodos menores, com menor responsabilidade, mais coesos, então, consequentemente, foi necessária a construção de um maior número de métodos para a implementação de determinadas funcionalidades. Os resultados podem ser visualizados na Figura 4.7 para o *software* base e na Figura 4.8 para o *software* onde foi utilizado o TDD.

Metric	Value
Average Number of Methods Per Type	4.44
Bonus.java	3.00
Celular.java	11.00
CelularFactory.java	1.00
Cliente.java	4.00
Internet.java	4.00
Ligacao.java	3.00
MinhaException.java	1.00
Minutos.java	3.00
Operadora.java	16.00
Plano.java	9.00
PlanoFactory.java	1.00
PosPago.java	5.00
PrePago.java	5.00
Promocao.java	6.00
PromocaoFactory.java	1.00
RegularPhone.java	3.00
SaldoException.java	1.00
SmartPhone.java	3.00

Figura 4.7: Número Médio de Métodos por Tipo (*Software* base)

Metric	Value
Average Number of Methods Per Type	3.00
Celular.java	18.00
Cliente.java	1.00
Plano.java	3.00
Promocao.java	2.00

Figura 4.8: Número Médio de Métodos por Tipo (*Software* TDD)

4.1.3 Número de linhas de código

Para o número de linhas de código, assim como nas duas métricas anteriores, o número encontrado utilizando TDD foi inferior ao número encontrado na computação das métricas para o *software* base, sendo, respectivamente, os valores 513 e 1117. Os valores podem ser vizualizados na Figura 4.9 para o *software* base, e Figura 4.10 para o *software* que utilizou TDD.

Metric	Value
Lines of Code	1,117
Bonus.java	87
Celular.java	107
CelularFactory.java	13
Cliente.java	62
Internet.java	68
Ligacao.java	79
MinhaException.java	14
Minutos.java	59
Operadora.java	233
Plano.java	53
PlanoFactory.java	13
PosPago.java	48
PrePago.java	58
Promocao.java	36
PromocaoFactory.java	28
RegularPhone.java	61
SaldoException.java	16
SmartPhone.java	82

Figura 4.9: Número De Linhas de Código (*Software* base)

Metric	Value
Lines of Code	513
Celular.java	206
Cliente.java	39
Endereco.java	24
Ligacao.java	52
PessoaFisica.java	10
PessoaJuridica.java	10
Plano.java	75
Promocao.java	97

Figura 4.10: Número De Linhas de Código (*Software* TDD)

4.1.4 Número de tipos

O número de tipos também é influenciado pelo uso da técnica proposta, uma vez que ela influencia no *design* de classes de acordo com o andamento da codificação (ANICHE, 2012). Para o código do *software* base, foi encontrado um número de 18 tipos, onde, para o *software* desenvolvido utilizando TDD, encontrou-se um valor de 8 tipos. Os tipos podem ser vizualizados na Figura 4.11 e Figura 4.12, para o *software* base e para o *software* proposto, respectivamente.

Vale ressaltar que, no uso da técnica TDD, existe uma tendência de que o desenvolvedor crie um número maior de classes (JANZEN; SAIEDIAN, 2008), porém, devido ao fato do sistema desenvolvido neste estudo ser de pequeno porte, esta tendência não foi observada.

Number of Types	18
Bonus.java	1
Celular.java	1
CelularFactory.java	1
Cliente.java	1
Internet.java	1
Ligacao.java	1
MinhaException.java	1
Minutos.java	1
Operadora.java	1
Plano.java	1
PlanoFactory.java	1
PosPago.java	1
PrePago.java	1
Promocao.java	1
PromocaoFactory.java	1
RegularPhone.java	1
SaldoException.java	1
SmartPhone.java	1

Figura 4.11: Número De Tipos (*Software* base)

Number of Types	8
Celular.java	1
Cliente.java	1
Endereco.java	1
Ligacao.java	1
PessoaFisica.java	1
PessoaJuridica.java	1
Plano.java	1
Promocao.java	1

Figura 4.12: Número De Tipos (*Software* TDD)

4.2 Impacto na Complexidade

Tamanho é uma medida de complexidade. Métodos e classes menores são, geralmente, mais simples e mais inteligíveis (JANZEN; SAIEDIAN, 2008). Esta seção demonstra os ganhos obtidos com o uso da técnica TDD em relação às métricas: Profundidade Média de Bloco, Complexidade Ciclomática e Métodos Ponderados. Como proposto, uma das metas para se obter código de qualidade, é reduzir a complexidade do mesmo. Os resultados da computação das métricas desse grupo mostram que o uso de TDD gerou resultados satisfatórios quanto ao quesito complexidade. A Figura 4.13 ilustra esta situação.

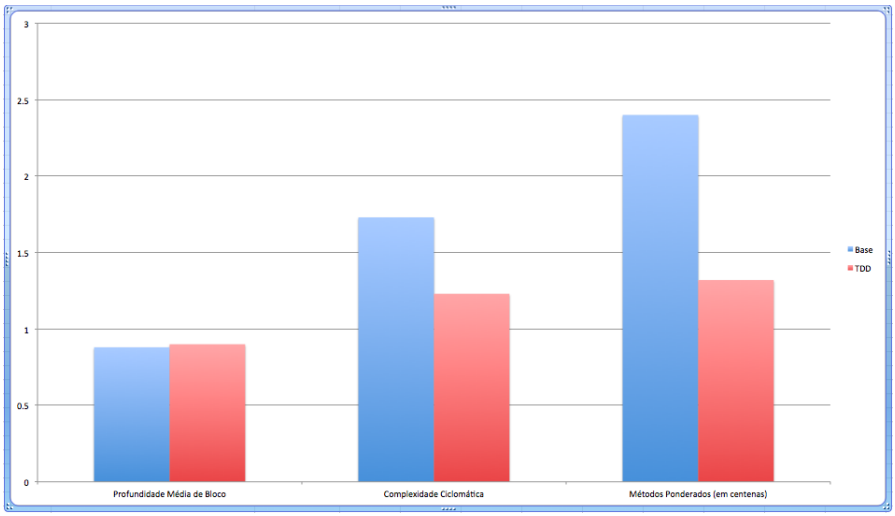

Figura 4.13: Métricas de Complexidade

4.2.1 Profundidade média de bloco

Para profundidade média de bloco, um valor ligeiramente maior foi encontrado para o *software* desenvolvido com TDD. Isso se explica pelo fato de que uma classe foi ignorada durante a computação das métricas para o *software* base. Esta situação será detalhada na última seção deste capítulo.

A Figura 4.14 e Figura 4.15 ilustram os resultados para cada classe dos *softwares*.

Average Block Depth	0.88
celular	0.88
SmartPhone.java	2.00
SaldoException.java	0.50
RegularPhone.java	1.60
PromocaoFactory.java	2.00
Promocao.java	0.50
PrePago.java	0.81
PosPago.java	0.88
PlanoFactory.java	1.00
Plano.java	0.53
Operadora.java	1.14
Minutos.java	0.81
MinhaException.java	0.66
Ligacao.java	0.76
Internet.java	0.75
Cliente.java	0.84
CelularFactory.java	1.00
Celular.java	0.80
Bonus.java	0.84

Figura 4.14: Profundidade Média de Bloco - (*software* base)

Average Block Depth	0.90
lib	0.90
Promocao.java	0.71
Plano.java	0.75
PessoaJuridica.java	0.66
PessoaFisica.java	0.66
Ligacao.java	0.72
Endereco.java	0.66
Cliente.java	0.83
Celular.java	1.31

Figura 4.15: Profundidade Média de Bloco - (*software* TDD)

4.2.2 Complexidade ciclomática

Para a métrica de complexidade ciclomática, foram encontrados, para o *software* base, e para o *software* proposto, os valores 1.73 e 1.23, respectivamente.

Como o TDD propõe que o desenvolvedor escreva métodos menores, com menos responsabilidades, devido ao processo iterativo de desenvolvimento e ao *refactor*, a complexidade ciclomática média do projeto é influenciada, pois, com métodos menores, o número de caminhos possíveis, dentro do método, é menor.

Esta situação pode ser visualizada também nas Figuras 4.5 e 4.6, onde nota-se a diferença não só no tamanho dos métodos, mas também nas responsabilidades de cada um deles.

O TDD guia o desenvolvimento dos métodos de maneira que os mesmos tenham responsabilidades pequenas. Neste exemplo, pode-se perceber que o método "gerarNumeroCelular( )" tem como responsabilidades gerar o número aleatório do celular, organizar o retorno do método em uma *string*, de modo que fique amigável ao usuário, e certificar-se de que não exista nenhum outro número igual, já o método "randomizarNumero( )" apenas gera um número aleatório para o celular, deixando as outras responsabilidades para outros métodos.

Os valores detalhados para cada uma das classes pode ser detalhado nas Figuras 4.16 e 4.17.

4.2.3 Métodos ponderados

Os resultados encontrados para o número de métodos ponderados, foram, para o *software* base e *software* TDD, respectivamente, 240 e 132. É natural que, como consequência dos valores encontrados na metrificação de complexidade ciclomática, haja uma diferença considerável nos valores encontrados para os métodos ponderados, visto que esta métrica é dependente da métrica anterior. Estes valores podem ser vistos com mais detalhes em Figura 4.18 e 4.19.

Average Cyclomatic Complexity	1.73
celular	1.73
SmartPhone.java	3.60
SaldoException.java	1.00
RegularPhone.java	2.60
PromocaoFactory.java	5.00
Promocao.java	2.00
PrePago.java	1.00
PosPago.java	1.00
PlanoFactory.java	3.00
Plano.java	1.53
Operadora.java	1.95
Minutos.java	1.12
MinhaException.java	1.00
Ligacao.java	2.22
Internet.java	1.09
Cliente.java	1.70
CelularFactory.java	3.00
Celular.java	1.80
Bonus.java	1.66

Figura 4.16: Complexidade Ciclonática Média - (*software* base)

Average Cyclomatic Complexity	1.23
lib	1.23
Promocao.java	1.08
Plano.java	1.00
PessoaJuridica.java	1.00
PessoaFisica.java	1.00
Ligacao.java	1.00
Endereco.java	1.00
Cliente.java	1.00
Celular.java	1.67

Figura 4.17: Complexidade Ciclomática Média - (*software* TDD)

4.3 Impacto no Acoplamento

Como citado nas seções anteriores, existe uma tendência de que os desenvolvedores "*test-first*" criem soluções com mais classes, classes menores, podendo gerar assim muitas conexões entre essas classes. Porém, ainda assim, foram encontrados valores menores para o *software* implementado utilizando o TDD.

Weighted Methods	240
celular	240
SmartPhone.java	18
SaldoException.java	2
RegularPhone.java	13
PromocaoFactory.java	5
Promocao.java	14
PrePago.java	9
PosPago.java	8
PlanoFactory.java	3
Plano.java	20
Operadora.java	43
Minutos.java	9
MinhaException.java	2
Ligacao.java	20
Internet.java	12
Cliente.java	17
CelularFactory.java	3
Celular.java	27
Bonus.java	15

Figura 4.18: Métodos Ponderados - (*software* base)

Weighted Methods	132
lib	132
Promocao.java	25
Plano.java	18
PessoaJuridica.java	2
PessoaFisica.java	2
Ligacao.java	13
Endereco.java	6
Cliente.java	9
Celular.java	57

Figura 4.19: Métodos Ponderados - (*software* TDD)

4.3.1 Acoplamentos eferentes

Para esta métrica, foram encontrados os valores 17 e 8, para o *software* base e o *software* desenvolvido, respectivamente. Isso significa que, foram encontradas 17 situações no *software* base, onde foi necessário fazer o acesso de um

método ou objeto, de dentro do objeto foco da metrificação, onde, no *software* desenvolvido com o uso da técnica proposta, essa situação foi encontrada 8 vezes.

4.4 Considerações Finais

É válido destacar alguns pontos importantes quanto às considerações acerca das métricas aplicadas, do código-fonte produzido e da técnica utilizada, bem como destacar os problemas encontrados.

1. Em vários momentos no desenvolvimento do *software*, se fez necessária a reutilização de métodos já implementados e o uso do *refactor*, dois fatores que tiveram influência significativa no resultado da computação das métricas escolhidas.

2. Como o TDD é um processo iterativo (BECK, 2002), o *software* foi feito de maneira incremental, com etapas curtas, buscando sempre testar a menor parte possível de determinada funcionalidade. Isso torna o *software* mais modularizado e com testes que referenciam apenas uma classe.

3. A computação da métrica "Profundidade Média de Bloco" foi prejudicada no trabalho devido a uma classe pertencente ao *software* base que foi ignorada durante a metrificação. Esta classe é responsável pela interação com o usuário. Ela foi desconsiderada devido à grande quantidade de código-fonte produzido para a construção de um menu de interação com o usuário, usando estruturas tais como *switches*, métodos de manipulação de arquivos, etc.. Esta classe de interface também contém parte da lógica do sistema, que, no *software* desenvolvido com a técnica TDD, é implementada nas classes de negócio do sistema. Se esta classe fosse considerada na computação das métricas, todos os resultados anteriormente descritos seriam alterados, inclusive a profundidade média de bloco, tornando o resultado encontrado favorável ao *software* desenvolvido utilizando o TDD. As Figuras 4.20, 4.21

e 4.22 mostram os resultados da computação das métricas para, respectivamente, o *software* base, considerando a classe de interação com o usuário, o *software* base sem a classe de interação com o usuário, e para o *software* desenvolvido com o uso de TDD.

4. Neste ponto do desenvolvimento do trabalho, foi possível notar as características proporcionadas pelo uso do TDD. Notou-se que cada método implementado foi feito com um único objetivo, fazendo apenas aquilo que foi projetado para fazer.

Metric	Value
Average Block Depth	0.98
Average Cyclomatic Complexity	2.69
Average Lines Of Code Per Method	11.75
Average Number of Methods Per Type	4.57
Efferent Couplings	18
Lines of Code	1,858
Number of Types	19
Weighted Methods	391

Figura 4.20: Resultado das métricas para o *Software* base considerando a classe de interação com o usuário

Metric	Value
Average Block Depth	0.88
Average Cyclomatic Complexity	1.73
Average Lines Of Code Per Method	7.05
Average Number of Methods Per Type	4.44
Efferent Couplings	0
Lines of Code	1,117
Number of Types	18
Weighted Methods	240

Figura 4.21: Resultado das métricas para o *Software* base sem considerar a classe de interação com o usuário

Metric	Value
Average Block Depth	0.90
Average Cyclomatic Complexity	1.23
Average Lines Of Code Per Method	4.14
Average Number of Methods Per Type	3.00
Efferent Couplings	0
Lines of Code	513
Number of Types	8
Weighted Methods	132

Figura 4.22: Resultado das métricas para o *Software* TDD

5 CONCLUSÃO

Buscar a melhoria da qualidade do código-fonte é uma tarefa que exige muito mais que apenas vontade por parte das equipes. Realizando o estudo em um trabalho didático, envolvendo apenas dois desenvolvedores, é muito diferente de fazer o mesmo em uma empresa de desenvolvimento, com equipes grandes, onde os prazos são curtos e, muitas vezes, é difícil encontrar tempo para treinamentos e adaptações.

O desenvolvimento do trabalho permitiu perceber que o uso da abordagem TDD não é uma tarefa simples, principalmente no começo, pois inverte a relação tradicional entre desenvolvimento-teste. Além disso, uma vez que o desenvolvedor começa a se acostumar com os passos do TDD, seguir pequenas iterações se torna mais difícil, já que o desenvolvedor tende a pular alguns passos.

Apesar disso, foi possível concluir que os beneficios do uso da técnica são percebidos rapidamente, principalmente no que diz respeito à responsabilidade de cada método, devido ao fato do teste já estar pronto e esperando um determinado resultado. Comparando os códigos dos dois *softwares*, foi possível, de acordo com os resultados da computação das métricas, encontrar valores favoráveis em relação ao código em que se utilizou o TDD.

No final, obteve-se um *software* onde todas as métricas aplicadas apresentaram resultados melhores, o que aponta melhorias na qualidade do código produzido, corroborando os conhecimentos já existentes sobre a abordagem TDD, segundo o trabalho de Janzen e Saiedian (2008). Porém, mesmo com bons resultados, vale destacar que este estudo não possui peso estatístico; conclusões fundamentadas sobre as vantagens obtidas pelo uso do TDD, em termos de melhoria de código fonte, poderiam ser obtidas replicando a metodologia deste estudo para um *software* de grande porte e analisando estatisticamente o nível de significância dos resultados obtidos com as métricas.

5.1 Trabalhos Futuros

Pode-se citar, como possibilidades para trabalhos futuros, a replicação do presente estudo, com números maiores de equipes de desenvolvimento, tanto para a construção do *software* utilizando a técnica TDD, quanto para a construção do *software* utilizando metodologias tradicionais de desenvolvimento, para se obter volume de código suficiente para um estudo estatístico da aplicação da técnica. Ainda, pode ser proposta a aplicação da técnica para um *software* real, de tamanho e complexidade maiores, visto que, no presente trabalho, o problema proposto foi um sistema para fins didáticos.

Outro possível trabalho, seria o envolvimento de desenvolvedores que não fizeram parte do desenvolvimento dos *softwares* produzidos, para avaliar a manutenibilidade dos mesmos. Assim, seria possível concluir se a manutenibilidade do *software* foi influenciada pelo uso da técnica.

Por fim, analisando a técnica com um enfoque no ensino de fundamentos de desenvolvimento de *software*, talvez fosse possível propor uma metodologia de ensino, onde o desenvolvimento de *software* seria ensinado com uma abordagem "*test-first*", e não "*test-last*".

REFERÊNCIAS BIBLIOGRÁFICAS

ABRAN, A.; BOURQUE, P.; DUPUIS, R.; MOORE, J. W.; TRIPP, L. L. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. 2004 version. ed. Piscataway, NJ, USA: IEEE Press, 2004. 1–202 p. ISBN 0769510000. Disponível em: <http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf>.

ANICHE, M. *TEST-DRIVEN DEVELOPMENT: TESTE E DESIGN NO MUNDO REAL*. [S.l.]: CASA DO CODIGO, 2012. ISBN 9788566250046.

Associação Brasileira de Normas Técnicas. *Tecnologia da Informação – Processo de Ciclo de Vida do Software*. [S.l.], 1998.

ASTELS, D. *Test-driven development: a practical guide*. [S.l.]: Prentice Hall PTR, 2003. (The Coad series). ISBN 9780131016491.

BECK, K. *Extreme Programming Explained: Embrace Change*. [S.l.]: Addison Wesley Professional, 2000. (An Alan R. Apt Book Series). ISBN 9780201616415.

BECK, K. *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321146530.

CANFORA, G.; VISAGGIO, A. Measuring the impact of testing on code structure in test driven development: metrics and empirical analysis. *First International Symposium on emerging trends in software metrics*, 2009.

CROSBY, P. *Quality is free: the art of making quality certain*. [S.l.]: McGraw-Hill, 1979. ISBN 9780070145122.

DUARTE, K. C.; FALBO, R. de A. *Uma Ontologia de Qualidade de Software Uma Ontologia de Qualidade de Software Uma Ontologia de Qualidade de Software*. Tese (Doutorado) — UFES - Universidade Federal do Espírito Santo, 2000.

DUVALL, P.; MATYAS, S. M.; GLOVER, A. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. [S.l.]: Addison-Wesley Professional, 2007. ISBN 0321336380.

ECLIPSE FOUNDATION. *Eclipse*. [S.l.], 2012. Disponível em: <http://www.eclipse.org/>.

FOWLER, M. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999. ISBN 0-201-48567-2.

GOOGLE. *Google Code Pro Analytix*. 2012. Disponível em: <https://developers.google.com/java-dev-tools/codepro/doc/>.

HANNA, M. Attention to process ups software quality. *Softw. Mag.*, Wiesner Publishing, LLC, Englewood, CO, USA, v. 13, n. 18, p. 43–ff., dez. 1993. ISSN 0897-8085.

HUMPHREY, W. S. *Managing the software process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0-201-18095-2.

ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. [S.l.]: ISO/IEC, 2001.

JANZEN, D.; SAIEDIAN, H. Test-driven development: Concepts, taxonomy, and future direction. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 38, n. 9, p. 43–50, set. 2005. ISSN 0018-9162.

JANZEN, D.; SAIEDIAN, H. Does test-driven development really improve software design quality? *Software, IEEE*, v. 25, n. 2, p. 77–84, 2008. ISSN 0740-7459.

JUNG, C. F. *Metodologia aplicada a projetos de pesquisa: Sistemas de Informação Ciência da Computação*. [S.l.]: Taquara, 2009.

JUNIT TEAM. *JUnit*. [S.l.], 2012. Disponível em: <http://junit.org/>.

KOSCIANSKI, A.; SOARES, M. dos S. *Qualidade de software: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software*. [S.l.]: Novatec Editora, 2007. ISBN 9788575221129.

LANZA, M.; MARINESCU, R.; DUCASSE, S. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. [S.l.]: Springer-Verlag Berlin and Heidelberg GmbH & Company KG, 2007. ISBN 9783540395386.

MüLLER, M.; HöFER, A. The effect of experience on the test-driven development process. *Empirical Software Engineering*, Springer Netherlands, v. 12, n. 6, p. 593–615, dez. 2007. ISSN 1382-3256.

ORACLE. *Java*. [S.l.], 1995. Disponível em: <www.oracle.com/technetwork/java/index.html>.

PIGOSKI, T. *Practical software maintenance: best practices for managing your software investment*. [S.l.]: Wiley Computer Pub., 1997. (Wiley computer publishing). ISBN 9780471170013.

POPPENDIECK, M.; POPPENDIECK, T. *Lean Software Development: An Agile Toolkit*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321150783.

PRESSMAN, R. *Software engineering: a practitioner's approach*. [S.l.]: McGraw-Hill Higher Education, 2010. (McGraw-Hill higher education). ISBN 9780073375977.

RIAZ, M.; MENDES, E.; TEMPERO, E. A systematic review of software maintainability prediction and metrics. In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. Washington, DC, USA: IEEE Computer Society, 2009. (ESEM '09), p. 367–377. ISBN 978-1-4244-4842-5.

SNEED, H. M.; BRÖSSLER, P. Critical success factors in software maintenance-a case study. In: *ICSM*. [S.l.]: IEEE Computer Society, 2003. p. 190–198. ISBN 0-7695-1905-9.

SOMMERVILLE, I. *Software Engineering*. [S.l.]: Addison-Wesley, 2007. (International Computer Science Series). ISBN 9780321313799.

TORCHIANO, M.; SILLITTI, A. Tdd = too dumb developers? implications of test-driven development on maintainability and comprehension of software. In: *ICPC*. [S.l.]: IEEE Computer Society, 2009. p. 280–282.

VU, J. H.; FROJD, N.; SHENKEL-THEROLF, C.; JAZEN, D. Evaluating test-driven development in a industry-sponsored capstone project. *Sixth International Conference on Information Technology: New Generations*, 2009.

WASMUS, H.; GROSS, H.-G. Evaluation of test-driven development - an industrial case study. In: GONZALEZ-PEREZ, C.; MACIASZEK, L. A. (Ed.). *ENASE*. [S.l.]: INSTICC Press, 2007. p. 103–110. ISBN 978-989-8111-10-4.

ZEISS, B.; VEGA, D.; SCHIEFERDECKER, I.; NEUKIRCHEN, H.; GRABOWSKI, J. Applying the iso 9126 quality model to test specifications – exemplified for ttcn-3 test specifications. In: *Software Engineering*. [S.l.: s.n.], 2007. p. 231–242.

A TRABALHO PRÁTICO UTILIZADO

Universidade Federal de Lavras Departamento de Ciência da Computação
GCC – 110 – Programação Orientada a Objetos

Trabalho Prático 2:

Sistema de Operadora de Celular

Implemente um conjunto de classes para um sistema de uma operadora de telefones celulares. A operadora necessita manter informações sobre seus celulares, seus clientes e suas contas. Uma descrição do funcionamento do sistema é dada abaixo:

- ·A operadora mantém as seguintes informações sobre seus clientes: *cpf* (ou *cnpj*), nome e endereço. Ao adquirir um celular, o cliente adere-se a um dos planos da operadora. Cada plano possui um nome e define um valor a ser cobrado por minuto de ligação. Por simplicidade do trabalho, considere que cada plano tem um único valor, independente do horário e do local da ligação;

- ·Um cliente pode possuir diversos celulares, mas um celular deve pertencer a apenas um cliente;

- ·Cada celular vendido pela operadora possui um número e está associado a um cliente e a um plano e, possui uma lista de ligações. Os celulares são classificados em *SmartPhones* ou *RegularPhones*, para efeitos de promoções que a operadora possa oferecer em seus planos.

- ·Os planos da operadora são do tipo cartão (pré-pago) ou assinatura (pós-pago). Para um plano do tipo cartão, a operadora deve controlar o valor dos créditos restantes e a data de validade desses créditos. Para um plano do tipo assinatura, a operadora deve controlar dia (do mês) de vencimento da fatura;

- ·Para cada ligação efetuada de um celular, a operadora registra a data/hora da ligação e a sua duração, em minutos. Uma ligação pode ser do tipo simples (chamada comum) ou ligação de Internet (consumo de pacotes proporcionais para Internet);

- ·Cada plano possui uma lista de promoções ofertadas de acordo com o tipo de plano e o tipo do celular. As promoções podem ser de três tipos: “Internet”, “Minutos” e “Bônus”.

- Internet: ofertada para celulares do tipo *SmartPhone*, habilitados em ambos os tipos de plano: pós e pré-pago. Uma promoção deste tipo deve possuir as seguintes informações: nome da promoção, velocidade, franquia, velocidade além da franquia.

- Minutos: ofertada para celulares de ambos os tipos (*SmartPhone* e *RegularPhone*) habilitados no plano pós-pago. Uma promoção deste tipo deve possuir as seguintes informações: nome da promoção, validade e quantidade.

- Bônus: ofertada para celulares de ambos os tipos (*Smartphone* e *Regularphone*) habilitados no plano pré-pago. Uma promoção deste tipo deve possuir as seguintes informações: nome da promoção, validade, quantidade e limite-diário.

O sistema deve possuir as seguintes opções:

- ·Cadastrar clientes e cadastrar planos. O usuário fornece os dados do cliente ou do plano;

- ·Cadastrar promoção. O usuário informa o tipo da promoção e os dados referentes a ela.

- ·Habilitar (criar um novo) um celular. O usuário escolhe o tipo de celular (*SmartPhone* ou *RegularPhone*), o tipo do plano (cartão ou assinatura), promoções (entre as disponíveis para o modelo de telefone e plano) e fornece o dia de vencimento da fatura (para celular habilitado em plano de assinatura). O número do celular é gerado automaticamente pelo sistema;

- ·Adicionar promoção. O usuário informa o número de celular. O sistema exibe as promoções disponíveis (que ainda não façam parte) que podem ser adicionadas àquele aparelho/plano.

- ·Adicionar créditos para celular habilitado no plano cartão. O usuário fornece o número do celular e o valor dos créditos a adicionar. O valor dos créditos é somado aos créditos atuais e a data de validade dos créditos é atualizada para 180 dias após a data corrente do sistema;

- ·Registrar ligação. O usuário fornece o número do celular, a data/hora da ligação e a sua duração, em minutos. No caso de celular habilitado no plano de cartão, o sistema deve gerar uma exceção se os créditos não forem suficientes para o tempo da ligação ou se a data de validade dos créditos estiver vencida. Por simplicidade do trabalho, a ligação é registrada após a sua ocorrência. Em um caso real, deveria haver opção para iniciar a ligação, para adicionar tempo de ligação, verificar tempo de crédito e para finalizar a ligação; tudo ocorrendo em tempo real; O sistema deve também levar em conta as promoções cadastradas para aquele aparelho para fins de contabilização dos minutos cobrados.

- ·Ligações do tipo Internet devem ser contabilizadas segundo o saldo do pacote promocional do plano. Uma exceção deve ser lançada caso haja a tentativa de registro de uma ligação do tipo Internet em um aparelho que não contemple esse tipo de ligação em seu plano.

- ·Listar os dados referente a Internet. O usuário fornece o número do celular. O sistema verifica a existência de uma promoção do tipo Internet cadastrada para o número fornecido. Caso exista, deve ser exibida a franquia (franquia da promoção – total gasto) e a velocidade atual (velocidade da promoção ou velocidade além da franquia).

- ·Listar o valor da conta para celular habilitado no plano de assinatura. O usuário fornece o número do celular. O sistema imprime na tela o valor total das ligações ocorridas após o dia de vencimento do mês anterior e o dia de vencimento do mês corrente, para o celular informado; O sistema deve também levar em conta as promoções cadastradas para este celular para fins de contabilização dos minutos.

- ·Listar o valor dos créditos e a data de validade para celular de cartão. O usuário fornece o número do celular. O sistema imprime na tela os dados do celular informado;

- ·Listar extrato de ligações. O usuário fornece o número do celular e uma data. O sistema imprime na tela todas as ligações efetuadas do celular a partir da data informada. Para cada ligação, devem ser impressos a data, a duração da ligação e o valor cobrado;

- ·Listar clientes, listar planos, listar promoções e listar celulares. O sistema imprime na tela todos os dados de cada cliente ou plano ou celular.

- ·Informativo de vencimento de fatura/créditos/promoções. Sempre que um desses itens alcançar a data de vencimento, o sistema deve informar ao usuário os dados do cliente e celular cujos item venceu.

- ·Verificar validade das promoções. O usuário informa o número do celular. O sistema exibe as promoções vinculadas àquele aparelho com as respectivas datas de vencimento.