RAMON HENRIQUE GONÇALVES

USO DE TDD PARA CONSTRUÇÃO DE CÓDIGO FONTE ORIENTADO A OBJETOS MANUTENÍVEL

LAVRAS - MG
2011

RAMON HENRIQUE GONÇALVES

USO DE TDD PARA CONSTRUÇÃO DE CÓDIGO FONTE ORIENTADO A OBJETOS MANUTENÍVEL

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 concentração em Engenharia de Software, para obtenção do título de Bacharel em Sistemas de Informação

Orientador
Dr. Heitor Augustus Xavier Costa

Co-Orientador
Bel. Igor Ribeiro Lima

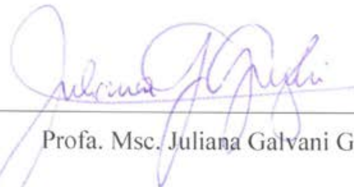
LAVRAS - MG
2011

RAMON HENRIQUE GONÇALVES

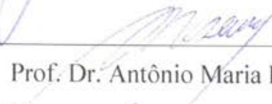
USO DE TDD PARA CONSTRUÇÃO DE CÓDIGO FONTE ORIENTADO A OBJETOS MANUTENÍVEL

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 para obtenção do título de Bacharel em Sistemas de Informação.

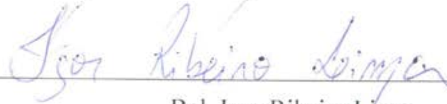
Aprovada em 24/11/2011



Profa. Msc. Juliana Galvani Greghi



Prof. Dr. Antônio Maria Pereira de Resende



Bel. Igor Ribeiro Lima
(Co-Orientador)

Prof. Dr. Heitor Augustus Xavier Costa
(Orientador)

LAVRAS
MINAS GERAIS - BRASIL

"É exatamente disso que a vida é feita, de MOMENTOS. Momentos que TEMOS que passar, sendo bons ou ruins, para o nosso próprio aprendizado. Nunca nos esquecendo do mais importante: _Nada nessa vida é por acaso. Absolutamente nada. Por isso, temos que nos preocupar em fazer a nossa parte, da melhor forma possível.
A vida nem sempre segue a nossa vontade, mas ela é perfeita naquilo que tem que ser."

Chico Xavier

AGRADECIMENTOS

Gostaria de agradecer a todas as pessoas que estão ao meu redor. Sem pessoas nossa alma não funciona, sem esta ligação que nos motiva a seguir, nada faz sentido.

Contudo, devo fazer agradecimentos especiais. Primeiramente, óbvio, devo agradecer a meus país, Adenilson e Elizabeth, por terem me dado a condição de concluir algo que poucos tem a oportunidade e, além de tudo, suportar reclamações em diversos momentos.

Devo agradecer também ao meu orientador Heitor. Sua exigência e rigidez fizeram desse trabalho plenamente possível. Eu sei de minhas condições e garanto que caso tivesse sido guiado de outra maneira, teria problemas. O Heitor soube como orientar e conseguiu extrair o melhor de mim em uma área que eu tenho dificuldades: escrever. O resultado disso tudo foram os elogios ao final, e isso é impagável. Muito obrigado mesmo, quando a gente olha pra trás e analisa os pontos que foram ligados, percebemos a diferença que fez.

Ao coorientador e novo amigo Igor, que foi quem praticamente iniciou este trabalho.

Aos amigos que me acompanharam durante estes anos de graduação e pude compartilhar tudo que foi de melhor na vida, e sem vocês não teria tido graça: Leandro, Daniel, Bachim, Netto, Mariana, PG, Gudinho (Leandro 2), Kuruma, Álvaro, Willian’s... muito obrigado pela presença.

Aos demais companheiros que convivi durante toda a graduação, obrigado por momentos inusitados que não sairão mais de minha memória, principalmente toda a turma do LabGTI.

Quero agradecer também ao professor Gonzaga, que mesmo não tendo feito parte deste trabalho em si, contribuiu durante toda a minha graduação para que eu fosse capaz de desenvolver minhas habilidades profissionais. Muito obrigado.

Por fim, para que fique mais recente na mente de quem estiver lendo, agradeço a minha namorada Karen. A importância dela na minha vida não pode ser descrita nem mensurada, mas há de se registrar que sem ela nada que eu penso ou faço faria sentido. Ela que procurou me ajudar de todas as formas para que eu concluísse este trabalho, inclusive quando eu pensava que não conseguiria, merece metade dos créditos ☺.

RESUMO

A atividade de manutenção é realizada visando a alteração da estrutura do software garantindo sua integridade. Porém esta tarefa pode tornar-se mais difícil caso o código-fonte e a arquitetura interna do software seja de difícil entendimento. Para melhorar o entendimento do código existe uma prática de desenvolvimento que procura tornar o código mais simples, denominada *Test-Driven Development* – TDD. Sugerindo um estilo de desenvolvimento passo a passo e com os testes de cada comportamento sendo escrito antes de cada funcionalidade, o TDD leva o desenvolvedor a construir um código simples e funcional. Neste trabalho esta prática de desenvolvimento é aplicada para o desenvolvimento de um software com o objetivo de alcançar um maior nível de manutenbilidade. Métricas de software foram aplicadas sobre o código-fonte gerado com o uso do TDD e estes valores foram comparados a valores das mesmas métricas referentes a um software previamente existente mas que não havia sido construído com o auxílio do TDD. Os resultados desta comparação são apresentados e mostraram as melhorias em relação à complexidade que se esperava ao se utilizar TDD para construir o software.

Palavras-chave: TDD. Manutenibilidade. Manutenção de Software. Teste de Software. Complexidade de Código.

ABSTRACT

The objective of this work was to use *Test-Driven Development* (TDD) — a technique employed to make the source code more intelligible and easier to maintain. The maintenance is done with the aim of altering the software structure thus guaranteeing its integrity. Nevertheless, this task may become more complex should the source code and the software internal architecture turn out being difficult to understand. As TDD suggests a step by step development style and with the behavior tests being written before each functionality, it leads the developer to build a simpler and more functional code. Such development technique is employed here on the software designing which will confer a higher maintainability level. Sofware metrics was applied over the developed code through TDD and its data were compared with those metrics data of previously existing software which hadn't employed TDD on its designing. The results of this comparison are presented and show the improvements in relation to the expected complexity when using TDD to help build software.

Keywords: Test-Driven Development. Maintainability. Software Maintenance. Software Test. Code Complexity.

LISTA DE FIGURAS

Figura 1 Tipos de pesquisa científica..17
Figura 2 Características de qualidade da norma ISO/IEC 9126 30
Figura 3 Exemplo de Uso de *Mock Objects*..44
Figura 4 Ciclo da Técnica TDD (Adaptado de Brief, 2001)...............................49
Figura 5 Grafo de Complexidade Ciclomática 59
Figura 6 Casos de uso do software simulador de bolsa de valores 64
Figura 7 Injeção de Dependência em uma Classe e Utilização do Padrão de Projetos *Factory*...71
Figura 8 Utilização do Padrão de Projetos *Factory* e do Polimorfismo Presente no Controlador 72
Figura 9 Trecho dos Códigos Fonte para Testar as Funções do Controlador *Datadays* 73
Figura 10 Visão Geral da Análise de Métricas no Google CodePro Analytix....76
Figura 11 Visualização de Dependências Utilizando o Google CodePro Analytix 77
Figura 12 Medidas do Código Fonte do Software Legado 78
Figura 13 Medidas do Código Fonte do Software Desenvolvido Utilizando a Técnica TDD...78
Figura 14 Valores do Código Fonte dos Dois Softwares para *Weighted per Methods, Efferent Coupling e Afferent Coupling* 79
Figura 15 Valores do Código Fonte dos Dois Softwares para *Average Cyclomatic Complexity* e *Average Lines of Code per Method* 79
Figura 16 Código Fonte de Manipulação de Notícias do Software Legado 81
Figura 17 Código Fonte de Manipulação de Notícias do Software Desenvolvido Utilizando a Técnica TDD 82
Figura 18 Grafo de Dependências entre Componentes do Software Legado 83

Figura 19 Grafo de Dependências entre Componentes do Software Desenvolvido Utilizando a Técnica TDD 84

LISTA DE QUADROS E TABELAS

Quadro 1 Tipos de manutenção 35

Tabela 1 Resultados das métricas utilizadas.. 78

SUMÁRIO

1 INTRODUÇÃO ... 13
1.1 Motivação ... 14
1.2 Objetivo ... 16
1.3 Metodologia de desenvolvimento ... 16
1.4 Tipo de pesquisa ... 16
1.4.1 Procedimentos metodológicos ... 18
1.5 Estrutura do trabalho ... 19
2 TRABALHOS RELACIONADOS ... 20
3 QUALIDADE DE SOFTWARE ... 25
3.1 Considerações iniciais ... 25
3.2 Visão geral ... 25
3.3 Norma ISO/IEC 9126 ... 27
3.4 Manutenção de software ... 33
3.4.1 Importância ... 37
3.4.2 Manutenibilidade ... 38
3.5 Considerações finais ... 39
4 TEST DRIVEN DEVELOPMENT - TDD ... 41
4.1 Considerações iniciais ... 41
4.2 Caracterização ... 41
4.3 Etapas ... 46
4.4 Objetivos ... 49
4.5 Benefícios ... 51
4.6 Dificuldades ... 52
4.7 Considerações finais ... 53
5 MÉTRICAS ... 55
5.1 Considerações iniciais ... 55

5.2 Tipos de métricas 55
5.3 Importância 57
5.4 Métricas utilizadas 58
5.5 Considerações finais 61
6 AVALIAÇÃO DA MANUTENIBILIDADE DE SOFTWARE ORIENTADO A OBJETO DESENVOLVIDO COM A TÉCNICA TDD .. 62
6.1 Considerações iniciais 62
6.2 O software simulador de bolsa de valores 62
6.3 Tecnologias utilizadas 64
6.4 Características do código fonte 66
6.4.1 Utilização da técnica TDD 67
6.4.2 Utilização de *Mock Objects* 68
6.4.3 Utilização de padrões de projeto 69
6.4.4 Benefícios alcançados 72
6.5 Resultados obtidos 75
6.5.1 Cálculo das métricas 75
6.5.2 Resultado das métricas 77
6.5.3 Análise dos resultados 79
6.5.4 Comparação entre os códigos fonte 85
6.6 Dificuldades com a aplicação da técnica TDD 88
6.7 Considerações finais 89
7 CONSIDERAÇÕES FINAIS 91
7.1 Conclusões 91
7.2 Contribuições 93
7.3 Trabalhos futuros 94
REFERÊNCIAS 96

1 INTRODUÇÃO

O desenvolvimento de software cresceu nas últimas décadas, em que as empresas passaram a demandar software para facilitar suas atividades diárias (Pressman, 2010). Nos dias atuais, não é um diferencial informatizar uma empresa, é essencial. Com tamanha demanda, muitas empresas desenvolvedoras de software surgiram e existem várias pessoas envolvidas nesse desenvolvimento. Com a mudança nas necessidades empresariais, o software também mudou. Os softwares ganharam novas funções e sua complexidade aumentou, o que acarretou acréscimo no custo para quem os desenvolve e para quem os financia (Astels, 2003).

Por outro lado, a manutenção de software é essencial, pois ele deve estar consonante com o meio onde está implantado para continuar útil. Uma das formas de realizar manutenção de software é aumentar sua funcionalidade (manutenção evolutiva) (Pressman, 2010; Pfleeger; Atlee, 2009; Sommerville, 2010). Adicionar funções em softwares legados significa realizar manutenção em códigos fonte, que podem ter sido escritos por outras pessoas as quais não estão mais envolvidas com o projeto desse software.

Com a complexidade alta do código fonte desses softwares, manutenções tornam-se árduas e demandam tempo para entendê-los, o que pode frustrar as expectativas dos clientes (Riaz *et al.*, 2009). Neste cenário de dificuldade de entendimento, surge a preocupação com a legibilidade do código fonte de softwares legados para facilitar sua manutenção (Janzen; Saiedian, 2008). Porém, essa preocupação pode não ser intrínseca de todas as pessoas envolvidas no processo de desenvolvimento ou manutenção de software, pois é algo difícil de atingir. Um código fonte legível significa ter um planejamento antes de começar a escrevê-lo, exige perícia do programador e demanda tempo de aprendizado e esforço (Astels, 2003).

Para facilitar esta tarefa, com o objetivo de guiar desenvolvedores a construir códigos fonte mais legíveis, existe uma técnica denominada *Test Driven Development* (TDD), criada por Kent Beck (Beck, 2002). O seu objetivo é construir código fonte simples, legível e que funcione conforme a necessidade de uso do software, descrita pelo cliente, e funcionar como parte da documentação do *design* (Canfora; Visaggio, 2009).

Um código fonte de fácil manutenção significa que as modificações podem ser feitas com certo grau de facilidade pelo desenvolvedor, em um fluxo de alterações mais contínuo (Beck, 2002). A técnica TDD permite aos desenvolvedores realizarem estas alterações com agilidade e segurança, pois pode-se atingir código fonte mais legível e assegurar o comportamento do software usando testes automatizados (Torchiano; Silliti, 2009).

1.1 Motivação

A utilização da técnica TDD para a construção de softwares é relativamente novo no mercado, tendo início entre 2002 e 2003, e ainda não está consolidada. Como uma das principais preocupações dessa técnica é obter um código fonte simples de manter (Astels, 2003), a manutenção desse código fonte tende a ser menos árdua. Outra preocupação é desenvolver o software com a funcionalidade que o cliente pediu (Beck 2002; Astels 2003), tendo em vista isso ser um desafio que a indústria de software enfrenta, pois muitas vezes os conceitos se perdem no meio dos processos (Beck, 1999; Pressman, 2010). O desenvolvimento de software utilizando a técnica TDD sugere começar mais simplesmente, escrevendo os casos de testes conforme a descrição do software, e, posteriormente, refatorar o código fonte para atingir a funcionalidade desejada na sua completude. O teste é feito antes do desenvolvimento da lógica real a ser implementada no software.

Em seguida, é feita a remoção dos erros, um por vez, atingindo a funcionalidade desejada ao fim de cada interação. Assim, assegura-se que o código fonte escrito funcione da maneira que se espera, obtendo um código fonte confiável. A técnica TDD guia o desenvolvedor para fazer o certo, ou seja, para o software atender aos requisitos do cliente. Além disso, ela ajuda a reduzir o custo final do projeto e, principalmente, o tempo de desenvolvimento do software, pois levam o desenvolvedor a construir código fonte com menor probabilidade de ocorrência de falhas (Pressman, 2010; Astels, 2003).

Os desenvolvedores tomam pequenas decisões após cada teste, o que aumenta a taxa de atualização e de adição das funções do software. Pequenos testes devem ser escritos e, após obterem sucesso, o desenvolvedor avança ao próximo teste, o que caracteriza a implementação total da função e contribuição na redução dos erros.

A técnica possui algumas regras que ajudam a manter o *design* e o funcionamento do código fonte. Uma regra importante é (Beck, 2002),

> "If you can't write a test for what you are about to code, then you shouldn't even be think about coding"

que relata a importância do teste para a técnica TDD. Em uma tradução livre dessa regra para o português, pode-se interpretá-la como: se não é possível escrever um teste para o caso a se programar, então a programação não deve ocorrer.

Como a técnica TDD antecipa as falhas, evitam-se problemas no resultado final do software, por causa da realização antecipada dos testes, o que permite reduzir custos e tempo. Estas reduções são as principais motivações para o aprendizado, o uso e o estudo desta técnica, a qual proporciona um software mais robusto ao cliente.

1.2 Objetivo

O objetivo do trabalho é aplicar a técnica conhecida como TDD (*Test-Driven Development*) em uma aplicação real – Software Simulador de Bolsa de Valores - para alcançar um código-fonte com maior manutenibilidade.

Espera-se obter código fonte legível, de fácil compreensão e que sua manutenção seja facilmente realizada por qualquer membro da equipe ou por pessoas que não tiveram envolvimento no seu projeto de desenvolvimento.

1.3 Metodologia de desenvolvimento

A metodologia de pesquisa é um conjunto de métodos, técnicas e procedimentos cuja finalidade é viabilizar a execução da pesquisa que tem como resultado um novo produto, processo ou conhecimento (Jung, 2009).

1.4 Tipo de pesquisa

Uma pesquisa pode ser classificada em (Figura 1) (Jung, 2009):

a) Quanto a Natureza: i) Pesquisa Básica (gerar conhecimento sem finalidades de aplicação) e ii) Pesquisa Aplicada (gerar conhecimento sem finalidades de aplicação);
b) Quanto aos Objetivos: i) Exploratória (descobrir/inovar); ii) descritiva (como?); e iii) explicativa (por que?);
c) Quanto as Abordagens: i) Quantitativa e ii) Qualitativa;
d) Quanto aos Procedimentos: i) Survey; ii) Pesquisa-Ação; iii) Estudo de Caso Único ou Múltiplos; iv) Operacional; e v) Experimental.

Além disso, os métodos para a coleta dos dados podem ser por meio de (i) observação do participante, (ii) grupos focados, (iii) entrevistas, (iv) questionário, (v) experimentação e (vi) observação.

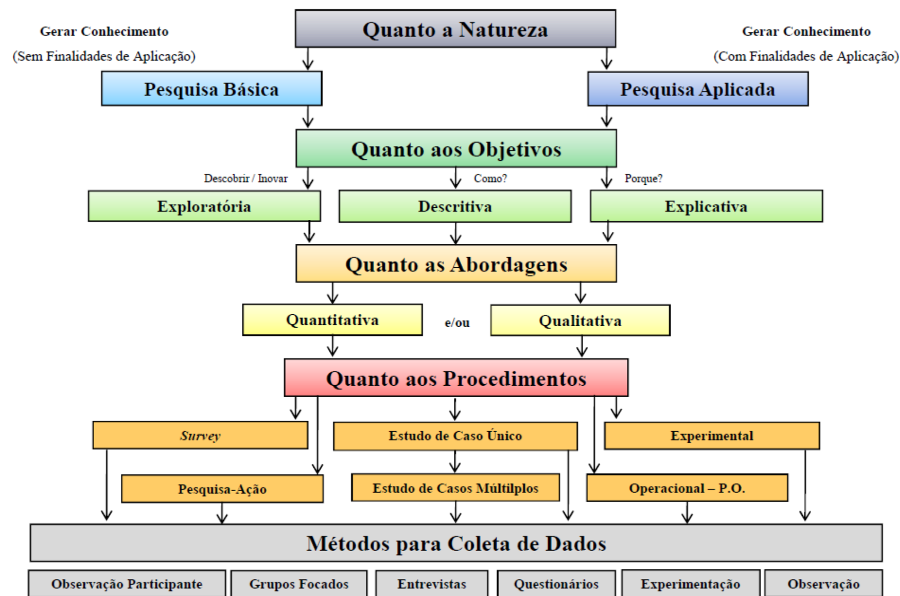


Figura 1 Tipos de pesquisa científica
Fonte: Jung, 2009.

Quanto a natureza, este trabalho classifica-se como **pesquisa aplicada**, pois é realizada sobre um problema com finalidades de aplicação. Quanto aos seus objetivos, este trabalho pode ser caracterizado como **pesquisa exploratória**, pois avalia a melhoria da característica de qualidade de manutenibilidade na reconstrução de um software legado orientado a objetos utilizando a técnica TDD para descobrir possíveis impactos, negativos ou positivos, nessa reconstrução. Quanto à sua abordagem, este trabalho é uma **pesquisa quantitativa**, pois visa à medição e à análise de algumas características do código fonte de dois softwares com mesma funcionalidade,

porém desenvolvidos de maneira diferente. Quanto aos procedimentos, este trabalho pode ser caracterizado como um **estudo de caso**, pois é apresentado um estudo da aplicação de uma técnica e da análise de seus impactos (fenômenos) no desenvolvimento de um software. A coleta de dados deste trabalho se dá pela **observação** do código fonte dos dois softwares em análise.

1.4.1 Procedimentos metodológicos

Inicialmente, houve a realização de uma pesquisa bibliográfica a fim de levantar conceitos e exemplos da aplicação da técnica TDD no desenvolvimento de um software. Em seguida, alguns exemplos de uso da técnica em softwares com poucas linhas de código (*toy example*) foram analisados e implementados para familiarizar-se com a técnica, pois a sua prática difere da maneira tradicional de desenvolvimento de software. O estudo e a implementação destes exemplos colaboraram para adaptação com a maneira que a técnica TDD guia o desenvolvedor a programar um software.

Em seguida, um software foi desenvolvido com base em casos de uso que descrevem sua funcionalidade. Estes casos de uso são originários da versão anterior do código fonte de um software desenvolvido sem a utilização da técnica TDD (software legado). Após a implementação desses casos de uso, métricas de complexidade de software foram estudadas para identificar aquelas mais apropriadas para o contexto do trabalho. As métricas escolhidas foram aplicadas no código fonte de ambos os softwares para obter medidas referentes à características internas do código fonte para verificar a característica de qualidade de manutenibilidade, por exemplo acoplamento entre os componentes e complexidade desses componentes. As medidas foram obtidas utilizando o

software Google CodePro Analytix[1], um *plug-in* para a IDE (*Integrated Development Environment*) Eclipse.

Em seguida, foram feitas observações a respeito das diferenças e semelhanças entre estes códigos fonte e quais os benefícios que a técnica TDD pode proporcionar ao código fonte final. Cabe ressaltar que, durante o desenvolvimento deste trabalho, houve reuniões semanais com o orientador e coorientador para sanar dúvidas e nortear os próximos passos.

1.5 Estrutura do trabalho

Este trabalho está estruturado da seguinte maneira.

Alguns trabalhos relacionados são citados no Capítulo 2.

Conceitos sobre qualidade de software e aspectos de qualidade que o trabalho analisa são sucintamente comentados no Capítulo 3.

Referências sobre a técnica TDD, mostrando principais conceitos utilizados no trabalho à respeito desta técnica, são apresentadas no Capítulo 4.

Caracterização de métricas de complexidade e identificação das métricas mais adequadas a serem utilizadas no contexto deste trabalho são resumidamente explicadas no Capítulo 5.

Resultados obtidos com implementação de um software e a aplicação das métricas identificadas, comparando o código fonte das duas versões do software para avaliar a utilização da técnica TDD são discutidos no Capítulo 6.

Considerações finais do trabalho realizado relacionando conclusões, contribuições e sugestões de trabalhos futuros são apresentadas no Capítulo 7.

---

[1] http://code.google.com/intl/pt-BR/javadevtools/codepro/doc/index.html

2 TRABALHOS RELACIONADOS

O objetivo deste trabalho é agregar valor no cenário da qualidade de software sob o ponto de vista da característica de qualidade de manutenibilidade utilizando a técnica TDD no desenvolvimento de um software real. Para isso, alguns fatores no código fonte do software desenvolvido são observados, tais como, facilidade de modificação, de legibilidade e de compreensão e baixo acoplamento e alta coesão entre os componentes.

Profissionais e pesquisadores da academia e da indústria envolvidos no desenvolvimento de softwares têm investido esforços para encontrar maneiras de diminuir custos financeiros e de tempo empregados na realização de atividade de manutenção de softwares. Nesse cenário, alguns trabalhos podem ser encontrados na literatura e são brevemente relatados a seguir.

Em um destes trabalhos (Janzen; Saiedian, 2008), foi apresentado um estudo com alunos de graduação e pós-graduação organizados em duas equipes. Este estudo consistiu no desenvolvimento de um software utilizando a tecnologia de orientação a objetos, sendo que uma equipe o desenvolveu utilizando abordagem tradicional e aplicou os testes posteriormente (*test-after*) e a outra equipe utilizou a técnica TDD (*test-first*). O objetivo foi concluir se a técnica TDD melhora o *design* do código fonte, realizando comparações dos resultados obtidos em ambas as abordagens. As equipes desenvolveram o mesmo software sem terem interação. Após a finalização do experimento, medições foram feitas utilizando métricas para analisar o código fonte, tais como complexidade ciclomática, número de ramificações de dependências, peso de um método quanto a sua complexidade, quantidade de linhas por classe e por método e métodos por classe. De posse dessas medidas, foram avaliadas a influência entre as classes e o quanto o tamanho do código fonte e seu acoplamento impactam na qualidade interna. A conclusão foi que a técnica TDD

permite a construção de código fonte mais coesos e o seu reaproveitamento foi maior. Além disso, houve tendência de escrever métodos mais simples e com menos responsabilidades, o que propiciou simplicidade do código fonte e, consequentemente, facilidade de compreensão. O software foi construído com classes menores em relação às classes das equipes que utilizaram o *test-after* e com maior capacidade de escrever testes para suas funções, ou seja, o software possuía maior possibilidade de verificação de suas funções o que garantia seu bom funcionamento.

Outro trabalho (Canfora; Visaggio, 2008) obteve resultados semelhantes ao anterior. Foram utilizadas duas equipes independentes para avaliar a aplicação da técnica TDD, uma equipe realizou testes antes (*test-first*) e outra equipe realizou testes depois (*test-after*). O conjunto de métricas utilizado foi similar ao trabalho anterior que consistiu, basicamente, na avaliação da complexidade do código fonte. A conclusão desse trabalho foi que a técnica TDD é uma alternativa à forma tradicional de desenvolvimento de software e favorece a construção de código fonte manutenível. Além disso, este trabalho afirma que o desempenho na detecção de falhas no código fonte utilizando a técnica TDD é superior à técnica tradicional.

Um terceiro trabalho (Vu *et al.*, 2009) identificou outros valores relacionados à técnica TDD. Aplicando métodos semelhantes aos trabalhos descritos anteriormente, foi mostrado que, em algumas equipes, a técnica TDD foi responsável por uma queda na produtividade, considerando a entrega das funções ao comparar as equipes que utilizaram a técnica tradicional de testes automatizados. O contraponto é a técnica TDD depender da capacidade e da preferência da equipe que a utiliza. Neste estudo, as equipes não desenvolviam em apenas um estilo, ocorrendo revezamento. Foi possível observar que algumas equipes utilizaram a técnica de maneira não adequada quanto ao seu conceito, aplicando testes maiores do que é sugerido, ou seja, os testes devem ser o mais

simples possível (Beck, 2002). Dessa forma, a técnica TDD sofre influência da equipe e que o rendimento das pessoas envolvidas pode variar conforme seu perfil, o que pode dificultar a manutenção da simplicidade de código fonte em relação ao obtido por meio de desenvolvimento tradicional.

Além das características de qualidade de manutenibilidade, existem melhorias relacionadas à comunicação entre os membros da equipe, como mostra um quarto trabalho analisado (Crispin, 2006). Esse trabalho reforça os resultados apresentados no primeiro trabalho relatado nesse capítulo, argumentando os benefícios de iniciar por testes o desenvolvimento de softwares. O desenvolvedor é forçado a pensar mais sobre questões sobre funcionalidade desenvolvida antes de codificá-la, tornando o código fonte mais objetivo. Esta abordagem (*test-first*) gera uma descrição de cada função passo a passo; assim, não há preocupação se algo foi esquecido ou se foi entendido de maneira errada no momento de desenvolver. Os testes vão dizer ao desenvolvedor se está implementado de maneira correta. Com isso, observou-se maior capacidade dos desenvolvedores em realizar tarefas de manutenção com menos lentidão. Além disso, a comunicação entre estes desenvolvedores tornou-se eficiente. Enquanto eles escreviam uma parte do código fonte e, mais tarde, iam para a refatoração, havia troca maior de informação, pois um desenvolvedor procurava saber o significado de um teste escrito por outra pessoa. Essa troca de informação era constante dentro da equipe, o que proporcionava maior agilidade na hora de entender os componentes a serem modificados ou complementados e melhorava o grau de manutenibilidade do software. O trabalho mostrou que, além de maior facilidade na leitura do código fonte, a técnica TDD influenciou outros níveis do desenvolvimento, por exemplo a troca de informação na equipe, fato que contribuiu para maior produtividade no momento de realizar modificações e atualizações em um sistema.

Além destes fatores relacionados à legibilidade do código fonte, pode-se pensar na capacidade em encontrar problemas no software (Martin, 2007). Além dos benefícios apresentados anteriormente, este trabalho argumenta melhor capacidade de verificar uma falha em um software. Observou-se que os desenvolvedores que utilizavam a técnica TDD realizavam menos *debug* no código fonte, pois conseguiam identificar o local das falhas com rapidez; este fato foi caracterizado no trabalho como *minimal debugging*. Outro ponto levantado foi associar o seguimento das regras da técnica TDD por parte desenvolvedor implica que ele encontra e corrige os erros durante o desenvolvimento, pois sabe exatamente a localidade dos erros, o que diminui a execução de *debugs*. Contudo, não é uma tarefa fácil, deve-se seguir o ciclo da técnica TDD para que estes benefícios sejam alcançados. Portanto, ao seguir este ciclo, há forte tendência do desenvolvedor entregar funções com pequena quantidade de *bugs*, o que proporciona agilidade na hora de realizar manutenções.

Por fim, um sexto trabalho analisa a simplicidade do código fonte gerado quando se utilizou a técnica TDD (Vodde; Koskela, 2007). Quando equipes na empresa Nokia Networks aplicaram essa técnica no desenvolvimento de softwares, foi identificado um código fonte simples, onde era fácil entender o que havia sido feito, pois os métodos possuíam responsabilidades bem definidas em que as tarefas estavam claras a quem lesse seu código fonte. Além disso, foram abordados aspectos relacionados à produtividade da equipe e observado que a principal dificuldade foi iniciar o primeiro teste de cada função. No trabalho, foi relatado que, enquanto se realizava a refatoração, a equipe voltava para repensar o código fonte e conseguia identificar redundâncias e aspectos que deixavam o código fonte de difícil leitura. Durante o desenvolvimento, as melhorias aconteciam no momento dessa refatoração, onde removiam pequenas informações no código fonte (chamadas *stuff*). Conseguiram retirar do software

muitos trechos que contribuíam para que ele se tornasse mais rebuscado. O *design* foi pensado e reformulado passo a passo, removendo estes *stuffs*, e alcançaram um código fonte objetivo e simples.

Há uma tendência nos trabalhos relacionados à garantia de manutenibilidade e/ou ao bom *design* de código fonte em comparar a utilização ou não da técnica TDD. Existe a semelhança nos resultados a serem buscados, onde o estudo está relacionado ao quanto foi possível melhorar o código fonte e se o código fonte se tornou simples e limpo.

3 QUALIDADE DE SOFTWARE

3.1 Considerações iniciais

Este capítulo apresenta conceitos de qualidade de software, tomando como base a norma ISO/IEC 9126, que define características para avaliar um software quanto a sua qualidade. Após a apresentação da norma, o foco do capítulo é voltado para a característica de manutenibilidade, pertencente a esta norma e, por fim, são introduzidos os aspectos do software que devem ser avaliados neste trabalho.

Conceitos de qualidade relacionados a um software são brevemente comentados na Seção 3.2. A norma ISO/IEC 9126 é sucintamente apresentada na Seção 3.3. Manutenção de software e sua importância e característica de manutenibilidade de um software são apresentadas na Seção 3.4.

3.2 Visão geral

Qualidade de software consiste na existência de características no software que podem ser associadas a seus requisitos (Petrasch, 1999). Desenvolver software com qualidade é um desafio a ser enfrentado pelos engenheiros de software, pois se trata de questões abstratas, sendo difícil encontrar um parâmetro para medir os pontos de qualidade (Pressman, 2010). Porém, a qualidade é percebida pelos envolvidos em um processo de software (desenvolvimento ou manutenção). Uma pessoa atribui qualidade a determinado produto conforme sua experiência ao utilizá-lo, definindo se possui ou não qualidade.

O software não é algo material, como um objeto em que se possa avaliar com facilidade as características que determinem qualidade. Trata-se de algo

abstrato, tal qual é o conceito de qualidade (Petrasch, 1999). Contudo, na Engenharia de Software, a qualidade não está atrelada somente a fatos de experiência do usuário (Pressman, 2010). A qualidade do produto depende fortemente da qualidade de seu processo de desenvolvimento (Duarte; Falbo, 2000). Estes processos de desenvolvimento podem ser avaliados segundo requisitos de qualidade, onde alguns fatores são medidos para se tentar chegar a um valor, referente a esta medição da qualidade, mais compreensível (Rincon, 2009). Como os processos são padronizados, pode-se avaliá-los, tornando mais fácil a observação de fatores que determinem qualidade, conforme o foco do que é desenvolvido.

Além destes aspectos, a qualidade de software está atrelada à qualidade do código fonte desenvolvido (Beck, 2004). Um código fonte bem feito aumenta a capacidade relacionada da sua manutenção, bem como outros fatores, tais como segurança e confiabilidade. A qualidade está associada a fatores internos no software, invisíveis ao usuário, e a fatores de experiência que o usuário tem ao utilizar determinado software. As impressões que ele extrair do contato com o software agrega valor ao software, sendo um conceito bem pessoal (Beck, 2004; Pressman, 2010).

No ponto de vista interno dos aspectos de desenvolvimento, a medida de qualidade não se trata de opiniões pessoais, mas de processos avaliados e de características técnicas que devem obedecer às definições do projeto, ou seja, o software deve funcionar conforme os requisitos (Beck, 2004). Estudos acerca da qualidade de software têm sido segmentado de acordo com duas perspectivas: qualidade de processo e qualidade de produto (Dal Moro; Falbo, 2008). Existem algumas normas e modelos de qualidade, tais como CMMI-SW (CMMI, 2011), ISO/IEC 12207 (ISO/IEC 12207:2008, 2008) e MPS.BR (MR-MPS:2009, 2009) que focam aspectos da qualidade de processo de software, enquanto a norma

ISO/IEC 9126 tem o seu foco em aspectos relacionados à qualidade de software (produto gerado) (Dal Moro; Falbo, 2008).

Neste trabalho, são considerados aspectos referentes ao software, ou seja, as características do seu código fonte; portanto, a abordagem é qualidade de produto. Para isso, a norma ISO/IEC 9126 (ISO/IEC 9126-1:2001, 2001) é utilizada, pois apresenta definição das características de qualidade de produto de software. Para haver uma separação nas propriedades a serem analisadas de modo a avaliar qualidade de um software, essa norma organiza a qualidade de produto de software em seis características, as quais são brevemente descritas a seguir.

3.3 Norma ISO/IEC 9126

A norma ISO/IEC 9126 é um padrão definido pela *International Organization of Standardization/International Electrotechnical Commission* que define características relacionadas à qualidade de software (Zeiss *et al.*, 2007). Ela define um modelo de qualidade de software e tem a ideia de ser utilizada como um *framework* para a obtenção dessa qualidade e a organiza em interna e externa. As seis características são (i) funcionalidade, (ii) confiabilidade, (iii) usabilidade, (iv) eficiência, (v) manutenibilidade e (vi) portabilidade. Cada característica possui um conjunto de subcaracterísticas conforme o seu escopo (Figura 3-1). A definição destas características e de suas subcaracterísticas é (ISO/IEC 9126-1:2001, 2001):

a) Confiabilidade: é o conjunto de atributos que evidencia a capacidade do software em manter um bom nível de desempenho sob determinadas condições e em um determinado período de tempo. Subcaracterísticas:

- Maturidade: consiste em atributos do software que evidenciam a frequência de falhas por defeitos no produto de software;
- Recuperabilidade: consiste em atributos do software que evidenciam a sua capacidade de retornar ao funcionamento normal, por meio da recuperação dos dados após as falhas e o tempo e o esforço necessários para isso;
- Tolerância a Falhas: consiste em atributos do software que evidenciam a sua capacidade em continuar o nível de desempenho especificado nos casos de falhas no software ou de violação nas interfaces especificadas.

b) Eficiência: é o conjunto de atributos que evidencia o relacionamento entre o nível de desempenho do software e a quantidade de recursos que utiliza sob determinadas condições. Subcaracterísticas:

- Comportamento em Relação a Recursos: consiste em atributos do software que evidenciam a quantidade de recursos utilizados e a duração de seu uso na execução de suas funções;
- Comportamento em Relação ao Tempo: consiste em atributos do software que evidenciam o seu tempo de processamento e de resposta e a velocidade na execução de suas funções.

c) Funcionalidade: é o conjunto de atributos que evidencia a existência de um conjunto de funções e suas propriedades que satisfazem às necessidades implícitas e explícitas. Subcaracterísticas:

- Adequação: consiste em atributos do software que evidenciam a presença de um conjunto de funções e a sua adequação para as tarefas especificadas;
- Acurácia: consiste em atributos do software que evidenciam a geração de resultados ou efeitos corretos ou conforme acordados;
- Interoperabilidade: consiste em atributos do software que evidenciam a sua capacidade de interagir com outros softwares;

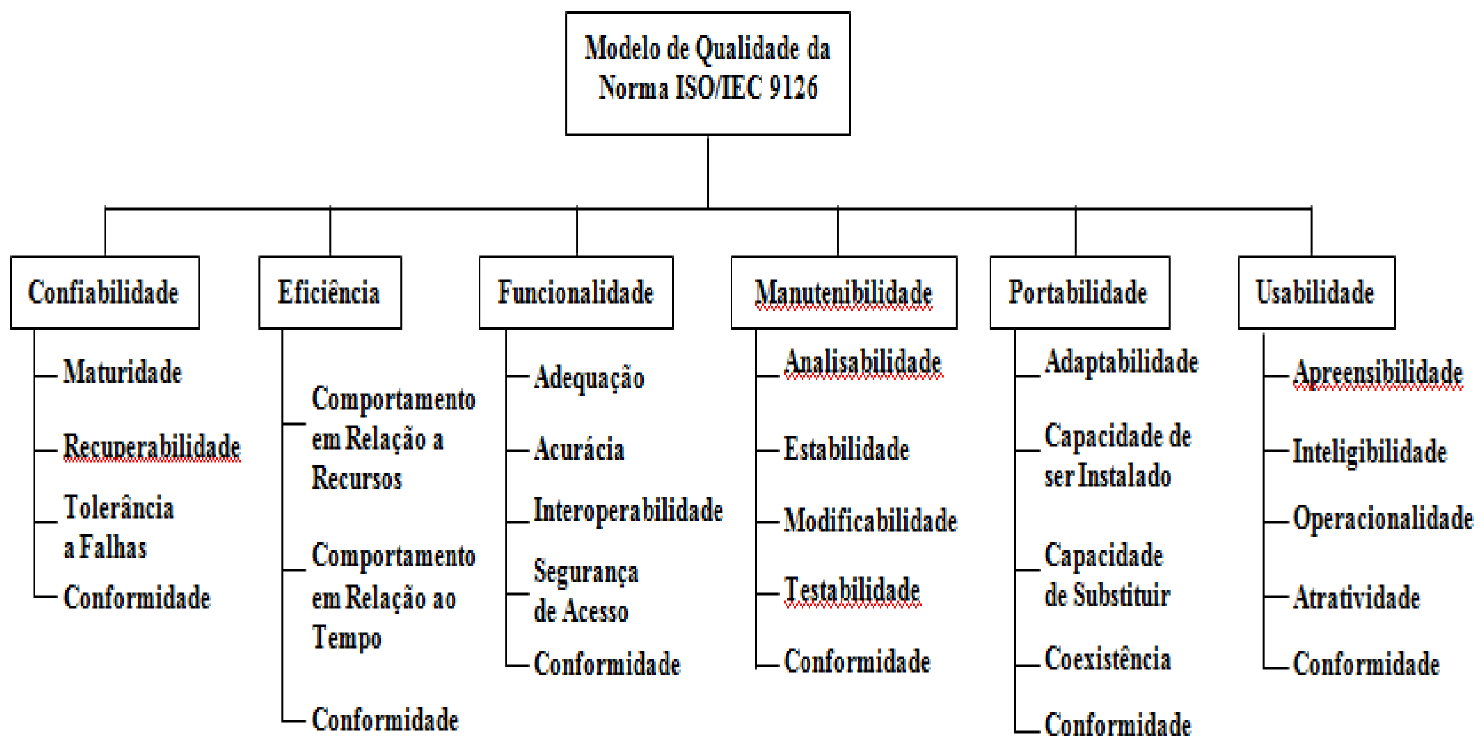


Figura 2 Características de qualidade da norma ISO/IEC 9126

d) Segurança de Acesso: consiste em atributos do software que evidenciam a sua capacidade de evitar acessos não autorizados, acidentais ou deliberados aos dados e ao software.

- Manutenibilidade: é o conjunto de atributos que evidencia o esforço necessário para realizar modificações específicas no software. Subcaracterísticas:

a) Analisabilidade: consiste em atributos do software que evidenciam o esforço necessário para diagnosticar deficiências ou causas de falhas, ou para identificar partes a serem modificadas;
b) Estabilidade: consiste em atributos do software que evidenciam o risco de efeitos inesperados, ocasionados por modificações;
c) Modificabilidade: consiste em atributos do software que evidenciam o esforço necessário para modificá-lo, remover seus defeitos ou adaptá-lo a mudanças ambientais;
d) Testabilidade: consiste em atributos do software que evidenciam o esforço necessário para validar o produto de software modificado.

- Portabilidade: é o conjunto de atributos que evidencia a capacidade do software de ser transferido de um ambiente para outro. Subcaracterísticas:

a) Adaptabilidade: consiste em atributos do software que evidenciam a sua capacidade de ser adaptado a ambientes diferentes do especificado, sem a necessidade de aplicação de outras ações ou meios além daqueles fornecidos para essa finalidade pelo software considerado;

b) Capacidade para ser Instalado: consiste em atributos do software que evidenciam o esforço necessário para a sua instalação em um determinado ambiente;
c) Capacidade para Substituir: consiste em atributos do software que evidenciam o esforço necessário para substituir outro software, no ambiente estabelecido para este segundo;
d) Coexistência: consiste em atributos do software que evidenciam a facilidade com que este software coexiste com outros softwares instalados no mesmo ambiente consiste em atributos do software que evidenciam a facilidade com que este software coexiste com outros softwares instalados no mesmo ambiente.

- Usabilidade: é o conjunto de atributos que evidencia o esforço necessário para poder utilizar o software, bem como o julgamento individual desse uso, por um conjunto implícito ou explícito de usuários. Subcaracterísticas:

a) Apreensibilidade: consiste em atributos do software que evidenciam o esforço do usuário para aprender a sua aplicação;
b) Inteligibilidade: consiste em atributos do software que evidenciam o esforço do usuário para reconhecer o conceito lógico e a sua aplicabilidade;
c) Operacionalidade: consiste em atributos do software que evidenciam o esforço do usuário para a sua operação e o controle desta operação;
d) Atratividade: consiste em atributos do software que evidenciam a capacidade do software em atrair um potencial usuário para o sistema, o que pode incluir desde a adequação das informações

prestadas para o usuário até técnicas visuais utilizadas na sua interface gráfica.

Todas as características possuem a subcaracterística Conformidade que consiste em atributos do software que evidenciam o quanto o software obedece aos requisitos de legislação e todo o tipo de padronização ou normalização aplicável ao contexto.

3.4 Manutenção de software

O processo de desenvolvimento de software está completo quando ele foi entregue e é utilizado pelos usuários. Qualquer mudança, após o software estar operacional, é considerada manutenção. Como mudanças são inevitáveis ao longo da sua vida, mecanismos devem ser previstos para avaliar, controlar e fazer essas modificações.

Dessa forma, há a necessidade de direcionar esforços para a manutenção de software, pois muitos dos existentes são de 10 a 15 anos atrás. Ainda que eles tenham sido desenvolvidos utilizando as melhores práticas de *design* e implementação na época, os critérios para um bom software eram um pouco diferentes dos atuais, pois a preocupação era de tamanho do código fonte e espaço de armazenamento. Vários destes softwares têm sido migrados para novas plataformas, adaptados às mudanças de equipamentos e de tecnologia e incrementados para atender às novas necessidades dos usuários. Nestas ocasiões, pode-se deparar com estruturas e lógicas pobremente projetadas, com código fonte nem sempre organizado e com documentação deficiente dos softwares que devem continuar funcionando (Osborne; Chikofsky, 1990 *apud* Pressman, 2010).

A preocupação pela manutenção não é dos dias de hoje, pois, na década de 80, foi proposto um conjunto de leis de evolução de softwares denominado de Leis de Lehman (Lehman; Belady, 1985). Dentre elas, a primeira é a mais relevante para este trabalho e diz que a manutenção de software é inevitável. A eliminação de falhas é apenas uma parte das atividades de manutenção, pois, para o software ser útil, a mudança dos seus requisitos deve ser incorporada ao software. A razão para isso é que o ambiente, onde ele está inserido, pode mudar com o tempo de maneira que os requisitos dos usuários também mudem.

Os softwares, em geral, são desenvolvidos sem a preocupação com o seu tempo de vida útil, não sendo projetados objetivando facilitar a sua manutenção (Pressman, 2010). Dessa forma, observa-se que o problema não está no software, mas na forma como ele tem sido desenvolvido.

Os custos para aumentar a funcionalidade de um software, após ter sido posto em operação, são maiores do que fornecer funcionalidade similar quando ele ainda está em desenvolvimento. Há algumas razões para isso (Sommerville, 2010):

a) a equipe de manutenção é relativamente inexperiente e não familiarizada com o domínio da aplicação. A manutenção é considerada como uma tarefa que necessita de pouca habilidade em comparação com o desenvolvimento e é destinada ao pessoal mais novo;
b) os softwares em manutenção podem ter sido desenvolvidos há anos sem considerar as técnicas de engenharia de software. Eles podem estar desestruturados e eficientes ao invés de ter características de facilitar o entendimento;

c) mudanças realizadas em softwares podem introduzir novos erros por causa da complexidade do software, acarretando dificuldade na avaliação dos efeitos das mudanças;
d) por causa das constantes mudanças no software, a sua estrutura tende a degradar. Isso faz com que o software seja mais difícil de ser entendido e dificulta futuras mudanças, gerando um software menos coeso;
e) a correspondência entre o código fonte e a documentação do software pode se perder durante o processo de manutenção. Sendo assim, a documentação torna-se não confiável para o seu entendimento.

A manutenção de software pode ser: corretiva, adaptativa, perfectiva e preventiva (Quadro 1) (Pfleeger (2009), Pressman (2010) e Sommerville (2010)).

Quadro 1 Tipos de manutenção

Tipo de Manutenção	**Motivo da Manutenção**
Adaptativa	É realizada quando o produto de software precisa ser adaptado às novas tecnologias (hardware e software) implantadas no ambiente operacional.
Corretiva	É realizada quando são corrigidos erros não identificados durante o Fluxo de Trabalho de Teste.
Evolutiva	É realizada quando o produto de software deve englobar novos requisitos ou melhorias decorrentes da evolução na tecnologia de implementação empregada.
Preventiva	É realizada quando o produto de software é alterado para aumentar sua manutenibilidade e/ou confiabilidade. Este tipo de manutenção é relativamente raro em ambientes de desenvolvimento. Modificações realizadas neste tipo de manutenção não afetam o comportamento funcional do produto de software.

Alguns fatores que afetam a manutenção podem ser caracterizados como em técnicos e não-técnicos (Sommerville, 2010). Os fatores técnicos são:

a) Independência de Módulos. Deve ser possível modificar um componente de um software sem afetar outros componentes.
b) Linguagem de Programação. O software escrito em uma linguagem de programação alto-nível é, geralmente, mais fácil de ser entendido e, assim, ser mantido do que aquele escrito em linguagem de programação baixo-nível;
c) Estilo de Programação. A maneira em que um software é escrito contribui para seu entendimento e facilita a sua modificação;
d) Teste e Validação de Software. Geralmente, quanto mais tempo e esforços forem gastos na validação do *design* e no teste, menor é a quantidade de erros;
e) Qualidade da Documentação do Software. Se um software é apoiado por uma documentação clara, completa e concisa, a tarefa de entendê-lo é facilitada;
f) Técnicas de Gerência de Configuração. Há um custo significativo para manter a rastreabilidade e garantir a consistência da documentação de um software.

Os fatores não-técnicos são:

a) Domínio da Aplicação. Se estiver bem definido e entendido, a probabilidade da completude dos requisitos é razoável;
b) Estabilidade da Equipe. Custos de manutenção são reduzidos se os engenheiros de software que desenvolvem o software são responsáveis pela sua manutenção;
c) Tempo de Vida do Software. Quanto mais antigo é o software, mais ele pode ter sofrido ações de manutenção, portanto mais cara pode

se tornar a sua manutenção devido à degradação de sua estrutura após diversas modificações;

d) Dependência do Software com o Meio Externo. Quando o meio externo muda, o software deve mudar para acompanhar as alterações no meio externo;

e) Estabilidade do Hardware. Se o software foi desenvolvido para uma configuração específica do hardware, quando há alteração da configuração, o software deve ser alterado para acompanhar a nova configuração.

3.4.1 Importância

A deterioração do código fonte do software é um dos fatores que requer a sua manutenção para que sejam melhoradas características de desempenho e de utilização de recursos, por exemplo (Janzen; Saiedian, 2008; Pressman, 2010; Fowler, 1999). O código fonte pode estar correto do ponto de vista dos seus usuários. Mesmo assim, ele pode estar deteriorado, ou seja, a alteração no código fonte é mais difícil do que naturalmente deveria ser, sendo excessivamente complexa (Eick *et al.*, 2001). Essa deterioração pode ocorre por causa da evolução do meio em que o software está inserido ou por várias manutenções realizadas nele (Ping, 2010).

Essa atualização afeta o *design* interno (código fonte) do software que sofre impactos a cada modificação efetuada, mesmo que a independência de seus componentes/módulos seja razoável (alta coesão e baixo acoplamento) (Izurieta; Bieman, 2007). Dessa forma, se mal planejadas, as alterações podem prejudicar essa independência o que torna a manutenção de software mais árdua, necessitando de refatoração dos seus componentes/módulos (Eick *et al.*, 2001; Pressman, 2010).

A manutenção de software consome aproximadamente 75% do esforço do processo de desenvolvimento de um software (Beck, 2002; Pressman, 2010). Trata-se de uma fase do ciclo de vida do software em que ele é melhorado e corrigido, e tempo e esforços são dedicados. Este esforço excessivo ocorre por diversos motivos, mas, em especial, por causa da equipe de desenvolvimento muitas vezes adotar a cultura de corrigir os erros e não de preveni-los (Janert; Rikfin, 2004). Além disso, o cliente muitas vezes pede para que algumas funções sejam alteradas ou complementadas, ocasionando modificações e acarretando esforço prolongado nesta etapa (Beck, 2004; Pressman, 2010).

3.4.2 Manutenibilidade

A qualidade de um software começa por seu *design* interior, o código fonte (Beck, 2002). Assim, é importante predizer a manutenibilidade de software para gerenciar os custos com mais eficiência (Riaz *et al*., 2009). A manutenibilidade é a capacidade que o software tem de ser modificado, para efetuar correções de erros, falhas, melhorar o desempenho do software ou o *design* do código fonte tentando fazê-lo mais simples e legível (IEEE Std 610.12, 1990).

Portanto, a manutenibilidade não é um processo do desenvolvimento do software, mas uma característica de qualidade utilizada para avaliar um software (Riaz; Mendes; Tempero, 2009). É um fator que pode impactar significativamente os custos do software, sendo importante garanti-la e conseguir prever seus níveis (Riaz *et al.*, 2009; Chen; Huang, 2009).

A manutenibilidade é de grande interesse na disciplina Engenharia de Software, sendo este interesse originado da ligação entre a manutenibilidade e a influência das práticas de *design* na manutenção de softwares. Idealmente, os softwares devem ser projetados tendo em vista a manutenibilidade e a sua

documentação deve permitir a correção e o reúso de código fonte durante o seu aprimoramento, bem como eliminar a necessidade da engenharia reversa[2].

A manutenibilidade de software tende a ser negligenciada por causa de diversos fatores, entre eles:

a) anseios das organizações responsáveis pelo seu desenvolvimento em atender rapidamente o mercado;
b) cultura de que a manutenção não é uma atividade nobre;
c) caracterização da atividade de manutenção como um problema a ser tratado posteriormente ao desenvolvimento de software.

Não existe uma maneira de medir manutenibilidade diretamente, sendo necessário utilizar medidas indiretas (Pressman, 2010), por exemplo, realizar análises sobre o tempo total para ponderar sobre as mudanças requisitadas, elaborar uma modificação apropriada, além de implementar as mudanças e testá-las. A partir desta análise, pode-se inferir se o software possui bom nível de manutenibilidade ao averiguar a complexidade do seu código fonte (Pressman, 2010).

3.5 Considerações finais

Conceito sobre qualidade de software foi abordado e especificado que, dentro do escopo do trabalho, as análises têm foco no software, portanto a abordagem de qualidade utilizada foi referente a qualidade do produto,

---

[2] É prática comum durante a manutenção de software, pois a documentação produzida durante o desenvolvimento de um software costuma estar desatualizada com o software existente. Ela consiste na reconstrução da documentação a partir do código-fonte do produto de software existente.

excluindo as que analisam fatores referentes ao processo e ao projeto do software.

Meio a esta visão de qualidade de produto está inserida a norma ISO/IEC 9126, que apresenta a definição de seis características e de suas subcaracterísticas para analisar a qualidade do software como produto. Desta maneira, o presente trabalho considera utilizar uma das características de qualidade presente nesta norma: manutenibilidade, cujo objetivo é aspectos relacionados à capacidade de manutenção de um software. A importância desta característica foi discutida embasada em alguns fatores como a deterioração do software.

Por fim, a característica de qualidade de manutenibilidade e o que se leva em consideração para caracterizar se um software possui bom grau de qualidade sob o ponto de vista desta característica foram abordados. A partir destes atributos considerados nessa característica, pode-se ter conhecimento sobre quais aspectos o código fonte pode ser analisado para afirmar algo sobre sua manutenibilidade. Este tipo de análise é feita para verificar se esse grau no código fonte do software desenvolvido aumenta ao utilizar a técnica TDD.

4 TEST DRIVEN DEVELOPMENT - TDD

4.1 Considerações iniciais

A técnica *Test Driven Development* (TDD) tem pouco mais de 10 anos e foi criada dentro do escopo do *eXtreme Programming* (XP) (Wasmus; Gross, 2007). O seu objetivo é proporcionar ao desenvolvedor uma maneira simples de desenvolver software e que resulte em código fonte com legibilidade e fácil de manter. Apesar de uma década de existência, esta técnica ainda não tem seu uso frequente; as empresas estão adotando a técnica TDD apenas recentemente (Wasmus; Gross, 2007). Esta baixa utilização deve-se ao fato de ser parte de uma metodologia ágil que também ainda não se solidificou no mercado, existindo apenas pequenos nichos, sendo as metodologias tradicionais as mais utilizadas no desenvolvimento de softwares (Janzen; Saiedian, 2005).

Este capítulo é destinado à apresentação da técnica TDD, técnica aplicada no desenvolvimento da aplicação base deste trabalho. Uma breve descrição da técnica e seus objetivos e princípios são apresentados na Seção 4.2, cujo foco é no desenvolvimento e na motivação do desenvolvedor, que obtém resultados instantâneos sobre a confiabilidade de seu código fonte. As iterações da técnica para obter um ciclo de desenvolvimento-teste e resultar em código fonte funcional ao final são apresentadas na Seção 4.3. O objetivo da técnica TDD é discutido na Seção 4.4. Benefícios e dificuldades são discutidos na Seção 4.5 e na Seção 4.6, respectivamente.

4.2 Caracterização

O princípio da técnica TDD é desenvolver software onde o desenvolvedor se orienta por testes, ou seja, ele escreve testes correspondentes a

uma função especificada nos casos de uso e guia-se por este teste para programá-la (Beck, 2002). A técnica também é conhecida como *test-first*, em oposição ao *test-after*, quando os testes são escritos após a implementação do software (Janzen; Saiedian, 2005).

Diferentemente das metodologias tradicionais, o desenvolvedor tem informações a respeito do comportamento de uma função implementada instantaneamente, logo que desenvolve, sendo um processo mais dinâmico do que o tradicional, quando o teste é feito após a escrita do código fonte (Astels, 2003). A técnica TDD se torna interessante na medida em que existe o desejo de reduzir custos. Tradicionalmente, existe a política de correção de testes ao fim do desenvolvimento, porém esta é uma tarefa em que se tem custo extra de tempo e de recursos (Beck, 2002). Com esta técnica, o intuito é evitar que falhas aconteçam o que pode ser obtido com a criação de testes antes da implementação da função.

Na técnica TDD, o desenvolvedor escreve o código fonte do software à medida em que avança no desenvolvimento dos testes e, no decorrer do processo, tomam-se decisões referentes à arquitetura interna do software, modificando o código fonte com frequência para tornar seu *design* legível, simples e funcional (Janzen; Saiedian, 2005; Shull *et al*., 2010). Essa técnica pode ser vista como um conjunto de iterações para completar uma tarefa (Beck, 2002), ou seja, quando um teste é escrito, o desenvolvedor realiza modificações no código fonte até que este teste tenha obtido sucesso, caracterizando iterações até alcançar o comportamento desejado. Após estas iterações e o sucesso do teste, a tarefa é completada e o desenvolvedor inicia a implementação da próxima função.

Quando o desenvolvedor escreve o software em pequenas iterações (Astels, 2003)

> ele quer trabalhar com pequenos incrementos. Às vezes incrementos tão pequenos que parecerão ridículos. Porém, quando compreender o significado disto, então estará pronto para dominar a técnica TDD.

Um código fonte dividido em pequenas interações de desenvolvimento diminui o acoplamento e aumenta a coesão do código fonte, fator determinante para a melhora característica de qualidade de manutenibilidade e o aumento do reúso de funções (Beck, 2002). Este tipo de desenvolvimento pode parecer ao desenvolvedor que não lhe trará benefícios, mas a médio e longo prazos as vantagens começam a ser notadas. Isso acontece, pois, no momento de escrever novas funções, pode-se reaproveitar o que foi escrito e o desenvolvimento flui mais rapidamente, além do programador saber se o que ele escreveu afeta o software negativamente (Wirfs-Brock, 2007).

Os testes elaborados na aplicação da técnica TDD são unitários e escritos pelo próprio desenvolvedor, que os formula conforme as necessidades do desenvolvimento (Wasmus; Gross, 2007; Janzen; Saiedian, 2005). Cada função nova é atrelada a um teste, não sendo desenvolvida sem antes ter um teste feito especificamente para ela (Crispin, 2006; Martin, 2007).

A fim de escrever teste apenas para uma função, as dependências que uma classe possuir não são utilizadas com implementações reais, mas com simulações delas por um objeto falso chamado *Mock Objects* (Astels, 2003). Desta forma, o desenvolvedor garante que apenas uma classe tem seu código fonte coberto pelo teste e estes *Mock Objects* apenas respondem o que o desenvolvedor configura, sem execução/processamento associado.

Um exemplo da utilização de *Mock Object* é apresentado na Figura 3, onde o código fonte de uma classe é encarregado por testar as buscas de notícias exclusivamente da fonte Folha de São Paulo. Contudo, este buscador utiliza uma classe chamada RestTemplate a qual não é interessante testar nesse momento.

Assim, o desenvolvedor utiliza um *Mock Object* para simular o comportamento de um método da classe RestTemplate, que no caso é o método getForObject(...).

Na linha 45 do código fonte apresentado na Figura 3, está representado o local em que o comportamento deste método é configurado e seu retorno é o argumento passado como parâmetro para o método thenReturn(...), que pertence ao *framework* que configura os *Mock Objects*. Na linha 48, o buscador é instanciado e a sua dependência é recebida como parâmetro no construtor, porém apenas com o comportamento simulado.

```
28  public class TestSearchNewsFromFolha {
29
30      @Mock
31      private RestTemplate restTemplate;
32
33      private NewsSearcher searcher;
34
35      private List<News> list;
36      @Mock
37      private DateConvertor convertor;
38      @Before
39      public void init() throws RestClientException, IOException{
40          MockitoAnnotations.initMocks(this);
41          Map<String, String> params = new HashMap<String, String>();
42          params.put("q", "bovespa");
43          params.put("sr", String.valueOf(1));
44          String link = "http://search.folha.com.br/search?q={q}&sr={sr}";
45          Mockito
46              .when(restTemplate.getForObject(link, String.class, params))
47              .thenReturn(stringNewsFromFrolha());
48          searcher = new NewsSearcherFromFolha(restTemplate, convertor);
49          list = searcher.getNews();
50      }
51
```

Figura 3 Exemplo de Uso de *Mock Objects*

Para garantir a qualidade do código fonte, a técnica TDD propõe que os testes sejam feitos de maneira exaustiva e em pequenos passos (Beck, 2002). À princípio, não deve haver grande preocupação com a codificação do software propriamente dito, mas com o teste. Desta maneira, após o teste ter sido escrito, ele vai falhar por ainda não existir a função que ele testa e, a partir deste ponto, o

desenvolvedor começa a codificá-la (Astels, 2003). Para isso, o desenvolvedor formula o teste de acordo com a sua descrição especificada nos casos de uso. A ideia é dividir para conquistar: deve-se dividir a funcionalidade do software em vários pontos (funções) e "atacar" uma por vez (Beck, 2002).

A técnica TDD sugere um tipo de desenvolvimento onde se procura começar a codificação pela parte mais crucial para a função escrita naquele instante. Para isso, realizam-se pequenos passos na escrita do código fonte e pequenas modificações no decorrer da programação, até conseguir o resultado esperado para esta função, e seguir adiante com a implementação de uma nova função (Janzen; Saiedian, 2008).

Primeiramente, o desenvolvedor deve ter em mente a construção de um código fonte funcional, que responda corretamente o que o teste exige. Após ter escrito o teste, o desenvolvedor executa-o e observa que ele falha; em seguida, o desenvolvedor escreve um código fonte que responda corretamente ao teste, assegurando que o teste obtenha sucesso. Posteriormente, o desenvolvedor refatora o código fonte de maneira que seu *design* se torne simples, sem alterar o sucesso da aplicação dos testes. Essa refatoração não deve afetar o comportamento pré-estabelecido, mas que seja uma implementação real da lógica de negócio do software.

Estes passos caracterizam etapas e formam um ciclo, o qual sugere ao desenvolvedor segui-los para alcançar a simplicidade do código fonte e, ao mesmo tempo, tentar garantir o funcionamento, eliminando redundâncias, etc. Assim, o código fonte resultante tende a ser funcional, limpo e de estar testado (Astels, 2003).

4.3 Etapas

A técnica TDD é organizada em etapas iterativas e conectadas entre si (Beck, 2002). O desenvolvedor deve respeitá-las para obter *feedback* e seguir adiante com a implementação do software. Após escrever o primeiro teste para uma determinada função, o desenvolvedor segue passos de maneira que sejam constantemente testadas, baseados nos casos de teste. O ciclo deve ser repetido quando o resultado da aplicação de um novo teste obtém insucesso e/ou nova função deve ser implementada (Shull *et al*., 2010).

Nas referências sobre a técnica TDD, as etapas são descritas, mas não há especificação de uma nomenclatura específica. Essas etapas são divididas em atividades e organizadas em um ciclo que o desenvolvedor deve seguir para formalmente caracterizar a utilização da técnica (Beck, 2002). Porém, as etapas podem ser denominadas por (Beck, 2002):

a) Escrever o Teste. A escrita de testes é a principal característica da técnica TDD. O desenvolvedor deve iniciar a construção das funções do software pelo teste, o qual é baseado nos casos de uso analisados e cada função a ser desenvolvida deve estar relacionada ao teste escrito. O primeiro teste escrito não vai obter sucesso, pois não existe uma linha de código fonte referente a função a ser implementada. Desta maneira, observando os erros encontrados, o desenvolvedor foca a sua correção com o objetivo de fazer com que os testes tenham sucesso. Um dos efeitos de trabalhar construindo testes primeiro é o desenvolvedor possuir um conjunto de testes programados o que proporciona certa segurança para que as mudanças efetuadas não causem efeitos laterais (Astels, 2003). Se caso algum efeito colateral ocorrer, os testes escritos acusam a falha

de maneira que o desenvolvedor deve refatorar o código fonte e repetir a iteração do desenvolvimento neste ponto, partindo para o próximo e garantindo o funcionamento das novas funções. Como benefício maior, o desenvolvedor foca no que está fazendo e não em como está fazendo (Astels, 2003);

b) Executar os Testes e Esperar por Falhas. Os testes escritos pelo desenvolvedor são automatizados (Janzen; Saiedian, 2005; Shull *et al*., 2010). Para isso, os testes são executados dentro da IDE utilizada no desenvolvimento. O teste executado acusará na tela da IDE qual requisito não atendido, seja lógico ou semântico (Keuffel, 2004). O objetivo desta etapa é o desenvolvedor executar os testes e esperar a ocorrência de falhas. A partir deste momento, ele tem uma direção a seguir para escrever o código fonte funcional do software. Essa fase é determinante para o desenvolvedor ter uma visão com foco na resolução dos problemas que o código fonte possui;

c) Realizar Mudanças no Código Fonte. Esta é a etapa em que o desenvolvedor realiza as correções em seu código fonte para que o respectivo teste tenha sucesso, sem se preocupar se a lógica está ou não correta, focando no resultado que a função deve retornar. Cada modificação deve ser simples e, de maneira incremental, a função é implementada até chegar ao que realmente deseja (Wirfs-Brock, 2007). Após realizar estas modificações e obter sucesso no teste, há garantia de resultados quando novas mudanças forem feitas posteriormente. Desta forma, o desenvolvedor pode realizar mudanças amparadas por um teste relacionado o qual especifica o que esta função modificada deve retornar e pelos testes de outras funções, que acusam se algum efeito colateral modifica o

comportamento esperado que o software deve ter (Beck, 2002, Janzen; Saiedian, 2008);

d) Executar Novamente os Testes e Esperar Obter Sucesso. Os testes são novamente executados e espera-se que as modificações realizadas tenham sucesso. Caso contrário, volta-se à etapa anterior para realizar novas mudanças e executar os testes até obter sucesso;

e) Refatorar o Código Fonte. Após garantir que o software responde corretamente aos testes, o desenvolvedor passa a se preocupar com uma implementação mais concreta do código fonte (Astels, 2002; Shull *et al*., 2010). Neste ponto, a função testada é incrementada com a escrita de algoritmos para a resolução do problema. Ao chegar nesta etapa, existe a segurança de que os testes exigem comportamento específico da função, pois obtiveram sucesso na etapa anterior. Assim, é o momento de remover possíveis duplicações de código fonte, adequar melhor o código fonte ao problema, deixar o software mais dinâmico, entre outras decisões que o desenvolvedor deve tomar para tornar o código fonte legível, limpo, robusto e simples. Após refatorado, o código fonte é novamente testado. O processo se repete até o desenvolvedor concluir que a função esteja adequadamente implementada e terminada.

Três momentos base da técnica TDD são apresentados na Figura 4, onde existem as situações: Vermelho (cor vermelha), Verde (cor verde) e Refatoração (cor roxa) (Brief, 2001). O círculo Vermelho simboliza o momento em que não se pode seguir adiante com o desenvolvimento. Quando os testes falham, o desenvolvedor tem um sinal vermelho significando que não deve implementar novas funções até estes testes terem sucesso. Respeitando esta premissa, o

desenvolvedor garante que o software corresponde à especificação do teste. O círculo Verde representa o momento em que o desenvolvedor pode dar continuidade a implementação. Neste momento, o desenvolvedor tem a garantia do funcionamento explícito no resultados dos testes, pois eles obtiveram sucesso. O círculo Refatoração representa a refatoração do código fonte que o desenvolvedor deve realizar. As redundâncias devem ser eliminadas e as melhorias na arquitetura do código fonte devem ser feitas. Se após a refatoração os testes acusarem erro, o desenvolvedor retorna para o estado representado pelo círculo Vermelho. Caso contrário, o desenvolvedor pode prosseguir com o desenvolvimento.

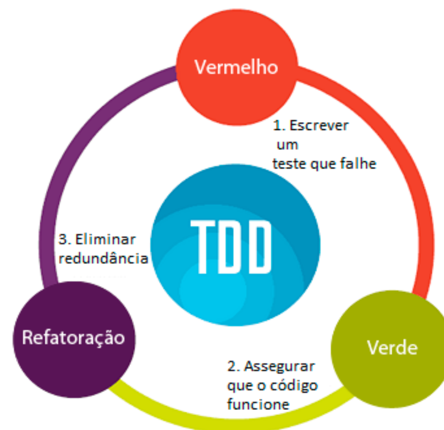


Figura 4 Ciclo da Técnica TDD (Adaptado de Brief, 2001)

4.4 Objetivos

A técnica TDD não tem o objetivo principal de testar o software, sendo uma ferramenta que vai além desse conceito. O objetivo é (Beck, 2002)

"clean code that works"

isto é, um código fonte limpo e funcional, pois executa as tarefas corretamente e tem razoável legibilidade, sem misturar responsabilidades em um componente/módulo (Martin, 2007).

Muito se argumenta sobre a produtividade de desenvolvimento de softwares utilizando a técnica TDD, por exemplo, empresas afirmam não terem tempo para testes ou que perdem tempo desenvolvendo pequenos passos (Asltels, 2003). Contudo, os resultados são obtidos a médio e longo prazos, pois essa técnica garante qualidade de código fonte, fator que pode influenciar diretamente na sua capacidade de manutenção. Como o desenvolvimento é dirigido por testes, a possibilidade do desenvolvedor escrever código fonte para funções não presente nos casos de uso do software é menor, pois não constarão nos testes definidos (Janert; Rikfin, 2004; Vodde; Koskela, 2007).

A técnica TDD propicia baixa ocorrência de erros e garante que o código fonte não fique defasado, sendo possível aplicar-lhe mudanças e melhoramentos a qualquer momento, inclusive por pessoas novas na equipe de desenvolvimento que podem ter melhor entendimento com a leitura dos testes (Beck, 2002; Martin, 2007; Crispin, 2006). Novos membros nas equipes são beneficiados pela utilização dessa técnica, pois os testes elaborados são de fácil compreensão e correspondem exatamente a funcionalidade desenvolvida.

Os softwares desenvolvidos não estão livres de sofrerem manutenções (Pressman, 2010; Astels, 2003), pois, eventualmente, modificações são solicitadas pelo cliente a fim de, por exemplo, atender novas necessidades (novo relatório, nova função, ...). Assim, quando o código fonte de software são de má qualidade e de difícil entendimento, recursos de tempo e dinheiro são demandados mais do que o necessário, sendo a utilização da técnica TDD uma alternativa para evitar essas perdas (Beck, 2002). Ela garante *design* e comunicação razoáveis entre os desenvolvedores e, quando modificações são realizadas e os testes são executados, qualquer erro de comportamento é

assinalado e mudanças podem ser feitas com rapidez e segurança (Crispin, 2006).

4.5 Benefícios

Utilizar a técnica TDD não é trivial, pois é difícil de aprendê-la (Crispin, 2006). Existe a necessidade de análise eficiente por parte dos desenvolvedores sobre a funcionalidade do software, pois os testes são baseados nos seus casos de uso (Beck, 2002). A equipe deve conhecer o software a ser desenvolvido e estar ciente por qual função iniciará o desenvolvimento.

Contudo, mesmo com dificuldades, a técnica TDD possui benefícios. O *feedback* instantâneo proporciona segurança ao desenvolvedor, pois ele sabe quais falhas ocorrem e se elas ainda ocorrem, depois de efetuadas as correções. Isto gera outro benefício: atualização do código fonte. Se o programador desenvolve respeitando o ciclo da técnica, ele realiza testes constantemente, obtém *feedback* instantâneo sobre o funcionamento do código fonte escrito e realiza as mudanças necessárias. Desta forma, a técnica TDD assegura ganho de produção a que se propõe, pois os *bugs* são reduzidos e evita-se a necessidade de corrigi-los (Astels, 2003).

Sendo assim, o tempo perdido para os desenvolvedores encontrarem a origem do problema é reduzido (Johnson *et al.*, 2007), pois ele sabe onde está o erro (o teste mostra o local), propiciando que modificações são feitas apenas no local correto e com mais rapidez (Crispin, 2006).

Além do ganho de produtividade em relação à identificação dos erros indesejados, a técnica TDD diminui o acoplamento entre as funções. O desenvolvedor escreve seu código fonte pensando exclusivamente no teste que ele mesmo concebeu; assim, as funções são feitas de maneira independente,

sendo o código fonte desenvolvido para interfaces, o que diminui o acoplamento (Beck, 2002; Freeman *et al.*, 2004).

Escrever o código fonte focado em interfaces, e não na implementação dele, resultam em uma abstração mais próxima do mundo real (Freeman *et al.*, 2004). Durante a programação, o desenvolvedor se torna capaz implementar soluções existentes com maior facilidade, como padrões de projetos. Devido à característica de desenvolvimento passo a passo, a utilização de padrões de projetos como o *Strategy, State* e *Factory* se torna mais frequente no código.

Por isso o acoplamento entre os componentes é reduzido. Esse é o resultado da técnica TDD aplicado de maneira correta e consciente (Beck, 2002). O código fonte desacoplado, e muito mais coeso, será essencial para a garantia de manutenibilidade.

4.6 Dificuldades

Dentre as vantagens apresentadas, algumas dificuldades são comumente encontradas ao utilizar a técnica TDD. Uma delas, talvez a principal, é a identificação de quais testes começar a escrever (Asltels, 2003; Crispin, 2006). O desenvolvedor é quem modela os testes para o seu próprio código fonte; para isso, ele deve ter plena consciência das funções a serem implementadas.

Além dos testes terem que ser bem elaborados (Astels, 2003; Beck, 2002; Crispin, 2006), o desenvolvedor não deve seguir processos cegamente, mas deve encontrar a sua melhor maneira de desenvolver software utilizando a técnica TDD (Beck, 2004). Por esta razão, muitos desenvolvedores não conseguem se encaixar no modelo de desenvolvimento de software que propõem a técnica TDD.

O desenvolvedor deve ser experiente para que consiga observar os casos de testes que tem que ser desenvolvidos e implementar os testes corretamente

(Crispin, 2006; Janzen; Saiedian, 2008; Vodde; Koskela, 2007). Caso contrário, a utilização da técnica TDD não vai proporcionar vantagem sobre as metodologias tradicionais, pois os testes errados não vão traduzir o que o cliente realmente deseja do software.

4.7 Considerações finais

A utilização da técnica TDD guia o desenvolvedor a escrever um código fonte mais próximo de uma API (*Application Programming Interface*), que significa tê-lo com baixo acoplamento e alta coesão, onde cada função é bem definida. Isto pode acontecer quando o desenvolvedor pensa e desenvolve uma função por vez.

Pelos passos da técnica TDD serem pequenos, o desenvolvedor escreve de maneira incremental. Assim, as classes e os métodos tornam-se menores e mais objetivos, com responsabilidades bem definidas.

Desta forma, possíveis erros, redundâncias e duplicações no código fonte passam a ser facilmente identificadas e corrigidas. Além do código ser mais simples, existe a cobertura do teste que indica com maior precisão onde um comportamento indesejado está ocorrendo. Estes são grandes benefícios que o desenvolvedor tem ao utilizar a técnica TDD, propiciando maior qualidade ao software gerado.

Contudo, essa técnica não é a solução pronta para todos os problemas. Ela depende do conhecimento do desenvolvedor e se ele consegue assimilar as tarefas a serem cumprida, além de exigir disciplina. Porém, no momento em que a equipe se conhece e sabe de suas limitações, adotar a técnica TDD não é algo tão complexo. Os benefícios de um desenvolvimento com a utilização dessa técnica são inúmeros e deve ser este o foco motivacional para uma equipe utilizá-la.

Seguindo a técnica TDD de forma correta, o software tende a ter um código fonte enxuto e de fácil manutenção e compreensíveis para a equipe e para pessoas externas ao projeto. A técnica TDD, portanto, é de grande benefício para o desenvolvimento do software, conseguindo proporcionar melhorias desde características do código fonte à satisfação do cliente, que tem suas solicitações de mudanças mais rapidamente atendidas.

5 MÉTRICAS

5.1 Considerações iniciais

Conceitos de métricas de software, a sua relevância no processo de construção do software e as métricas utilizadas para analisar o código fonte deste trabalho são discutidos neste capítulo. Para isso, cada métrica é descrita e são explicados os motivos de sua escolha e a sua importância no presente trabalho.

Breve descrição de métricas de software é apresentada na Seção 5.2. A importância das métricas no desenvolvimento do software é discutida na Seção 5.3. As métricas utilizadas são descritas na Seção 5.4.

5.2 Tipos de métricas

Algumas métricas de software são utilizadas para medir aspectos relacionados ao desenvolvimento do software, podendo considerar fatores técnicos, da lógica de programação ou relacionados ao desempenho do software (Akingbehin, 2009). Outras métricas podem ter abordagem em níveis mais altos no processo de desenvolvimento de software, como análise de recursos utilizados, produtividade e pessoas (Soliman *et al*., 2010).

As métricas de software podem ser organizadas em três categorias (Akingbehin, 2009; Kan, 2003):

a) Métricas de Processo. Essa métricas descrevem e melhoram o desenvolvimento do software e a manutenção das atividades, tais como tempo de desenvolvimento, eficácia na remoção de erros e tempo para correção de erros;

b) Métricas de Projeto. Essas métricas descrevem as características de um projeto de software, tais como recursos humanos, custo, tempo e produtividade;
c) Métricas de Produto. Essas métricas descrevem características do produto, tais como tamanho, complexidade, performance e qualidade, sendo utilizadas para melhorar aspectos técnicos do software. Apenas essas métricas são abordadas neste trabalho.

As métricas de produto se dividem nas classes (Sommerville, 2010) (i) métricas dinâmicas, coletadas por medições feitas de um software em execução, e (ii) métricas estáticas, coletadas por medições feitas das representações do software. A segunda classe é apresentada com mais detalhes por serem relativas ao código fonte de um software e utilizadas para a realização desse trabalho.

As métricas estáticas tem uma relação indireta com os atributos de qualidade, sendo que alguns estudos foram realizados a fim de conseguir estabelecer uma relação entre elas e fatores como complexidade, facilidade de compreensão e facilidade de manutenção do software (Sommerville, 2010). Algumas métricas utilizadas para a medição da qualidade de software são (Sommerville, 2010):

a) *Fan-in*/*Fan-out* (conhecidas por acoplamento aferente e acoplamento eferente, respectivamente);
b) Tamanho do código fonte;
c) Complexidade ciclomática;
d) Extensões dos identificadores;
e) Profundidade de declarações de condicionais aninhadas;
f) Índice fog.

Estas métricas são utilizadas para estabelecer medições referentes à complexidade do software. Além delas, existem métricas específicas para softwares orientados a objetos, tais como (Sommerville, 2010):

a) Profundidade da árvore de herança;
b) Método de *fan-in/fan-out*;
c) Métodos ponderados por classe;
d) Número de operações sobrepostas.

Estas métricas são utilizadas para medir a complexidade do software, analisando as dependências, a complexidade dos códigos fonte, entre outros fatores. Portanto, elas podem ser utilizadas para avaliar a manutenibilidade do software, sendo algumas utilizadas neste trabalho.

5.3 Importância

As métricas de software são utilizadas para ajudar os gerentes de projetos e os desenvolvedores a tomar decisões a respeito de aspectos do código fonte em desenvolvimento, quando apenas a observação do software não é suficiente (Gyimothy, 2009). A utilização das métricas é essencial para fazer uma análise da complexidade do código fonte, inclusive identificando trechos onde podem ser melhorados (Meananeatra; Rongviriyapanish, 2011).

Contudo, a relevância de uma métrica específica varia de projeto para projeto, dependendo das metas da equipe de gerenciamento de qualidade e do tipo de software em desenvolvimento (Sommerville, 2010). Elas podem ser úteis em algumas situações, mas em outros momentos podem não ser.

A partir da aplicação das métricas no código fonte, são fornecidos valores que indicam a sua complexidade e que podem comprometer o

entendimento do que foi escrito quando as medidas estão em graus elevados (Kan, 2003). Por exemplo, o grau de acoplamento entre componentes/módulo do software é um exemplo de fator que contribui para a dificuldade de manutenção e que pode ser calculado com o uso de métricas (Prasad; Nagar, 2009). De posse dos valores resultantes da aplicação das métricas, é possível identificar diferenças e pontos onde o código fonte pode ser modificado para obter melhorias no que diz respeito à legibilidade e alcançar um código fonte cuja manutenção seja simples e eficiente (Gyimóthy *et al*., 2005; Shrivastava; Jain, 2011).

5.4 Métricas utilizadas

As métricas de software podem ser voltadas para as análises de código fonte orientado a aspectos, de código fonte orientado a objeto, de código fonte com estrutura procedural ou de código fonte que combine as três (Honglei *et al.*, 2009). Para a realização deste trabalho, cinco métricas foram selecionadas para realizar a avaliação do código fonte a ser desenvolvido utilizando a técnica TDD:

Complexidade Ciclomática. Essa métrica é utilizada para medir a quantidade de ciclos em um código fonte causados pela ocorrência de estruturas condicional e de repetição e chamadas de métodos e de funções, ou seja, ela mede a número de caminhos que a execução do software possui (Bhat; Nagappan, 2006). Ela pode ser utilizada para obter uma medida que mostre a complexidade do código fonte, informando se a sua legibilidade está ruim. Além disso, o seu valor permite analisar o grau de acoplamento entre classes (Janzen; Saiedian, 2008). Um exemplo de um grafo de complexidade ciclomática de um código fonte hipotético é apresentado na Figura 5, o qual representa a variação dos caminhos. Esta métrica é a diferença entre a quantidade de nós do grafo e a

quantidade de arestas do grafo mais 2, ou seja, quantidade de arestas - quantidade de nós + 2 (Pfleeger; Atlee, 2009). Cada aresta é um caminho e cada condicional ou uma execução é um nó. As bifurcações podem ocorrer por causa de uma condicional ou da chamada de métodos internos ou externos a uma classe. Quanto maior a quantidade de bifurcações, maior é o valor da complexidade ciclomática (Kan, 2003);

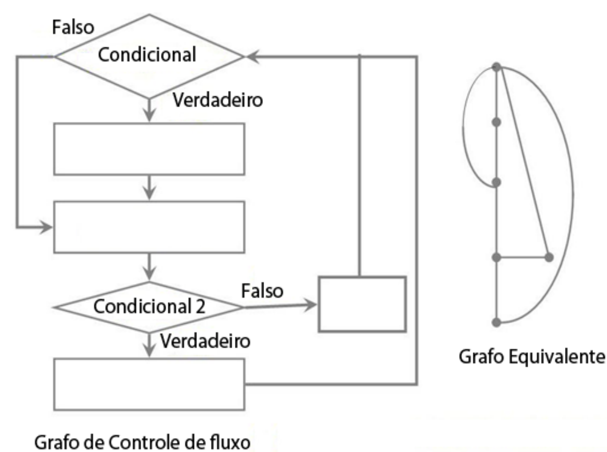


Figura 5 Grafo de Complexidade Ciclomática
Fonte: Adaptado de Pfleeger; Atlee, 2009.

Peso dos Métodos por Classe. Essa métrica mede a soma das complexidades ciclomáticas dos métodos de uma classe (Janzen; Saiedian, 2008; Honglei *et al*., 2009). O resultado desta soma representa os níveis de responsabilidade do objeto em relação ao software, verificando a quantidade de tarefas que seus métodos executam. Quanto maior o valor obtido desta métrica, mais complexo é o código fonte do software e maior é a responsabilidade da classe (componente/módulo) (Honglei *et al*., 2009), ou seja, os esforços de trabalho para manutenção nesta classe será maior;

Média do Número de Linhas por Método. Essa métrica verifica a quantidade de linhas que as classes possuem em cada um de seus métodos (Janzen; Saiedian, 2008). Sua análise indica se o código fonte presentes nos métodos são longos, o que aumenta a probabilidade deles possuírem maior complexidade e responsabilidades do que deveriam (Beck, 2002; Fowler, 1999). Se os métodos das classes são mais curtos então a sua complexidade é menor, logo realizar manutenções nestes métodos pode ser um trabalho mais simples, pois o código fonte tende a ser mais legível (Janzen; Saiedian, 2008);

Acoplamento Aferente (*Fan-in*). Essa métrica representa a quantidade de classes externas de um pacote (componente/módulo) que dependem de classes pertencentes a este mesmo pacote (componente/módulo) (Meirelles, 2008). Quanto maior este valor, maior pode ser considerado o reúso de código fonte, pois significa que uma classe é muito utilizada em outros componentes (Janzen; Saiedian, 2008);

Acoplamento Eferente (*Fan-out*). Essa métrica representa a quantidade de classes em um pacote (componente/módulo) que utilizam outras classes fora deste pacote (componente/módulo) (Meirelles, 2008). O seu valor pode caracterizar o grau de acoplamento entre os pacotes (componentes/módulos). Quanto menor o valor desta métrica, menos uma classe afeta outra, o que quer dizer que mais independente são as classes. Quanto maior for este valor, menor será o reúso de código fonte (Meirelles, 2008; Janzen; Saiedian, 2008). Quando é menor o reúso, a equipe de manutenção tem dificuldades, pois tem que entender muitos métodos para efetuar a modificação em apenas um método, tornando a manutenção uma tarefa árdua e demorada (Kastenberg, 2004).

5.5 Considerações finais

As métricas selecionadas para a realização deste trabalho foram escolhidas para obter um valor que fosse possível fazer observações sobre a complexidade do código fonte de um software. O uso dessas métricas se dá sobre o código fonte existente, mas considera apenas fatores de mais baixo nível, como a relação entre os objetos.

Contudo, alguns fatores importantes, como a nomenclatura dos métodos e dos objetos condizentes com a situação, não são abordados. Neste trabalho, a preocupação se deu na forma de tentar manter bom *design* de código fonte e menor acoplamento, onde modificações pudessem ser feitas de maneira modularizada.

O uso das métricas é crucial na análise, para que os resultados deste trabalho sejam alcançados. Desta forma, com base nas métricas que analisam o software, conclusões são feitas, realizando uma comparação entre o código fonte das duas versões do software avaliadas: o software legado (desenvolvido sem a utilização da técnica TDD) e o software desenvolvido utilizando a técnica TDD.

6 AVALIAÇÃO DA MANUTENIBILIDADE DE SOFTWARE ORIENTADO A OBJETO DESENVOLVIDO COM A TÉCNICA TDD

6.1 Considerações iniciais

Os resultados obtidos com as métricas selecionadas no Software Simulador de Bolsa de Valores, desenvolvido utilizando a técnica TDD, são discutidos neste capítulo. As métricas foram aplicadas no código fonte da versão anterior (software legado, não foi desenvolvido utilizando a técnica TDD) e da nova versão (utilizando a técnica TDD) do software. Para isso, a mesma funcionalidade (os mesmos casos de uso) do software legado foram utilizados para desenvolver a nova versão do software.

As características e os casos de uso do software legado são descritos na Seção 6.2. As tecnologias utilizadas para o desenvolvimento da nova versão do software são mostradas na Seção 6.3. As características do código fonte da nova versão do software, abordando questões de utilização da técnica TDD e seus benefícios, são apresentadas na Seção 6.4. Os resultados obtidos em decorrência da comparação dos valores das métricas aplicadas no código fonte de ambas as versões do software são discutidos na Seção 6.5. Algumas dificuldades encontradas durante o desenvolvimento da nova versão são apresentadas na Seção 6.6.

6.2 O software simulador de bolsa de valores

O Software Simulador de Bolsa de Valores, também conhecido como Simulador de *Homebroker*, é um software que simula um ambiente de compra e de venda de ações em bolsas de valores. Esse software é uma aplicação Web,

cuja funcionalidade permite ao usuário (i) obter informações sobre cotações de ativos, (ii) visualizar dados em relação ao tempo, (iii) simular compra e venda de ativos, (iv) simular investimentos financeiros e (v) buscar/armazenar notícias referentes às ações da Bovespa (Bolsa de Valores de São Paulo).

O conteúdo das notícias da Bovespa é armazenado em um repositório de dados e utilizado para analisar o humor do mercado. Com esse conteúdo, pode-se realizar previsões sobre o comportamento do mercado financeiro utilizando inteligência artificial (por exemplo, lógica *fuzzy* (Lima *et al.*, 2010)). Inicialmente, apenas o núcleo desse software foi desenvolvido, sem a construção da interface com o usuário. Foram desenvolvidos apenas serviços que podem ser acessados usando URL's (*Uniform Resource Locator*) de acordo com a funcionalidade desejada.

A funcionalidade do software desenvolvido utilizando a técnica TDD é baseada nos casos de uso identificados na sua versão anterior (software legado), os quais são documentados no Diagrama de Casos de Uso sugerido na UML[3] (*Unified Modeling Language*) (Figura 6). O usuário desse software pode (i) Filtrar intraday, (ii) Filtrar notícias, (iii) Emitir ordem de compra/venda, (iv) Visualizar portifólio, (v) Visualizar intraday, (vi) Visualizar gráfico comparativo, (vii) Cadastrar usuário, (viii) Autenticar usuário, (ix) Visualizar custódia e (x) Visualizar extrato.

Alguns casos de uso, por não estarem implementados na versão previamente existente do software, não foram implementado na nova versão do software. Estes casos de uso foram descartados, pois, mesmo que fossem desenvolvidos, não seria possível efetuar comparações a fim de avaliar melhorias porque estariam presentes em apenas uma versão do software.

---

[3] http://www.uml.org/

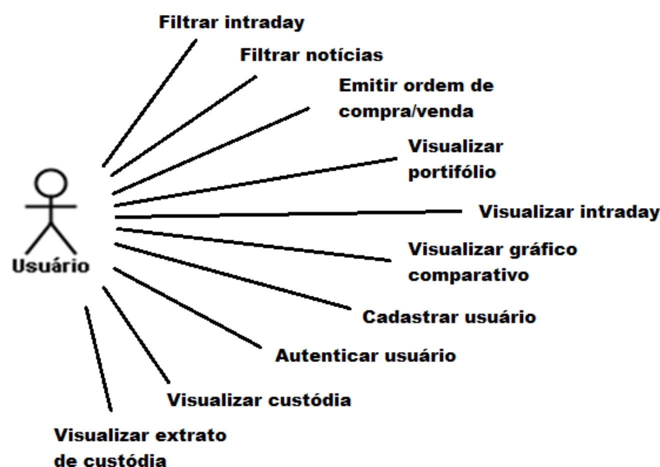


Figura 6 Casos de uso do software simulador de bolsa de valores

6.3 Tecnologias utilizadas

No desenvolvimento do Software Simulador de Bolsa de Valores, foi utilizada a linguagem de programação Java com o auxílio de três *frameworks*: vRaptor[4], Hibernate[5] e jUnit[6].

O *framework* vRaptor age no controle dos componentes do software, bem como na interação entre os módulos. Além disso, ele é o responsável pela construção de *servlets* que recebem as requisições HTTP (*Hypertext Transfer Protocol*) dos usuários. O objetivo da utilização do *framework* vRaptor é facilitar a construção de componentes voltados à plataforma Web. Sem a utilização desse *framework*, há a necessidade de realizar o tratamento das URL's usando *servlets* e a manipulação e a configuração de um extenso XML (*Extensible Markup Language*) que direciona as requisições.

---

[4] http://vraptor.caelum.com.br/
[5] http://www.hibernate.org/
[6] http://www.junit.org/

Esse *framework* controla as requisições do software por gerenciar os tipos das requisições HTTP. O desenvolvedor precisa direcionar para qual lógica do software uma determina requisição deve seguir. Além de levar a requisição do usuário até a lógica do software, ele trata as respostas do servidor que deve ser retornada ao usuário que fez a requisição, sendo a sua origem uma interface gráfica em um *browser* ou outro tipo de acesso.

O *framework* Hibernate abstrai o acesso ao banco de dados, ficando responsável por gerar as consultas necessárias nas tabelas de acordo com o serviço requisitado pelo usuário ou por uma requisição interna do software, sendo independente de interação externa. Além disso, ele é o responsável por estabelecer a conexão com o banco de dados e controlar a quantidade de conexões permitidas em um determinado momento. O *framework* Hibernate permite a construção das tabelas do software, incluindo chaves primárias e estrangeiras e relacionamentos, usando mapeamento das classes implementadas pelo desenvolvedor.

Pelo fato de abstrair o banco de dados, esse *framework* elimina a necessidade do desenvolvedor saber detalhes sobre o tipo de sistema gerenciador de banco de dados (SGBD) utilizado, o desenvolvedor apenas acessa a API disponibilizada pelo Hibernate. Com esta API, pode-se realizar consultas, exclusões, inserções e alterações sem diferenciar versões e tecnologias do SGBD em uso.

O framework jUnit pode ser utilizado em qualquer IDE (Netbeans, Eclipse ou outro) para realizar a execução dos testes automatizados. Além da execução dos testes, este framework fornece uma API em que o desenvolvedor pode utilizar suas funções para realizar verificações no código. Estas funções são responsáveis pela verificação de um determinado comportamento do código conforme previamente definido pelos desenvolvedores e na análise dos requisitos do software.

Este *framework* permite executar os testes individualmente ou em conjunto. Após a execução dos testes, a tela do framework jUnit mostra quantos testes falharam quantos testes obtiveram sucesso. Para realizar esta contagem, ele considera um teste os métodos que possuem a anotação @Test sobre sua assinatura. Desta forma ele exibe quais testes tiveram os resultados condizentes com os programados pelo desenvolvedor.

O Google CodePro Analytix é um *plug-in* desenvolvido na linguagem de programação Java para o Eclipse e é utilizado para analisar características de código fonte de um software, por exemplo repetição de código, dependências entre módulos e cobertura de código, e auxiliar nos testes.

Algumas das funções disponíveis na interface gráfica desse *plug-in* permite apresentar análises em formato de gráficos e tabelas e gerar relatórios no padrão HTML (*HyperText Markup Language*). Esse *plug-in* foi utilizado para o cálculo das métricas do código fonte das duas versões do software.

6.4 Características do código fonte

Existem muitas classes na nova versão do software pelo fato de cada uma ter sido escrita conforme as funções eram desenvolvidas. Em algumas situações eram identificadas necessidades específicas do componentes para serem transformadas em classes, o que contribuiu para esse aumento. Isto colaborou para as responsabilidades das classes ficarem bem divididas e localizadas. O teste unitário induziu o desenvolvedor a pensar em pequenas responsabilidades referentes àquela classe, sem misturar código fonte que, por conceito, deveriam ser responsabilidades de outras classes.

Além dessa divisão natural entre as classes, o que se pôde observar é as classes possuírem quantidade maior de métodos, pois cada ação era pensada de forma separada. Em decorrência do estilo de desenvolvimento, os métodos das

classes ficaram com menor quantidade de linhas de código fonte e menos complexos, pois faziam apenas o que deveriam fazer.

Em vários momentos do desenvolvimento, observou-se a necessidade de utilizar métodos existentes, que faziam partes de outra classe. Neste ponto, foi utilizado o conceito de Injeção de Dependências que consiste em adicionar um atributo do tipo da classe que possui um método na classe que o utiliza. Por exemplo, ao testar a classe A, observou-se a necessidade de utilizar um método da classe B; logo, a classe A recebe um objeto da classe B como parâmetro em seu construtor. Não compete ao objeto da classe A controlar a instância do objeto da classe B, apenas recebe como argumento no construtor e a dependência estava concretizada, bem como reaproveitamento do código fonte.

6.4.1 Utilização da técnica TDD

Por a técnica TDD ser um processo iterativo, no desenvolvimento do software, adotou-se uma abordagem passo a passo com etapas curtas, pensando na menor unidade possível a ser testada. Assim, mostrou-se presente uma importante característica no código fonte: a separação em várias unidades distintas, pois o código fonte se torna mais bem modularizado e com testes referenciando apenas uma unidade.

Por usar a técnica TDD, o código fonte foi escrito baseado em pequenas funções, uma por vez, conforme sua identificação no software. Além das funções que devem ter interação direta com o usuário, funções auxiliares foram desenvolvidas, por exemplo a conversão das datas para serem persistidas no banco de dados (a entrada é uma *string* e o retorno é um tipo de data específico da linguagem de programação Java).

Estas funções foram identificadas antes e durante o desenvolvimento do software, sendo algumas identificadas conforme o surgimento das necessidades

em decorrência da evolução do software. Este é um processo natural da técnica TDD, em que o desenvolvedor descobre durante a programação funções não identificadas na etapa Análise de Requisitos. As dificuldades enfrentadas no desenvolvimento são discutidas ao fim deste capítulo.

6.4.2 Utilização de *Mock Objects*

Em testes unitários quando, por exemplo, um objeto da classe A é testado, não se deve instanciar um objeto da classe B nos códigos dos testes automatizados, pois perde-se a ideia de testes unitários, passando a ser teste de integração (Beck, 2002; Fowler, 2007). Para tratar esta situação, o conceito de *Mock Objects* foi utilizado o qual consiste em um recurso no desenvolvimento orientado a testes por serem implementações "falsas" de um determinado componente que não se quer testar naquele momento. No exemplo mencionado, no teste da classe A, não há um objeto da classe B, mas um *Mock Object* desta classe (sem sua implementação real).

Em outras palavras, um *Mock Object* possibilita a simulação do comportamento do objeto original com os mesmos nomes dos métodos (Fowler, 2007). No entanto, o retorno dos "métodos" de um *Mock Object* é controlado pelo desenvolvedor dos testes, o qual fornece valores de retorno apropriados para não prejudicar a realização do teste. Por exemplo, se o objeto da classe B tem um método X que retorna um valor conforme alguma operação, o *Mock Object* desse objeto deve ter esse método X, mas com valor de retorno definido explicitamente pelo desenvolvedor. Além disso, a utilização de *Mock Object* permitiu fazer asserções mais abrangentes sobre os valores que deveriam ser retornados e que contribuíssem com o teste.

Conforme a evolução do software, observou-se a dependência entre seus componentes e, em certos momentos, foi identificada a necessidade de funções

dependentes que ainda não haviam sido escritas. A utilização de *Mock Objects* neste caso foi importante porque, ao invés de considerar a implementação real de uma classe, focou-se apenas nos nomes de cada função que deveria ser utilizada. O desenvolvimento foi voltado a interfaces e não à implementação, fator que aumenta a capacidade de abstração, pois somente o comportamento esperado foi definido, não havendo preocupação imediata em como esse comportamento seria programado.

6.4.3 Utilização de padrões de projeto

O desenvolvimento focado em uma interface de funções, não na implementação propriamente dita, proporcionou a criação de objetos mais genéricos, o que evita a escrita de códigos fonte repetitivos, como nas classes que acessam banco de dados e comportamentos semelhantes. Assim, o desenvolvimento foi pensado no comportamento que uma classe deveria ter, sem pensar em como elas seriam implementadas, em um primeiro momento. Por exemplo, foram criadas uma família de serviços e uma família de repositórios para controlar o armazenamento de dados, independente do recurso utilizado, ou seja, se é um banco de dados, um XML ou JSON[7] (*JavaScript Object Notation*).

As classes de serviços não precisavam saber qual objeto concreto de acesso a uma base de dados elas utilizam para executar as funções desejadas. Como nas classes de serviços elas tratam apenas o tipo abstrato, representado por uma interface, há a certeza de que qualquer repositório que implementasse aquela interface, existiria a funcionalidade requisitada. Desta forma, as classes de serviço se tornaram mais flexíveis a mudanças por causa desta abstração dos objetos que acessavam dados armazenados. Se houvesse a necessidade de mudar

---

[7] http://www.json.org/

o tipo de armazenamento de dados, estas modificações seriam feitas em outros objetos, mas não nas classes de serviços, que permaneceriam intactas e com a segurança de estarem utilizando as funções de uma interface.

Esta abstração caracterizou um nível razoável de polimorfismo, onde muitas classes não tinham seu uso explícito, usando sua instância direta. As classes que utilizavam este recurso apenas sabem da interface da família que pertence, mas não conhecem as classes que implementam esta interface. Dessa forma, a capacidade de mudança do código fonte tornou-se perceptível e a flexibilidade com a utilização uso de interfaces tornou as mudanças concentradas nas unidades em que se desejava agir, não se estendendo a outras classes. Além disso, a presença dos testes, nesse momento em que a utilização do polimorfismo estava mais sólida, contribuiu para a identificação das mudanças de comportamentos. Uma modificação em um método não deve mudar o retorno esperado, nem seu nome. Como a assinatura e o retorno mantêm-se os mesmos, as modificações fluem de maneira rápida e, caso algum comportamento tenha fugido do esperado, os testes estão presentes para sinalizar esta mudança.

Para auxiliar na distribuição destes componentes nas classes dependentes e tornar o polimorfismo mais sólido, foi utilizado o padrão de projetos Factory (Gamma *et al*., 2004) para estes tipos de objetos. Conforme o tipo do objeto que utiliza as dependências, esse padrão de projeto fornece o melhor objeto para aquele tipo de requisição. Assim, a classe dependente não possui necessidade de conhecer explicitamente qual recurso utiliza. Dessa forma, foi alcançado razoável nível de abstração com um código fonte legível, pois apenas os nomes das ações/métodos foram listados nas classes, não havendo preocupação com a implementação destas ações/métodos.

Um exemplo do polimorfismo alcançado no código fonte é apresentado na Figura 7, o qual representa uma classe que controla as requisições dos dados

de cotações, chamada DatadaysController. Este tipo de dados consiste em manipular os objetos Intraday e Diary, porém desenvolver um serviço para cada um deles seria repetitivo. Sendo assim, as requisições ficaram concentradas em uma classe que decide em tempo de execução qual tipo de objeto deve tratar (Intraday ou Diary), porém isso acontece longe dos olhos de quem lê a classe. Para isso, um objeto da classe que implementa o padrão de projetos Factory foi passada com argumento no construtor.

```
@Path("/datadays")
public class DatadaysController {

    private final Result result;

    private final ServiceFactory serviceFactory;

    private final DateConvertor convertor;

    public DatadaysController(ServiceFactory serviceFactory, DateConvertor convertor, Result result) {
        this.serviceFactory = serviceFactory;
        this.convertor = convertor;
        this.result = result;
    }
```

Figura 7 Injeção de Dependência em uma Classe e Utilização do Padrão de Projetos *Factory*

O padrão de projetos Factory fornece em tempo de execução o tipo de serviço a ser tratado. O controlador apenas necessita de uma fábrica de serviços, sem saber qual deles estará trabalhando. Para que isso seja possível, existe um tipo abstrato de dado conhecido apenas por quem o utiliza. A utilização deste serviço fornecido pela classe ServiceFactory é apresentada na Figura 8 e cuja implementação torna-se mais genérica por causa do polimorfismo presente. Um método retorna o serviço necessário e a classe apenas utiliza uma interface genérica, sem a necessidade de conhecer a implementação.

```
@Get
@Path("/{type}/{active.code}/{beginDate}/{endDate}.json")
public void search(Active active, String type, String beginDate, String endDate) {
    if(typeIsNotValid(type)){
        typeLinkErrorMessage();
    }else{
        Service<DataDays> service = getService(type);
        Date begin = this.convertor.convertStringToDate(beginDate);
        Date end = this.convertor.convertStringToDate(endDate);
        if(begin != null && end != null){
            try{
                List<DataDays> datadays = service.search(active, begin, end);
```

Figura 8 Utilização do Padrão de Projetos *Factory* e do Polimorfismo Presente no Controlador

Na última linha do código fonte apresentada Figura 8, pode-se observar a utilização do serviço. A interface Service é implementada pelos serviços do software. Portanto, as classes que utilizam o serviço não precisam saber qual é instanciado, pois isto é de responsabilidade de outro componente.

6.4.4 Benefícios alcançados

Um trecho de código fonte que testa o controlador (Figura 7 e Figura 8) é apresentado na Figura 9. O resultado foi um código fonte legível e simples, como é discutido na apresentação das métricas utilizadas para mensurar a característica de qualidade de manutenibilidade dos softwares desenvolvido e legado.

Durante o desenvolvimento do código fonte, foram identificadas semelhanças entre classes existentes; em seguida, generalizações foram feitas de maneira natural, sem planejamento anterior, tornando um processo intuitivo. Após a identificação destas semelhanças e redundâncias, foi realizada refatoração para eliminar estas situações, sendo parte do ciclo da técnica TDD responsável por tornar o *design* do código fonte mais simples. Na refatoração, as manutenções são executadas e pôde-se perceber as dificuldades em modificar o código fonte nesse momento do desenvolvimento. Durante a programação, esses problemas são difíceis de enxergar em um planejamento feito antes do

desenvolvimento, contudo, com a utilização da técnica TDD, eles foram abordados de maneira pontual, sendo corrigidos um por vez, de acordo com a função em desenvolvimento.

```
@Test
public void testNotFoundIntraday() throws Exception{
    this.intradays.clear();
    Mockito.when(this.serviceFactory.getService(Mockito.anyString())).thenReturn(serviceIntraday);
    Mockito.when(serviceIntraday.search(Mockito.any(Active.class), Mockito.any(Date.class), Mockito.any(Date.class)))
    .thenThrow(new NothingFoundException());
    String type = "intraday";
    String beginDate = "01-10-2011";
    String endDate = "13-10-2011";
    this.controller.search(active, type, beginDate, endDate);
    String serialized = this.result.serializedResult();
    assertEquals("{\"message\": \"Nenhum resultado encontrado\"}", serialized);
}

@Test
public void testControllerMethodSave() throws Exception{
    Mockito.when(this.serviceFactory.getService(Mockito.anyString())).thenReturn(serviceIntraday);
    String type = "intraday";
    DataDays dataday = Mockito.mock(Intraday.class);
    this.controller.save(type, dataday);
    String serializedResult = this.result.serializedResult();
    assertEquals("{\"message\": \"Dados armazenados com sucesso\"}", serializedResult);
}

@Test
public void verifyInvalidType() throws Exception{
    String beginDate = "01-10-2011";
    String endDate = "12-10-2011";
    this.controller.search(active, "teste", beginDate, endDate);
    String serializedResult = this.result.serializedResult();
    assertEquals("{\"typeError\": \"O link deve ser no formato /diary/ ou /intraday/\"}", serializedResult);
}
```

Figura 9 Trecho dos Códigos Fonte para Testar as Funções do Controlador *Datadays*

Pôde-se perceber que cada teste é realizado em uma única função. A construção do *design* das classes foi dirigido às funções e cada método possui responsabilidades bem curtas e limitadas, não fugindo do seu escopo. Além disso, os *Mock Objects* simularam o comportamento de classes que o controlador depende. Vale ressaltar que cada teste foi programado individualmente e não foi escrito outro teste antes que o atual obtivesse sucesso nas verificações e que o código fonte tenha sido refatorado de acordo com a percepção do desenvolvedor.

Neste ponto do desenvolvimento, percebeu-se as características proporcionadas pela técnica TDD no código fonte. Notou-se que cada método foi construído para fazer somente o que se espera dele, sem misturar responsabilidade. Assim, o desenvolvedor é direcionado a escrever vários métodos em uma mesma classe. Porém, em alguns momentos, foram identificados alguns métodos que poderiam ser mudados de classe, sendo um dos benefícios da refatoração do código fonte.

Durante o desenvolvimento do controlador mostrado na Figura 7, foram escritos métodos nas classes que ficaram encarregados de converter datas. Conforme a programação avançou, constatou-se a necessidade de utilizar estes métodos em outro controlador (controlador de notícias). Nessa constatação, esses métodos foram retirados do escopo da classe e alocados em uma classe encarregada por esta responsabilidade. Desta forma, as classes que dependiam destes métodos passaram a receber um objeto dessa classe como parâmetro (injeção de dependência). Na Figura 7, pode-se notar que a classe recebe um objeto da classe DateConvertor como parâmetro no construtor, sendo este responsável pela manipulação das datas.

O desenvolvimento dirigido a testes proporcionou reúso de código fonte durante a programação e ficou latente a caracterização da utilização de polimorfismo: classes de níveis mais altos não têm visibilidade das classes de nível mais fundo. A característica mais marcante foi que estas classes de níveis mais altos visualizavam apenas níveis intermediários, preocupando-se apenas em que ação deveria chamar. Em momento algum houve a necessidade de conhecimento de detalhes técnicos do software. Assim, as modificações tornaram-se mais simples do que o comum.

No entanto, não se pode afirmar que estes são benefícios exclusivos da técnica TDD. Desenvolvedores experientes podem alcançar este nível de abstração em decorrência do conhecimento em desenvolvimento de software e

por prever este tipo de situação. Nas próximas seções são comparadas características do código fonte de ambas as versões do software.

6.5 Resultados obtidos

O código fonte da nova versão do software possui características não exclusivas a ele, por exemplo o grau de abstração, pois estava presente no código fonte do software legado. O código fonte do software legado também possui bom nível de abstração na sua implementação, sendo facilmente notada a separação em camadas (três camadas: classes de interface de interação com o usuário, classes de lógica de negócio e classes de acesso a dados). A comparação, no entanto, mostrou que existe diferença entre os dois códigos fonte, por exemplo foi possível identificar simplicidade no código fonte da nova versão do software.

Após a obtenção dos valores das métricas utilizando o *plug-in* Google CodePro Analytix, houve um ganho em relação a manutenibilidade no código fonte da nova versão do software. A seguir, o *plug-in* é apresentado com mais detalhes a fim de mostrar os tipos de relatórios que ele pode fornecer.

6.5.1 Cálculo das métricas

A interface do Google CodePro Analytix com uma visão geral após ser aplicado os cálculos das métricas sobre o código fonte da nova versão do software é apresentada na Figura 10. A lista ao lado esquerdo são as métricas previstas pelo Google CodePro Analytix. Essas métricas podem ter sua visualização expandida nos pacotes do software, podendo visualizar as classes

separadamente. No lado direito, encontra-se um gráfico relativo à primeira métrica da lista, o qual mostra a razão entre classes abstratas e concretas.

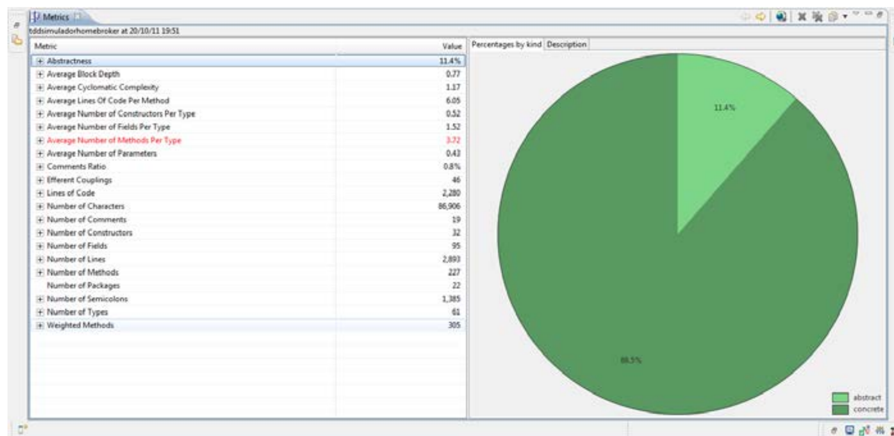


Figura 10 Visão Geral da Análise de Métricas no Google CodePro Analytix

Além desta interface, esse *plug-in* apresenta uma avaliação das dependências do código fonte em forma de grafo de dependência. Um grafo de dependência de primeiro nível da nova versão do software em relação aos *frameworks* utilizados e às bibliotecas de apoio é apresentado na Figura 11, sendo possível navegar nas dependências internas.

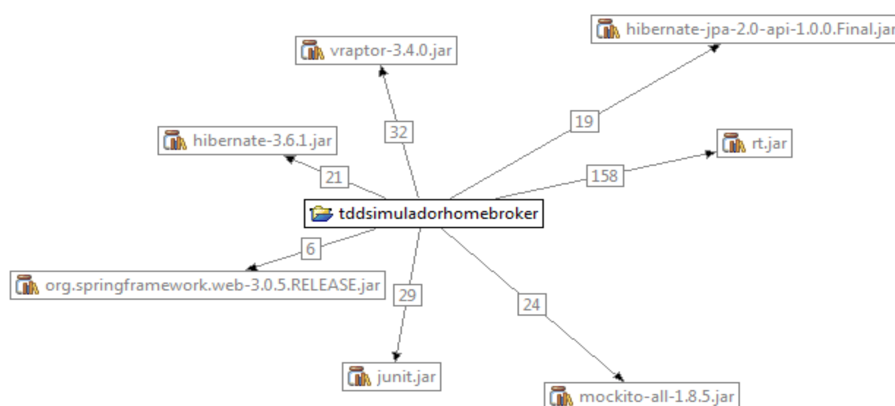


Figura 11 Visualização de Dependências Utilizando o Google CodePro Analytix

Além destes dois recursos, foi utilizada uma análise provida pelo *plug-in* que gera um documento em formato HTML. Esse documento apresenta tabelas com informações sobre as classes pertencentes a cada pacote na estrutura do código fonte e o grafo de dependência.

6.5.2 Resultado das métricas

Após a execução do *plug-in* Google CodePro Analytix, os valores das métricas foram calculados para o código fonte de ambas versões do software e são discutidos nesta seção. Os valores resultantes da aplicação das métricas no código fonte do software legado e da nova versão do software são apresentados na Figura 12 e na Figura 13, respectivamente.

As métricas consideradas foram *Weighted Methods*, *Average Lines Of Code Per Method*, *Average Cyclomatic Complexity*, *Efferent Couplings* e *Afferent Couplings* (Tabela 1). Os gráficos referentes a estas métricas são apresentados na Figura 14(*Weighted per Methods*, *Efferent Couplings* e *Afferent Couplings*) e na Figura 15 (*Average Lines Of Code Per Method* e *Average Cyclomatic Complexity*).

Para todas estas métricas, quanto maior o valor obtido, melhor será a classificação do código-fonte em que elas foram aplicadas.

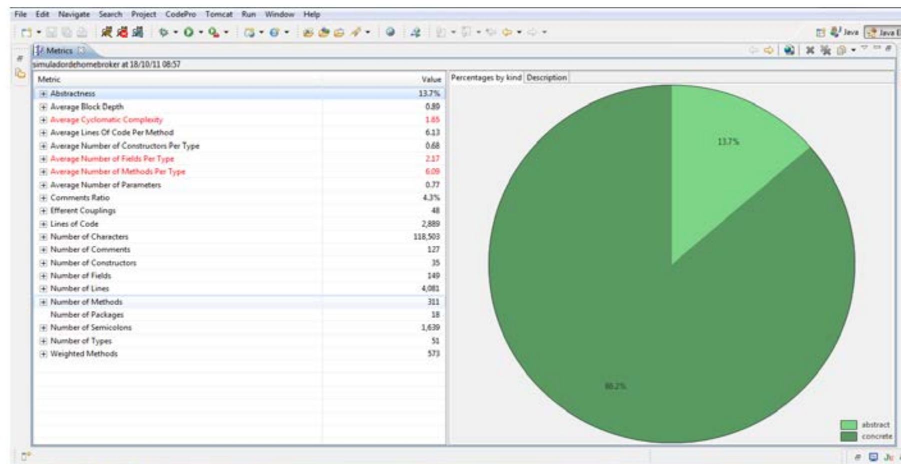


Figura 12 Medidas do Código Fonte do Software Legado

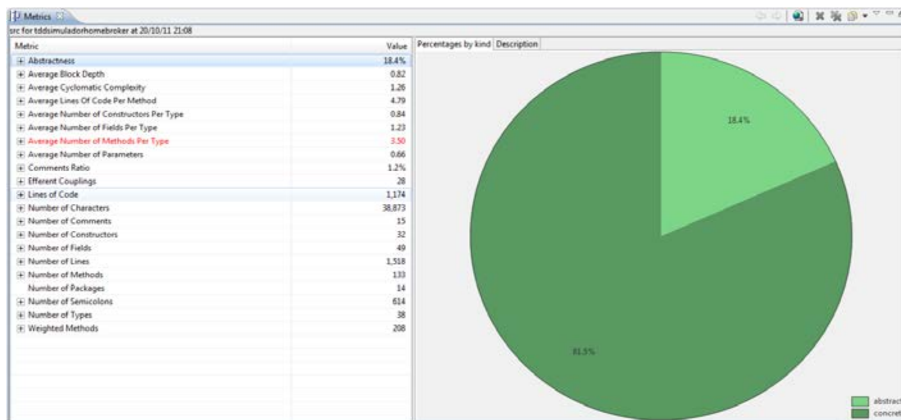


Figura 13 Medidas do Código Fonte do Software Desenvolvido Utilizando a Técnica TDD

Tabela 1 Resultados das métricas utilizadas

Código Fonte	*Weighted per Methods*	*Efferent Couplings*	*Afferent Couplings*	*Average Lines of Code per Method*	*Average Cyclomatic Complexity*
Sem TDD	573	48	20	6.13	1.65
Com TDD	305	28	18	4.79	1.26

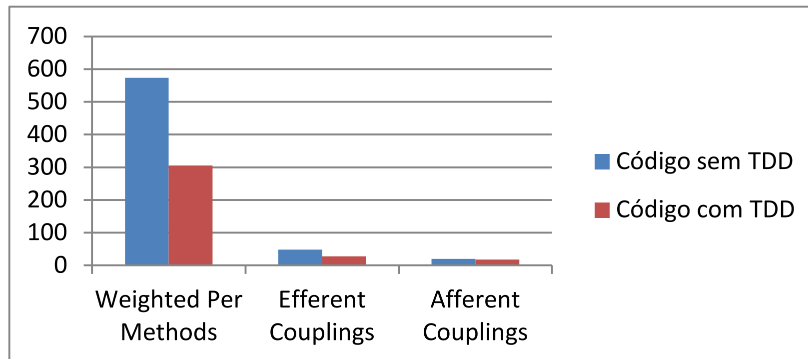


Figura 14 Valores do Código Fonte dos Dois Softwares para *Weighted per Methods, Efferent Coupling e Afferent Coupling*

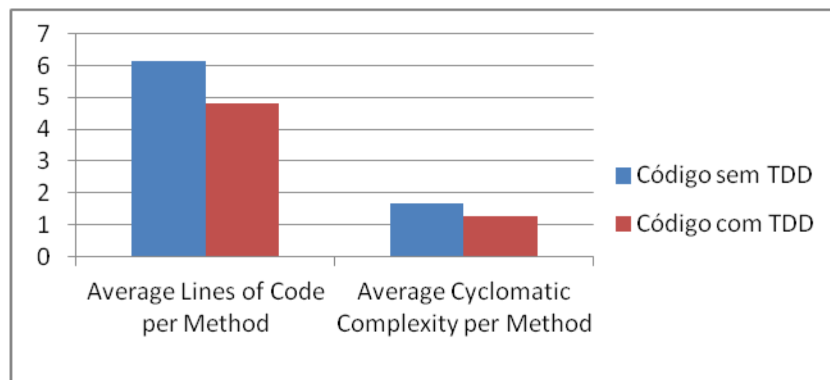


Figura 15 Valores do Código Fonte dos Dois Softwares para *Average Cyclomatic Complexity* e *Average Lines of Code per Method*

6.5.3 Análise dos resultados

Na primeira comparação (Figura 14), são confrontados os valores obtidos das métricas *Weighted per Methods*, *Efferent Couplings* e *Afferent Couplings* e pode-se perceber que o código fonte da nova versão do software possui vantagem em relação ao software legado. Isso ocorre por causa da programação orientada a testes, quando pequenas funções são implementadas

uma por vez, o que acarreta métodos menores, com apenas uma função considerada em cada escopo de teste. Desta maneira, o código fonte é direcionado a ter mais métodos com pesos menores, ou seja, menos responsabilidades.

As responsabilidades de cada funcionalidade ficaram divididas em vários métodos e classes, pois foi utilizado o conceito de delegação de ações. Um método delega atividades para outros métodos, concentrando-se apenas no que ele está designado a realizar como tarefa.

No código fonte do software legado, apesar de ter razoável grau de abstração, os métodos possuem mais pesos, como é mostrado pela métrica *Weighted per Methods*. Não se pode dizer que foi uma falha do desenvolvedor, mas, quando se desenvolve um software sem a prática de dividi-lo em pequenas iterações e funcionalidades, a tendência é criar métodos com mais ações do que deveriam.

Além do desenvolvimento ser passo a passo, a refatoração tende a diminuir as responsabilidades dos métodos. Na etapa de refatoração, dentro do ciclo da técnica TDD, é sugerido extrair trechos de um código fonte com responsabilidade extra. Na técnica TDD, o desenvolvedor deve realizar a refatoração no código fonte; assim, as chances de remover redundâncias e diminuir responsabilidades nos métodos são maiores. Isso é caracterizado na análise da métrica em questão.

A diferença entre métodos designados a realizarem a mesma tarefa é apresentada na Figura 16 e na Figura 17. Na primeira, encontra-se um método do software legado e, na segunda, encontra-se um trecho de código fonte designado à mesma tarefa, mas da nova versão do software. Pode-se perceber o aglomerado de código fonte apresentado na Figura 16. Porém, mesmo sem utilizar a técnica TDD, o desenvolvedor poderia utilizar da refatoração para facilitar a leitura do código fonte. Mas não há essa garantia e nem sempre o

desenvolvedor se atenta a esse fato. Como na técnica TDD o desenvolvedor é treinado para realizar a refatoração, essa prática é retratada no código fonte.

Enquanto no código fonte apresentado na Figura 16 as tarefas estão agrupadas dentro do laço de repetição, no código fonte apresentado na Figura 17, cada tarefa é um método, por exemplo getHeadline (...) e getContentText (...). Além disso, pode-se perceber acoplamento indesejado no código fonte da Figura 16. A classe em questão é concreta e busca apenas notícias de uma fonte, fato observado na chamada do método pesquisarNoticiaFromFOLHA (...). No código fonte da Figura 17, trata-se de uma implementação de uma classe de serviços que busca notícias, o que permitiu abstração da implementação de uma família de buscas. O código fonte apresentado faz parte de uma subclasse do serviço de busca notícias, sendo a classe abstrata, pois o controlador que chamar esse serviço não tem conhecimento de sua implementação. Ele terá conhecimento apenas que há retorno de uma lista de notícias; logo, a manutenção pode se tornar mais simples, não precisando de um tratamento para verificar a fonte da notícia.

```
public static List<Noticia> getNoticias( String palavraChave, Long indice ){
    List<Noticia> noticias = new ArrayList<Noticia>();
    String pesquisa = pesquisarNoticiaFromFOLHA(palavraChave, indice);
    String INICIO_LINK = ".</b> <a href=\"";
    String FIM_LINK = "\">";
    String FIM_MANCHETE = "</a><br>";
    String FIM_CONTEUDO = "<br>";
    while( pesquisa.indexOf( INICIO_LINK ) != -1 ){
        try{
            Noticia noticia = new Noticia();

            //pegar link
            pesquisa = pesquisa.substring( pesquisa.indexOf( INICIO_LINK ) + INICIO_LINK.length() );
            noticia.setLink( pesquisa.substring( 0, pesquisa.indexOf( FIM_LINK ) ) );

            //pegar manchete
```

F

```
@Component
public class NewsSearcherFromFolha extends NewsSearcher {

    public NewsSearcherFromFolha(RestTemplate restTemplate, DateConvertor dateConvertor) {
        super(restTemplate, dateConvertor);
    }

    protected List<News> list(String news){
        String[] arrayNews = news.split("<!--RESULTSET-->");
        String allNews = arrayNews[1];
        arrayNews = allNews.split("<!--/RESULTSET-->");
        allNews = arrayNews[0];
        allNews = allNews.substring(0, allNews.indexOf("<p>Mais resultados:"));
        convertNews(allNews);
        return list;
    }

    private void convertNews(String allNewsAsString){
        int x = 1;
        while(thereIsANews(allNewsAsString)){
            News news = new News();
            news.setLink(getLinkOfANews(allNewsAsString));
            news.setHeadline(getHeadline(allNewsAsString));
            news.setDate(getDate(news.getHeadline()));
            getContentText(allNewsAsString);
            news.setText(getContentText(allNewsAsString));
            list.add(news);
            String allNewsAsStringNew = removeLineRead(allNewsAsString, x, news);
            allNewsAsString = "";
            allNewsAsString = allNewsAsStringNew;
            x++;
        }
    }
```

Figura 17 Código Fonte de Manipulação de Notícias do Software Desenvolvido Utilizando a Técnica TDD

A segunda métrica apresentada na Figura 14 é *Efferent Coupling*, cujo valores mostram maior dependência de classes externas no código fonte do software legado. Esta métrica indica uma dependência entre as classes de escopos distintos (outros pacotes e ou outros módulos). No código fonte do software legado, o número de abstrações foi menor, havendo maior redundância no código fonte e vários componentes criados com códigos fonte com execução igual. Na nova versão do software, o número de classes abstratas foi ligeiramente maior, porém contribuiu para que muitos serviços e controladores fossem genéricos, diminuindo o número de classes concretas. Com essa diminuição, a quantidade de dependências diminuiu; enquanto no código fonte do software legado, vários componentes possuíam muitas dependências.

Por mais que as dependências externas caracterizem grau de acoplamento, elas são necessárias em certo momento. Durante o desenvolvimento do sotware e conforme as responsabilidades das classes identificadas, a tendência é injetar dependências nos construtores, para que um objeto possa delegar ao outro as funções que ele não deve exercer. Assim, é natural que sempre exista algum grau de dependência entre objetos de módulos diferentes, como relata esta métrica.

Por fim, a terceira métrica apresentada na Figura 14 é *Afferent Coupling*, cujos resultados obtidos não se pode concluir algo, pois o comportamento é praticamente o mesmo no código fonte dos dois softwares. Os números são semelhantes pois os serviços desenvolvidos em ambos os software são os mesmos. Mesmo que os módulos tenham sido reduzidos com a utilização da técnica TDD, os serviços em formas de funções se mantiveram, pois o objetivo do software não foi alterado.

Dois grafos de dependência são apresentados na Figura 18 (software legado) e na Figura 19 (nova versão do software), os quais mostram a relação entre os componentes do código fonte das duas versões do software, deixando visível as diferentes concentrações entre estas versões. No primeiro grafo, as dependências são mais dispersas, o que fortalece o fato apresentado anteriormente da possibilidade da existência de redundância do código fonte, pois vários objetos semelhantes podem possuir dependências semelhantes, mas são implementadas separadamente. No segundo grafo, a concentração de dependências é maior na nova versão do software, o que caracteriza a reutilização de um módulo.

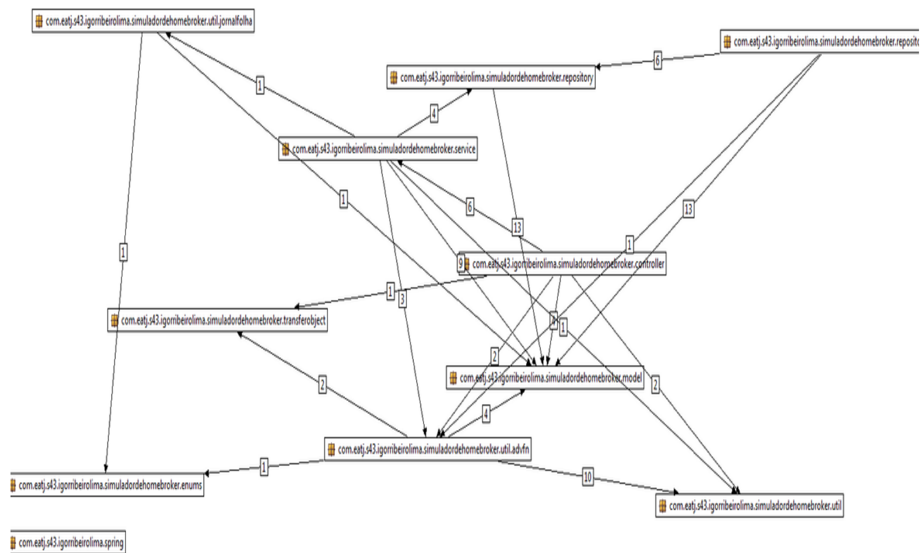


Figura 18 Grafo de Dependências entre Componentes do Software Legado

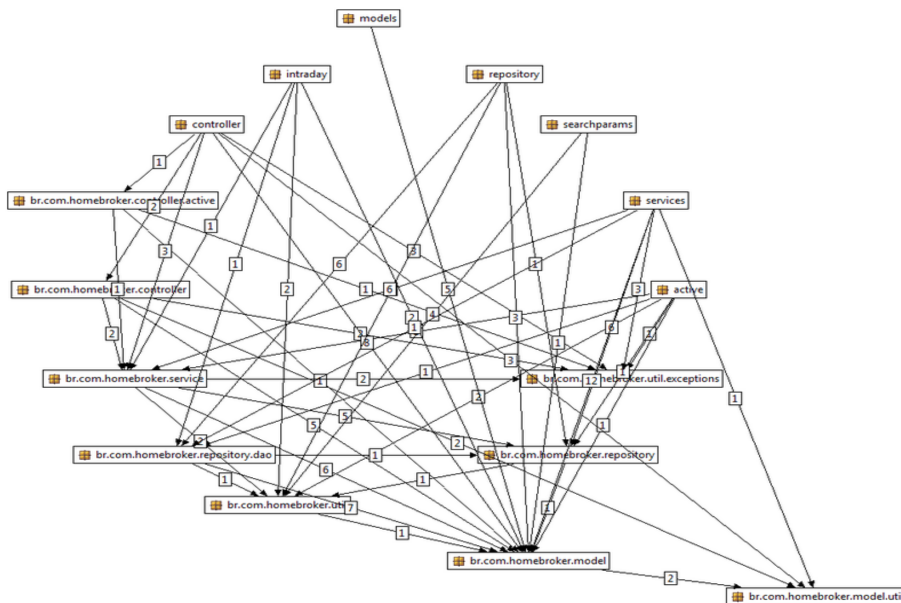


Figura 19 Grafo de Dependências entre Componentes do Software Desenvolvido Utilizando a Técnica TDD

As duas métricas restantes estão relacionadas às médias aritméticas entre os componentes do software. A métrica *Average Lines of Code per Method* considera a média de linhas por código fonte. No entanto, deve-se considerar que, por causa da utilização do *framework* Hibernate, há a exigência de métodos acessores (métodos *get* e *set*), cujo tamanho é de apenas uma linha de código fonte e contribuem para a redução da média do tamanho dos métodos. Esta é uma característica comum ao códigos fonte da duas versões do software, sendo considerada as médias de cada um, as conclusões não sofrem interferências pois as entidades envolvidas são as mesmas.

A média do tamanho dos métodos nos códigos fonte em que a técnica TDD não esteve envolvido são maiores. Este resultado reforça a afirmação da diferença de responsabilidades entre as duas versões do software. Uma menor média no tamanho dos métodos na versão orientada a testes mostra que há responsabilidade menor, ou seja, executa pequenas tarefas objetivas. As responsabilidades estão mais dividas em uma versão (nova versão do software) do que na outra (software legado). Neste ponto, a técnica TDD se mostra influente na estrutura do software.

O valor da métrica *Average Cyclomatic Complexity* é menor no código fonte da nova versão do software, por causa da constante utilização de polimorfismo. O fato de existir maior abstração em um código fonte faz com que existam menos ciclos, menor quantidade de estruturas condicionais que aumentam a complexidade ciclomática do código fonte.

6.5.4 Comparação entre os códigos fonte

Os resultados mostraram sensível diferença entre as duas versões do software. As métricas utilizadas mostraram melhor nível de manutenibilidade no código fonte da nova versão do software. Tais resultados foram condizentes com as expectativas que se tem ao utilizar a técnica TDD. A versão do código fonte da nova versão do software teve maior modularização de seus componente e maior distribuição das responsabilidades.

Uma característica marcante foi o reúso de código fonte. Os valores das métricas *Afferent Coupling* e *Efferent Coupling* combinados aos valores observados na *Weighted per Methods* mostraram que o software possui métodos com pouca responsabilidade e a concentração das dependências foi maior. Esta concentração sugere que um componente é utilizado por vários recursos diferentes, deixando as dependências menos espalhadas. Isso é um forte indício de reúso de código fonte, pois o componente tem sua funcionalidade compartilhada em vários outros objetos.

No software legado, existe razoável reúso de código fonte, o grafo de dependências mostra que os componentes tem alguma dispersão entre as funções. O fato de possuir valores das métricas maiores não significa que o código fonte seja de baixa qualidade. Contudo, esse grafo mostra ganho no aspecto de reúso de código fonte com a utilização da técnica TDD.

Ambas as versões do software foram desenvolvidos utilizando camadas (três camadas: camada de interface gráfica, camada de lógica de negócio e camada de acesso a dados); por isso, é esperado a existência de distribuição de responsabilidades entre os seus componentes. Observando o grafo de dependências, pode-se visualizar que a utilização da técnica TDD ofereceu melhoria no *design* do código fonte. Unido a estes fatos, a métrica *Average*

Lines of Code per Method reforça ainda mais o melhor *design* que a técnica TDD proporcionou por direcionar o desenvolvimento do código fonte mais sucinto.

Apenas observando o resultado da métrica *Average Lines of Code per Method*, não se pode garantir que o código fonte obteve ou não melhor *design*. Porém, a combinação entre estes valores e os valores de acoplamento e coesão podem levar a conclusão que o código fonte possui responsabilidades bem definidas nas classes e que cada método realmente faz o que realmente deve fazer, sem fugir do seu escopo. A técnica TDD guiou o desenvolvimento para que cada método pudesse apenas executar a tarefa prevista no teste. Algum comportamento que não fizesse parte do objetivo de um teste, mas que fosse necessário naquele momento, era delegado a uma dependência que o classe sendo testada possuísse, fato que caracteriza a redução de responsabilidade de um método.

Esta redução de responsabilidades do código fonte, aliada ao nível de abstração que o software, acarretou em um valor menor para a métrica *Average Cyclomatic Complexity*. O valor das métricas aplicadas no código fonte da nova versão do software é relativamente menor quando comparados com os valores obtidos no código fonte do software legado.

O grau de polimorfismo fornecido pela técnica TDD contribuiu para que o código fonte se tornasse mais genérico. Esta situação diminuiu o valor da métrica *Average Cyclomatic Complexity* nas classes da camada lógica de negócio que ficou encapsulada. As camadas de interface e de lógica de negócios da nova versão do software "enxergavam" apenas a abstração do código fonte, o que contribuiu para a diminuição do valor da métrica *Average Cyclomatic Complexity*.

A combinação do valor das cinco métricas selecionadas leva a constatação de alguns fatos interessantes: o código fonte da nova versão do

software é simples, o tamanho dos métodos (linhas de código fonte) é menor e as classes possuem responsabilidades bem definidas. Tais fatos podem caracterizar um alto grau da característica de qualidade de manutenibilidade no código fonte da nova versão do software.

Modificar um componente que tem responsabilidade definida é mais simples do que dar manutenção em códigos fonte que possuem muitas ações. As métricas mostram que os métodos são pequenos, fator que também determina uma boa manutenção. Também, existe o fato de todos os trechos de código fonte possuir um teste automatizado.

Os níveis de abstração atrelados a um código fonte mais simples fornecem razoável capacidade de mudança. Em um momento que haja uma necessidade de alterar a lógica de negócio, esta tarefa é mais natural, pois as camadas que a utilizam "enxergam" apenas a sua abstração. Estas classes não conhecem a implementação, apenas o que a lógica de negócio deve fazer, o que restringe a modificação à unidade. Além disso, como os métodos possuem poucas linhas de código fonte, a capacidade de identificar o que deve ser alterado é maior.

Portanto, ao modificar os componentes, devem-se executar os testes e observar se houve alguma mudança de comportamento imprevisto no software. Isso foi constatado no desenvolvimento da nova versão do software, pois, ao realizar a programação, abstrações eram criadas e modificações eram feitas com rapidez por cauda da generalização do software, que o tornou flexível. Os testes também sinalizavam instantaneamente se alguma modificação alterou os resultados esperados do software.

Porém, mesmo possuindo grau de manutenibilidade inferior ao obtido com a nova versão do software, o código fonte do software legado possui valor razoável para esse grau. O grafo de dependência, ainda que mais disperso,

mostra razoável divisão de responsabilidades para o código fonte do software legado.

6.6 Dificuldades com a aplicação da técnica TDD

Diversos fatores devem ser considerados ao analisar a diferença de característica do código fonte de ambas as versões do software. Um deles é a experiência de programação dos desenvolvedores envolvidos (aquele que desenvolveu o software legado e aquele que desenvolveu a nova versão do software). Mesmo quando não se utiliza a técnica TDD, um programador pode obter alto nível de manutenibilidade de código fonte. No caso do software legado, o desenvolvedor não utilizou a técnica TDD, mas possui longa experiência em desenvolvimento de software.

A técnica TDD facilita aumentar o grau da característica de qualidade de manutenibilidade, porém não é a única forma de chegar a esse ponto. Sendo assim, um desenvolvedor experiente consegue atingir valores compatíveis dessas métricas. As diferenças ocorrem por cauda do estilo de programação, pois a programação pensando em uma interface é alcançada mais facilmente na técnica TDD.

A experiência do desenvolvedor com a técnica TDD também deve ser maior para que os benefícios sejam alcançados. O desenvolvimento utilizando a técnica TDD não é tão trivial, quanto maior a experiência do desenvolvedor com a técnica, melhor podem ser os resultados (Crispin, 2006).

A falta de uma definição extensa e concreta sobre os requisitos do software causou um déficit nas funções desenvolvidas. Em muitos momentos, foi necessário uma especificação mais detalhada para que os testes fossem elaborados com eficiência. Alguns testes tiveram que ser excluídos, pois em determinado instante eles pareciam ser úteis, mas em outros ele não fazia

"sentido" dentro do rumo que a arquitetura tomava. Este foi um fator determinante para que o desenvolvimento utilizando a técnica TDD se aproximasse do desenvolvimento tradicional em certo momento. Se os testes não são tão eficientes, perdem-se alguns dos benefícios da técnica (pensar nos testes antes de iniciar a implementação do código fonte).

Quando a equipe de desenvolvimento tem grande experiência com a técnica TDD e os requisitos foram analisados visando ao desenvolvimento orientado a teste, os desenvolvedores possuem documentação mais próxima dos testes unitários a serem utilizados. Logo, os ganhos serão maiores. No caso deste trabalho, não houve esta especificação, o que pode ter causado alguma perda no potencial da utilização da técnica TDD.

6.7 Considerações finais

A utilização da técnica TDD proporcionou melhoria sensível no grau de manutenibilidade no código fonte da nova versão do software, mesmo havendo diferença de experiência entre os desenvolvedores de cada versão. O desenvolvimento da nova versão do software resultou em um código fonte simples e legível. Os objetivos de melhorar a capacidade de manutenção deste código fonte foram alcançados.

As métricas utilizadas neste trabalho mostraram que a utilização da técnica TDD contribuiu nos principais aspectos para tornar o código fonte fácil de manter. O código fonte se tornou compacto, as classes tiveram responsabilidades bem definidas e a complexidade do código fonte diminuiu.

Sendo assim, a característica de qualidade de manutenibilidade no código fonte da nova versão do software foi superior ao código fonte do software legado. Modificar um código fonte que possui métodos com menos linhas de código fonte e maior abstração de recursos é menos custoso, por causa

da facilidade que o desenvolvedor terá de entender o código fonte antes de modificá-lo, do que modificar códigos fonte em que seu entendimento é complexo.

7 CONSIDERAÇÕES FINAIS

7.1 Conclusões

Utilizar a técnica *Test-Driven Development* (TDD) para desenvolver software não é trivial. Seguir os passos corretos de maneira a obter os benefícios que esta técnica pode proporcionar exige disciplina, pois altera a maneira tradicional de desenvolver software. Esta mudança brusca no estilo de programar é o principal desafio no início da programação usando essa técnica.

Porém, seus benefícios são rapidamente percebidos. A cada passo durante a construção do código fonte é possível visualizar e perceber o software assumindo um *design* enxuto e objetivo, com um código fonte legível e fácil de entender. Estas percepções foram confirmadas ao fim do desenvolvimento do software utilizado como estudo de caso. Comparando o código fonte das duas versões do software, foi possível obter resultados que mostraram melhor legibilidade no código fonte no software desenvolvido utilizando a técnica TDD. Dessa forma, a manuteniblidade do software é melhorada, tendo em vista que a fácil legibilidade do código fonte ajuda na sua compreensão. O desenvolvimento com a técnica TDD proporcionou um código fonte com maior nível abstração, o qual foi ponto inicial para que outras vantagens pudessem ser alcançadas. Esta abstração fez com que trechos de código fonte (componentes/módulos) com lógica mais complexa ficassem encapsulados e escondidos de trechos de código fonte que os utilizam. Se um componente não conhece a complexidade de outro componente em que ele depende, o código fonte se torna mais flexível, pois não existe a preocupação de sabe o “como” ele realiza sua função, mas em saber "qual" ("o que") é sua função.

A técnica TDD direcionou o desenvolvimento do software para testes, sendo focado na funcionalidade, portanto o código fonte das classes foram

objetivos, não realizando tarefas que fugissem ao seu escopo. A cada passo, o código fonte se tornava mais simples e as responsabilidades de cada método ficava mais dividida entre as dependências.

Além disso, os resultados mostraram diferença no grau de acoplamento presente no código fonte dos dois softwares. Ambos possuem algum nível de acoplamento por causa da utilização de funções de uma classe dentro de outras classes. Contudo, a utilização da técnica TDD permitiu o desenvolvimento do software com grau menor de acoplamento. As métricas mostraram que, ao utilizar essa técnica, o código fonte se tornou concentrado e o reúso foi maior. Desta maneira, a tarefa de manutenção era menos árdua, pois o desenvolvedor não precisa entender vários componentes distintos para realizar a modificação em uma função do software. Esta modificação concentra-se em um menor quantidade de classes.

Diversas modificações foram realizadas durante o desenvolvimento e a característica de qualidade de manutenibilidade mostrou-se constantemente presente, o que pode ser percebido com a aplicação constante das métricas estudadas. Enquanto se realizava as refatorações como parte do ciclo da técnica TDD, as modificações eram simples e rápidas. Além do fato das classes não possuírem responsabilidades em excesso, os testes automatizados escritos para cada função indicavam se a modificação realizada mudava o comportamento esperado do software (teste de regressão). Assim, a programação fluía com naturalidade, pois existia a certeza do funcionamento do software não ser comprometido a cada componente criado ou modificado.

Ao final, obteve-se um software com menos linhas de código fonte, menos linhas por método e complexidade menor com a utilização da técnica TDD, mas existem semelhanças entre os códigos fonte. Essas semelhanças foram identificadas com maior clareza durante a etapa de refatoração e redundâncias foram eliminadas. Com o passar do tempo, o código fonte se

tornou menos redundante e a possibilidade de reúso do código fonte atingiu níveis não perceptíveis no software legado. Este fato fez com que as responsabilidades dos métodos diminuíssem, o que pode ser visto nos valores das métricas *Weighted per Methods* e *Average Lines of Code per Method*. Posteriormente, o grau de acoplamento e a complexidade dos algoritmos obtiveram valores melhores, como mostraram as métricas *Efferent Couplings*, *Afferent Couplings* e *Average Cyclomatic Complexity*.

Diante destes fatos, pode-se concluir que houve melhora significativa na qualidade do código fonte do software desenvolvido como estudo de caso. O software, além de estar funcional (garantido pelos testes), possui *design* simples, que acarreta facilidade em modificá-lo posteriormente. Além de melhorar a característica de qualidade de manutenibilidade, existe a possibilidade de reaproveitar código fonte na construção de um novo módulo e o *feedback* instantâneo dos testes que podem informar se a nova função em desenvolvimento afeta ou não de forma negativa o comportamento do software.

7.2 Contribuições

Este trabalho contribuiu na análise do código fonte de um software quanto a característica de qualidade de manutenibilidade quando é desenvolvido com estilo tradicional (realização de testes após o desenvolvimento de algum código) e com a utilização da técnica TDD (realização casos de testes antes do desenvolvimento de algum código). Foi possível perceber que a técnica TDD pode ser uma alternativa para desenvolver um software manutenível.

Com base nos resultados coletados, o desenvolvedor pode optar por escolher esta técnica visando a construção de um código fonte mais compreensível a outros desenvolvedores que não estiveram envolvidos inicialmente na sua programação. Assim, a informação é mais bem difundida

entre a equipe de desenvolvedores e de mantenedores dos software e as manutenções podem ser feitas de maneira menos árdua pelas pessoas envolvidas na continuação deste software.

Como consequência dessa análise, foi realizada a reengenharia um software existente, porém apenas em nível de código, para identificar a funcionalidade do software legado, pois sua documentação não existe. Essa funcionalidade permitiu guiar a elaboração dos testes para o desenvolvimento da nova versão do software utilizando a técnica TDD.

7.3 Trabalhos futuros

Dentre as possibilidades de um futuro trabalho, pode-se pensar na análise da utilização da técnica TDD em um ambiente em que, no desenvolvimento de um software, há maior número de desenvolvedores envolvidos. Neste ambiente, poderia ser verificada a difusão do código fonte entre as diferentes pessoas envolvidas na sua construção e qual a facilidade delas compreenderem o código fonte feitos por outros membros da equipe. Além disso, poderia ser analisado o reúso de código fonte alcançado na equipe, com a utilização de componentes desenvolvidos separadamente. A abordagem em equipes maiores pode envolver desafios como a análise de troca de informações entre seus membros e a observação da produtividade que eles alcançaram.

Além de analisar a característica de qualidade manutenibilidade, poderiam ser analisadas outras características de qualidade do software apresentadas na norma ISO/IEC 9126 que podem ser melhoradas com o uso da técnica TDD. Outra sugestão de continuidade é avaliar a capacidade da equipe em entregar software com menor quantidade de falhas e a compatibilidade com os requisitos do cliente.

Mais uma sugestão é a utilização da técnica TDD em um projeto de software para analisar a capacidade de aplicar manutenção neste software por

parte de programadores que não participaram desta equipe. Utilizando entrevistas com desenvolvedores e observação das atividades de manutenção, verificar a facilidade que esta equipe externa encontrou ao realizar modificações ou criar novos módulos neste software e analisar a manutenibilidade do software sob esta verificação. Para realizar esta análise, poderiam ser utilizadas métricas de projeto e métricas de processo, para que fossem verificadas questões relacionadas ao trabalho das pessoas da equipe de desenvolvimento.

Outra sugestão é o desenvolvimento de interfaces gráficas para interação do usuário com o software de maneira que o usuário pudesse utilizar os serviços desenvolvidos no software.

Por fim, aliada a análise da manutenibilidade do código sob o ponto de vista abordado neste trabalho, poderia ser considerada a nomenclatura de pacotes, de classes e de métodos. Há tendência na técnica TDD em atribuir identificadores para nome de métodos que traduzem o que ele deve estar fazendo. Portanto, poderia ser verificado o impacto que a nomenclatura dos métodos tem na percepção de pessoas que dão manutenção em um código e como podem contribuir para melhorar a capacidade de manutenção do software.

REFERÊNCIAS

Aguayo, M. T. V.; Guerra, A. C.; Colombo, R. M. T. Avaliação da Manutenibilidade de Produtos de Software. Conferência IADIS Ibero-Americana WWW/Internet, p. 432-436, 2005.

Aguayo, M. T. V.; Guerra, A. C.; Colombo, R. M. T. Processo de Avaliação da Manutenibilidade de Produtos de Software. VII Simpósio Internacional de Melhoria de Processos de Software, São Paulo, SP, 2005.

Akingbehin, K. Taguchi Smaller-the-Best Software Quality Metrics. 10TH ACIC International Conference on Software Engineering, Artificial Intelligences, Networking and Parellel/Distributed Computing, p. 585-588, 2009.

Astels, D. Test-Driven Development: A pratical Guide, 2003. 592 p.
Beck, K. Extreme Programming Explained: Embrance Change. ISBN, 2ª edição, 2004.

Beck, K. Test Driven Development By Example. 2002, 240 p.

Bhat, T.; Nagappan, N. Building Scalable Failure-proneness Models Using Complexity Metrics for Large Scale Software Systems. XIII Asia Pacific Softwre Engineering Conference (ASPEC), p. 361-366. 2006.

Canfora, G.; Visaggio, C. A. Measuring the impacto f testing on code structure in Test Driven Development: metrics and empirical analysis. First International Symposium on emerging trends in software metrics, 2009.

Chen, J.; Huang, S. An Empirical Analysis of the impact of software development problem factors on software maintainability. The Journal of Systems and Software, p. 981-992, janeiro de 2009.

CMMI. Wibbas CMMI Browser. Disponível em http://www.cmmi.de/#el=CMMI/0/HEAD/folder/folder.CMMI. Acessado em: 02/11/2011.

Crispin, L. Driving Software Quality: How Test-Driven Development Impacts Software Quality. IEEE Software, p. 70-71, Novembro de 2006.

Dal Moro, R.; Falbo, R. A. Uma Ontologia par ao Domínio de Qualidade de Software com Foco em Produtos e Processos de Software. 3rd Workshop on

Ontologies and Metamodeling Software and Data Engineering, XXII Simpósio Brasileiro de Egenharia de Software – SBES, 2008.

Duarte, C; Falbo, R. A. Uma ontologia de qualidade de software. Workshop de Qualidade de Software. Joao Pessoa, p. 275-285, Outubro de 2000.

Eick, S. G.; Graves, T. L.; Karr, A. F.; Marron, J. S., Mockus, A. Does Code Decay? Assessing the Evidence from Change Manangement Data. IEEE Transactions on Software engineering, vol. 27, p. 1-12, 2001.

Eski, S.; Buzluca, F. An Empirical Study on Object-Oriented Metrics and Software Evolution in order to Reduce Testing Costs by Predicting Change-Prone Classes. Fourth International Conference on Software Testing, Verification and Validation Workshops, p. 566 – 571, 2011.

Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. Refactoring: Improving the Design of Existing Code. 464 p. 1999.

Fowler, M. Mock aren't Stubs, 2007. Disponível em: http://martinfowler.com/articles/mocksArentStubs.html. Último acesso em: 15/11/2011.

Freeman, E.; Freeman, E.; Bates, B.; Sierra, K.; Robson, E. Head First Design Patterns. O'Reilly Media. 688 p. 2004.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. 1994. 416 pages.

Gyimóthy, T.; Ferenc, R.; Siket, I. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. IEEE Transactions on Software Engineering, vol. 31, n. 10, p. 897 – 910, Outubro de 2005.

Honglei, T.; Wei, S., Yanan, Z. The Research of Software Metrics and Software Complexity Metrics. International Forum on Computer Science-Technology And Applications, p. 131-136, 2009.

ISO/IEC 12207: 2008. Disponível em http://www.iso.org/iso/catalogue_detail?csnumber=43447, acessado em 10/11. 2011.

ISO/IEC 9126-1: 2001. Disponível em http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749, acessado em 10/11/2011.

Izurieta, C.; Bieman, J. M. How Software Designs Decay: A Pilot Study of Pattern Evolution. First International Symposium of Empirical Software Engineering and Measurement, p. 449-451, 2007.

Janert, P. K. Introducing Test-Driven Software Development. IEEE Software, p. 100-101, novembro de 2004.

Jazen, D. S.; Saiedian, H. Does Test-Driven Development Really Improve Software Deisgn Quality? IEEE Software, p. 77-84, abril de 2008.

Jazen, D. S.; Saiedian, H. Test-Driven Development: Concepts, Taxonomy and Future Direction. IEEE Computer Software, p. 43-50, 2005.

Johnson, M. J.; Maximilien, E. M.; Ho, C.; Williams, L. Incorporating Performance Testing in Test-Driven Development. IEEE Software, p. 67-73, maio de 2007.

Jung, C. F. Metodologia aplicada a projetos de pesquisa: Sistemas de Informação & Ciência da Computação. Taquara, 2009. 1 CD-ROM.

Kan, S. H. Metrics and Models in Software Quality Engineering, 2ª edição. Addison Wesley, 560 p. 2003.

Koru, A. G.; Emam, K. E. The Theory of Relative Dependency: Higher Coupling Concentration in Smaller Modules. IEEE Software Measurement and quality, p. 81-89, 2010.

Lehman, M. M.; Belady, L. Program Evolution: Processes of Software Change. Academic Press, 1985. 538p.

Lima, I. R.; Costa, H. A. X.; Sugano, J. Y.; Parreira Júnior, P. A.; Souza, S. M. CellInvest - Um Sistema Especialista Móvel para Auxiliar na Tomada de Decisões em Mercado de Capitais. Congresso Brasileiro de Sistemas Fuzzy (CBSF), 2010. p. 455-463.

Martin, R. C. Professionalism and Test-Driven Development. IEEE Software, p. 32 – 36, maio de 2007.

Meananeatra, P.; Rongviriyapanish, S. Using Software Metrics to Selecet Refactoring for Long Method Bad Smell. The 8th Electrical Engineering/Eletronics, Computer, Telecommunications and Information Technology (ECTI) Association of Thaliand - Conference 2011, p. 492-495, 2011.

Meirelles, P. R. M. Levantamento de Métricas de Avaliação de Projetos de Software Livre. 78 p. São Paulo, 2008.

Osborne, W. M.; Chikkofsky, E. J. Fitting Pieces to the Maintenance Puzzle. In: IEEE Software. p. 10-11. Jan 1990.

Petrasch, R. The Definition of Software Quality: A Practical Approach. 2009.

Pfleeger, S. L.; Atlee, J. M. Software Engineering: Theory and Practice. Prentice Hall. 2009. 792 pages.

Ping, L. A Quantitative Approach to Software Maintainability Prediction. International Forum on Information Technology and Applications, p. 105-108, 2010.

Prasad, L.; Nagar, A. Experimental Analysis of Different Metrics (Object-Oriented and Structural) of Software. First International Conference on Computational Intelligence, Communication Systems and Networks. p. 235-240, 2009.

Pressman, R. S. Software Engineering - A practitioner's Approach. 2010, 860 p.

Riaz, M.; Mendes, E.; Tempero, E. A Systematic Review of Software Maintainability Prediction and Metrics. Third International Symposium on Empirical Software Engineering and Measurement. p. 367-377, 2009.

Rincon, A. M. Qualidade de Software. XI Encontro de Estudantes de Informática do Tocantins. p. 75-86, Palmas, 2009.

Robillard, M. What Makes APIs Hard to Learn? Answers from Developers. IEEE Computer Software, p. 27-34, 2009.

Shrivastava, D. P.; Jain, R. C. Unit Test Case Design Metrics In Test Driven Development. International Conference on Communications, Computing and Control Applications (CCCA). p. 1-6, 2011.

Soares, M. S. Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software. Revista Eletrônica de Sistemas de Informação, Vol. 3, No 1, 2004.

SOFTEX (2009), "MPS.BR: Melhoria de Processo do Software Brasileiro". Disponível em http://www.softex.br/mpsbr/, aceassado em 02/11/2011.

Soliman, T. H.; El-Swesy, A.; Ahmed, S. H. Utilizing CK Metrics Suite to UML Models: a Case Study of Microarray MIDAS Software. The 7th International Conference On Informatics and Systems (INFOS), p. 1-6, 2010.

Sommerville, I. Software Engineering. Addison Wesley. 2010. 792 pages.
Torchiano, M.; Silliti, A. TDD = Too Dumb Developers? Implication of Test-Driven Development on maintainability and comprehension of software.IEEE 17th Internacional Conference on Program Comprehension, p. 280-282, 2009.

Ullah, M. I.; Zaidi, W. A. Quality Assurance Activities in Agile - Philosophy to Practice. Master Thesis, Blekinge Institute of Technology, Suécia, 2009.
Vodde, B.; Koskela, L. Learning Test-Driven Development by Counting Lines. IEEE Software, p. 74-79, maio de 2007.

Vu, J. H.; Frojd, N.; Shenkel-Therolf, C.; Jazen, D. Evaluating Test-Driven Development in a Industry-sponsored Capstone Project. Sixth Internacional Conference on Information Technology: New Generations, p. 229-234, 2009.

Wasmus, H.; Gross, H. Evaluation of Test-Driven Development - An industrial Case Study. Software Engneering Research Group, Departament of Software Technology, 2007, 12 p.

Wirfs-Brock, R. J. Design For Test. IEEE Software. p. 92-93, Outubro de 2009.

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. Quality, v. 105, GI, p. 231-244, 2007.