Using TDD for Developing Object-Oriented Software - A Case Study

Ramon Gonçalves
Department of Computer Science
Federal University of Lavras
Lavras, Brazil
raigons@gmail.com

Igor Lima
Department of Computer Science
Federal University of Lavras
Lavras, Brazil
igorlima@comp.ufla.br

Heitor Costa
Department of Computer Science
Federal University of Lavras
Lavras, Brazil
heitor@dcc.ufla.br

***Abstract*— Maintenance of software is accomplished to meet the users' needs of this software (evolution/correction). But, it can become hardest if the source code architecture is difficult to understand. Test Driven Development technique can be used to reduce this difficulty, because it leads the developer to build software with source code simpler. In this paper, this technique is employed to develop software whose functionality is the same of legacy software, but it was developed of way traditional, to obtain more maintainable source code. Software metrics were applied in the source code of legacy and developed software and the results showed improvements in maintainability.**

***Keywords*— TDD; Software Evolution; Software Quality**

I. INTRODUÇÃO

Com a crescente demanda por novos sistemas de software [14], muitas empresas desenvolvedoras surgiram, existindo muitas pessoas envolvidas no processo de desenvolvimento. Com a mudança nas necessidades empresariais, esses sistemas também mudaram, ganhando novas funções e aumentando sua complexidade, o que acarretou acréscimo no custo para quem os desenvolve e para quem os financia [1]. Assim, a manutenção de sistemas de software é essencial, pois eles devem estar em acordo com as necessidades do meio no qual estão implantados para continuarem úteis.

Por causa da alta complexidade do código de sistemas de software legados, manutenções tornam-se árduas e demandam tempo para entendê-los, o que pode frustrar as expectativas dos clientes [15]. Assim, surge a preocupação com a legibilidade de código para tentar minimizar o esforço em realizar manutenções nesses sistemas [9]. Porém, essa preocupação pode não ser intrínseca das pessoas envolvidas no processo de desenvolvimento ou de manutenção de sistemas de software, pois pode ser difícil de atingir. Um código legível significa ter planejamento antes de começar a escrevê-lo, exigindo perícia do programador, tempo de aprendizado e esforço [1]. Para facilitar essa tarefa e guiar desenvolvedores a construir código mais legível, existe a técnica *Test Driven Development* (TDD) criada por Kent Beck [2]. O seu objetivo é guiar a construção de código simples, legível, que funcione conforme a necessidade de utilização do sistema de software, descrito pelo cliente, e esteja integrado à documentação do *design* [3].

A utilização de TDD para a construção de sistemas de software teve início entre os anos 2002 e 2003 e ainda não está consolidada. Como uma de suas principais preocupações é obter código simples de manter [1], a sua manutenção tende a ser menos árdua. Outra preocupação é desenvolver o software com a funcionalidade que o cliente solicitou [1, 2]. Atender as solicitações de seus clientes é um desafio que a indústria de software enfrenta, pois muitas vezes os conceitos perdem-se no meio dos processos [2, 14]. O desenvolvimento de software utilizando TDD sugere começar a escrever casos de testes conforme a descrição do software e, posteriormente, refatorar o código gerado para atingir a funcionalidade desejada na sua completude. O teste é feito antes do desenvolvimento da lógica real a ser implementada.

Neste artigo, o objetivo é apresentar o resultado da utilização de TDD no desenvolvimento de um sistema de software similar a um existente - **Software Simulador de Bolsa de Valores**. Espera-se obter, em relação à versão legada, código 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 desenvolvimento.

Este artigo está estruturado da seguinte maneira. A técnica TDD é brevemente apresentada na Seção II. Avaliação da manutenibilidade do sistema de software desenvolvido utilizando TDD é discutida na Seção III. Trabalhos relacionados são citados na Seção IV. Conclusões, contribuições e sugestões de trabalhos futuros são discutidas na Seção V.

II. *TEST DRIVEN DEVELOPMENT*

TDD, conhecida como *test-first* [10], foi criada no escopo do *eXtreme Programming* (XP) [18]. O seu objetivo é proporcionar ao desenvolvedor uma maneira de desenvolver sistemas de software que resulte em código com legibilidade e fácil de manter. Seu princípio é o desenvolvedor escrever testes correspondentes a uma função especificada nos casos de uso e guiar-se por esse teste para programá-la [2]. Diferentemente do desenvolvimento de sistemas de software tradicionais, o desenvolvedor tem informações a respeito do comportamento de uma função implementada instantaneamente, logo que a desenvolve, sendo um processo dinâmico [1]. Em TDD, o

978-1-4673-9143-6/15/$31.00

desenvolvedor escreve o código à medida que avança no desenvolvimento dos testes e, no decorrer do processo, são tomadas decisões referentes à arquitetura interna do software, modificando o código com frequência para tornar seu *design* legível, simples e funcional [10, 20]. Essa técnica pode ser vista como um conjunto de iterações para completar uma tarefa [2], pois, quando um teste é escrito, o desenvolvedor realiza modificações no código até o teste ter sucesso, caracterizando iterações até alcançar o comportamento desejado. Após essas iterações e o sucesso do teste, a tarefa é completada e o desenvolvedor inicia a implementação da próxima função.

Uma das características de TDD é guiar o desenvolvimento do software para a criação de código "testável". Pode parecer "óbvio", mas, muitas vezes, um código que parece bem feito em um diagrama pode não ser fácil de testar. Isso acontece porque, com TDD, é "exposta" (explicitada) a responsabilidade o código, antes não percebida, e a tornar mais "claras" as dependências entre as classes e os objetos.

Por isso, não é trivial atrelar a melhora na testabilidade do sistema de software apenas com a aplicação de técnicas de orientação a objetos. A utilização de TDD tende a deixar o código mais próximo de como ele será utilizado, sendo mais simples, flexível e coeso e menos acoplado. O programador percebe quando algo no código estiver "ruim", pois começará a sentir dificuldades em testá-lo.

A utilização de TDD pode promover o desenvolvimento de código dividido em pequenas interações de desenvolvimento, aumentando a coesão do código; fator determinante para aumentar a manutenibilidade e o reúso de funções [2]. Esse tipo de desenvolvimento pode parecer ao desenvolvedor que não lhe trará benefícios, mas vantagens começam a ser notadas a médio/longo prazos. Isso acontece, pois, no momento de escrever novas funções, pode-se reaproveitar o que foi escrito, além do programador saber se o código desenvolvido afeta o software negativamente [19]. Para garantir a qualidade do código, em TDD, são propostos testes a serem feitos de maneira exaustiva e em pequenos passos e o desenvolvimento com a codificação de partes mais cruciais para a função escrita naquele instante. TDD é organizado em atividades [2]:

- **Escrever Testes**. O desenvolvedor inicia a construção das funções do software pelo teste baseado nos casos de uso analisados e cada função a ser desenvolvida deve estar relacionada ao teste escrito;
- **Executar Testes e Esperar por Falhas**. Os testes escritos pelo desenvolvedor são automatizados [10, 20]. O objetivo é o desenvolvedor executar testes e esperar a ocorrência de falhas. Assim, ele tem uma direção a seguir para escrever o código funcional do software;
- **Realizar Mudanças no Código**. O desenvolvedor realiza correções em seu código para o teste ter sucesso, sem preocupar-se se a lógica está ou não correta, focando no resultado que a função deve retornar [19];
- **Executar Novamente Testes e Esperar por Sucesso**. Os testes são novamente executados e espera-se que as modificações realizadas tenham sucesso. Caso contrário, volta-se à atividade anterior para realizar novas mudanças e executar os testes até obter sucesso;
- **Refatorar o Código**. Após garantir que o sistema de software responda corretamente os testes, o desenvolvedor passa a preocupar-se com implementação mais concreta do código [1, 20]. É o momento de, por exemplo, remover duplicações de código, adequar o código ao problema e tornar o sistema de software mais dinâmico para tornar o seu código legível, limpo, robusto e simples.

III. AVALIAÇÃO DO SOFTWARE DESENVOLVIDO COM TDD

Um software legado (versão legada) - **Simulador de Bolsa de Valores** - foi escolhido para avaliar sua versão desenvolvida com TDD (versão nova). Para isso, foram utilizadas medidas de software para avaliar a qualidade interna dessas versões. Ressalta-se que a mesma funcionalidade da versão legada foi desenvolvida na versão nova.

A versão legada foi desenvolvida por um profissional com experiência de sete anos no mercado trabalho. A versão nova foi implementada por um profissional recém-chegado ao mercado de trabalho (pouca experiência). Durante o desenvolvimento da versão nova, uma das dificuldades para melhorar a versão legada foi identificação de alguns trechos de código difíceis de entender, por exemplo, identificador de variáveis não sugestivos e excesso de linhas de código em um único método. O principal responsável pela dificuldade em entender o código foi o uso demasiado do *statement* "if", o que eleva de forma significativa a complexidade ciclomática.

Mesmo com menos experiência em desenvolvimento, o profissional responsável por implementar a nova versão, utilizando TDD, melhorou significativamente a qualidade do código, reduzindo a quantidade de defeitos e diminuindo a complexidade ciclomática.

A. Sistema de Software: Simulador de Bolsa de Valores

O sistema de software **Simulador de Bolsa de Valores** é um ambiente de compra e de venda de ações em bolsas de valores. O conteúdo das notícias da Bovespa é armazenado em um repositório de dados e utilizado para analisar o "humor" do mercado. Com esse conteúdo, são realizadas previsões sobre o comportamento do mercado financeiro utilizando inteligência artificial (lógica *fuzzy*) [11]. Esse software possui 10 funções:

- Filtrar Intraday;
- Filtrar Notícias;
- Emitir Ordem de Compra/Venda;
- Visualizar Portfólio;
- Visualizar Intraday;
- Visualizar Gráfico Comparativo;
- Cadastrar Usuário;
- Autenticar Usuário;
- Visualizar Custódia;

- Visualizar Extrato.

B. Tecnologias Utilizadas

Foi utilizada a linguagem de programação Java com o auxílio de três *frameworks*:

- **vRaptor**. Age no controle dos componentes do software e na interação entre os módulos, sendo responsável pela construção de *servlets* que recebem requisições HTTP (*Hypertext Transfer Protocol*) dos usuários;
- **Hibernate**. Abstrai o acesso ao banco de dados e gera consultas nas tabelas conforme o serviço requisitado pelo usuário ou por uma requisição interna do software. Além disso, ele é o responsável por estabelecer a conexão com o banco de dados e controlar a quantidade de conexões permitidas em determinado momento;
- **jUnit**. Realiza testes automatizados.

O **Google CodePro Analytix** é um *plug-in* Java para o ambiente de desenvolvimento IDE Eclipse e é utilizado para analisar características do código de um sistema de software. Algumas de suas funções apresentam análises em formato de gráficos e de tabelas e gera relatórios em HTML (*HyperText Markup Language*). Esse *plug-in* foi utilizado para calcular o valor das medidas do código das duas versões do sistema de software analisado.

C. Medidas de Software Utilizadas

Cinco medidas de software foram selecionadas para realizar a avaliação do código das duas versões do sistema de software:

- ***Average Cyclomatic Complexity***. É a média da complexidade ciclomática de cada método definido nos elementos de destino. A complexidade ciclomática de um método é uma medida da quantidade de caminhos distintos de execução no âmbito do método. Ele é medido com a soma de um caminho para o método com cada um dos caminhos criados pelas declarações condicionais [8];
- ***Weighted per Methods***. É a soma da complexidade ciclomática dos métodos de uma classe [7, 9]. O resultado representa os níveis de responsabilidade do objeto em relação ao sistema de software, verificando a quantidade de tarefas que seus métodos executam. Quanto maior o valor dessa medida, mais complexo é o código do sistema de software e maior é a responsabilidade da classe (componente/módulo) [7];
- ***Average Lines Of Code Per Method***. É a média da quantidade de linhas de código (*Lines of Code* - LOC) em cada um dos métodos definidos nos elementos de destino [8]. Sua análise indica se os métodos são longos, o que aumenta a probabilidade deles possuírem mais complexidade e responsabilidades do que deveriam [2, 5];
- ***Afferent Couplings***. Representa a quantidade de classes externas de um pacote (componente/módulo) que dependem de classes pertencentes a esse pacote (componente/módulo) [13]. Quanto maior é o valor dessa medida, maior pode ser considerado o reúso de código [9];
- ***Efferent Couplings***. Representa a quantidade de classes em um pacote (componente/módulo) que utilizam outras classes fora desse pacote (componente/módulo) [13]. Quanto maior é o valor dessa medida, menor é o reúso de código, consequentemente, menos árdua e demorada é a manutenção [9, 13, 21].

D. Características do Código

Há muitas classes na versão nova pelo fato de cada uma ter sido escrita conforme as funções eram desenvolvidas. Em algumas situações, eram identificadas necessidades específicas dos componentes para serem transformadas em classes, contribuindo para esse aumento e para as responsabilidades das classes ficarem bem divididas e localizadas. Além dessa divisão, 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 e tornaram-se 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. Assim, foi utilizado o conceito **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 *classeA*, observou-se a necessidade de utilizar um método da *classeB*; logo, a *classeA* recebe um objeto da *classeB* como parâmetro em seu construtor. Não compete ao objeto da *classeA* controlar a instância do objeto da *classeB*, apenas recebe como argumento no construtor, caracterizando a dependência e o reaproveitamento do código.

O desenvolvimento focado na interface das funções, não na implementação propriamente dita, proporcionou a criação de objetos mais genéricos, o que evitou a escrita de código repetitivo. O desenvolvimento foi pensado no comportamento que uma classe deveria ter, sem pensar em como elas seriam implementadas, o que caracterizou um nível razoável de polimorfismo, em que muitas classes não tinham seu uso explícito, utilizando sua instância direta. Dessa forma, a capacidade de mudanças do código tornou-se perceptível e a flexibilidade com a utilização 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 comportamento.

Para auxiliar na distribuição desses componentes nas classes dependentes e tornar o polimorfismo "mais sólido", foi utilizado o padrão de projetos (*design pattern*) *Factory* [6]. Conforme o tipo do objeto que utiliza as dependências, esse padrão de projeto forneceu o melhor objeto para aquele tipo de requisição. Assim, não houve a necessidade da classe dependente ter “conhecimento” explicitamente de qual recurso ela utiliza. Dessa forma, foi alcançado razoável nível de abstração com código legível, pois apenas o nome das

ações/métodos foi listado nas classes, não havendo preocupação com a implementação dessas ações/métodos. O padrão de projetos *Factory* forneceu em tempo de execução o tipo de serviço a ser tratado. O controlador apenas necessitou 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.

Durante o desenvolvimento da versão nova, foram identificadas semelhanças entre classes existentes; em seguida, generalizações foram feitas de maneira natural, sem planejamento anterior, tornando um processo intuitivo. Após a identificação dessas semelhanças e redundâncias, foi realizada refatoração para eliminar essas situações. Na refatoração, as manutenções foram executadas e pôde-se perceber as dificuldades em modificar o código nesse momento do desenvolvimento. Durante a programação, esses problemas são difíceis de identificar em um planejamento feito antes do desenvolvimento, contudo, com a utilização de TDD, eles foram abordados de maneira pontual, sendo corrigidos.

Pôde-se perceber que cada teste foi realizado em uma única função. A construção do *design* das classes foi dirigida às funções e cada método possuía responsabilidades curtas e limitadas, não fugindo do seu escopo. Vale ressaltar que cada teste foi programado individualmente e não foi escrito outro teste antes que o corrente obtivesse sucesso nas verificações e que o código tenha sido refatorado de acordo com a percepção do desenvolvedor. Nesse ponto do desenvolvimento, notou-se que cada método foi construído para fazer somente o que se esperava dele, sem "misturar" responsabilidade. Porém, em alguns momentos, foram identificados métodos que poderiam ser mudados de classe, sendo um dos benefícios da refatoração de código.

Com TDD, foi possível reutilizar código durante a programação e ficou latente a caracterização da utilização de polimorfismo. A característica mais marcante foi classes de níveis mais altos visualizarem apenas níveis intermediários, preocupando-se apenas em que ação deveria chamar. Em momento algum houve a necessidade de conhecimento de detalhes técnicos do software, tornando as modificações menos árduas do que o comum. No entanto, desenvolvedores experientes podem alcançar esse nível de abstração em decorrência do conhecimento em desenvolver sistemas de software e por prever esse tipo de situação.

E. Discussão dos Resultados

Um dos gráficos construídos com o *Google CodePro Analytix* é o grafo de dependência que representa as dependências do código. Um grafo de dependência de primeiro nível da versão nova em relação aos *frameworks* utilizados e às bibliotecas de apoio é apresentado na Fig. 1. Nessa figura, são apresentados os componentes que o componente *tddsimuladorhomebroker* depende.

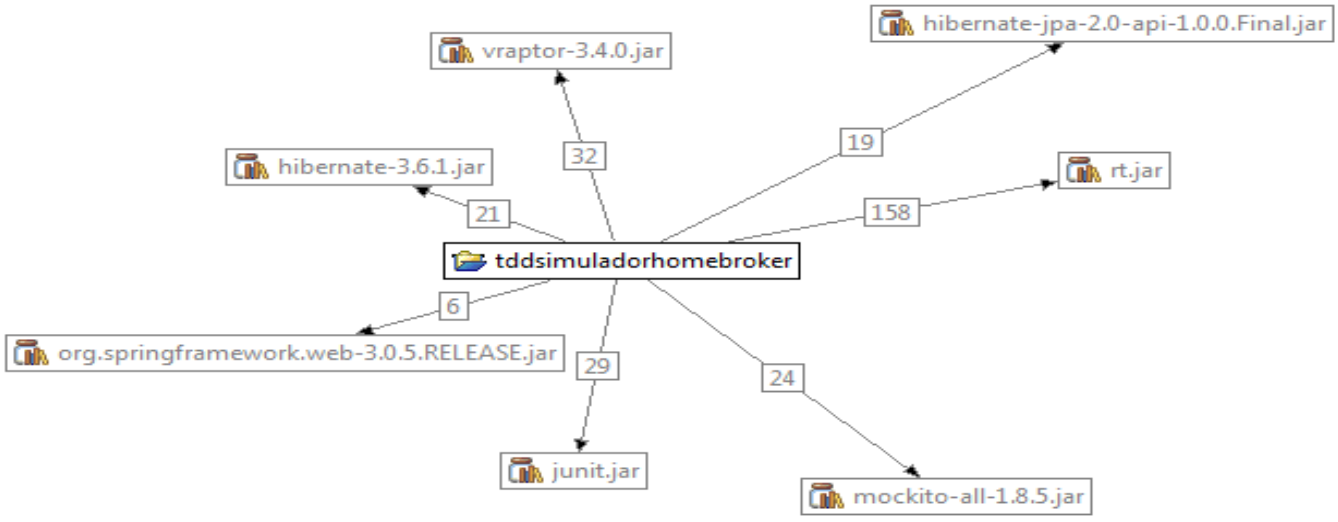

Fig. 1. Gráfico de Dependência

O valor das medidas utilizadas para avaliar as duas versões é apresentado na Table I. Pode-se perceber que esses valores diminuíram para a versão nova. O valor da medida *Weighted per Methods* diminuiu ~47% da versão legada para versão nova. Analogamente, o valor da medida *Average Cyclomatic Complexity* diminuiu ~24% da versão legada para versão nova. Os gráficos referentes a essas medidas são apresentados na Fig. 2 (*Weighted per Methods*, *Efferent Couplings* e *Afferent Couplings*) e na Fig. 3 (*Average Lines Of Code Per Method* e *Average Cyclomatic Complexity*).

TABLE I. RESULTADO DAS MEDIDAS

Código	*Weighted per Methods*		*Efferent Couplings*		*Afferent Couplings*		*Average Lines of Code per Method*		*Average Cyclomatic Complexity*	
Sem TDD	573	47%▼	48	58%▼	20	10%▼	6,13	22%▼	1,65	24%▼
Com TDD	305		28		18		4,79		1,26	

Na primeira comparação (Fig. 2), são confrontados os valores obtidos das medidas *Weighted per Methods*, *Efferent Couplings* e *Afferent Couplings* e pode-se perceber que o código da versão nova possui vantagem em relação à versão legada. Isso ocorre pelo uso de TDD, quando pequenas funções são implementadas uma por vez, o que acarreta métodos menores, com apenas uma função considerada em cada escopo de teste. Dessa maneira, o código é direcionado a ter mais métodos com pesos menores (menos responsabilidades). No código da versão legada, apesar de ter razoável grau de abstração, os métodos possuem mais pesos, como é mostrado na medida *Weighted per Methods*. Não se pode dizer que foi uma falha do desenvolvedor, mas, quando se desenvolve sistemas de software sem a prática de dividi-lo em pequenas iterações e funcionalidade, 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 refatoração, é sugerido extrair trechos de um código com responsabilidade extra. Em TDD, o desenvolvedor deve realizar a refatoração no código; as chances de remover redundâncias e diminuir responsabilidades nos métodos são maiores.

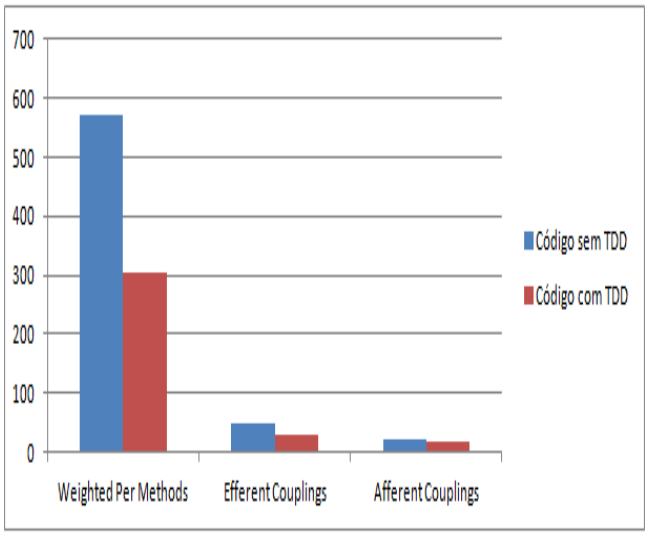

Fig. 2. Valor das Medidas *Weighted per Methods*, *Efferent Couplings* e *Afferent Couplings*

A diferença entre métodos designados a realizarem a mesma tarefa é apresentada na Fig. 4 e na Fig. 5. Na primeira, encontra-se um método da versão legada e, na segunda, encontra-se um trecho de código designado à mesma tarefa, mas da versão nova. Pode-se perceber o aglomerado de código apresentado na Fig. 4. Porém, mesmo sem utilizar TDD, o desenvolvedor poderia utilizar refatoração para facilitar a leitura do código. Como em TDD o desenvolvedor é "guiado" para realizar a refatoração, essa prática é retratada no código.

Enquanto no código apresentado na Fig. 4 as tarefas estão agrupadas dentro de um laço de repetição, no código apresentado na Fig. 5, cada tarefa é um método, por exemplo, *getHeadline(...)* e *getContentText(...)*. Além disso, pode-se perceber acoplamento indesejado no código da Fig. 4. A classe em questão é concreta e busca notícias de uma fonte, fato observado na chamada do método *pesquisarNoticiaFromFOLHA(...)*. No código da Fig. 5, 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 apresentado faz parte de uma subclasse do serviço de buscar notícias, sendo a classe abstrata, pois o controlador que chamar esse serviço não tem conhecimento de sua implementação. Ele terá conhecimento que há retorno de uma lista de notícias; logo, a manutenção pode tornar-se mais simples, não precisando de tratamento para verificar a fonte da notícia.

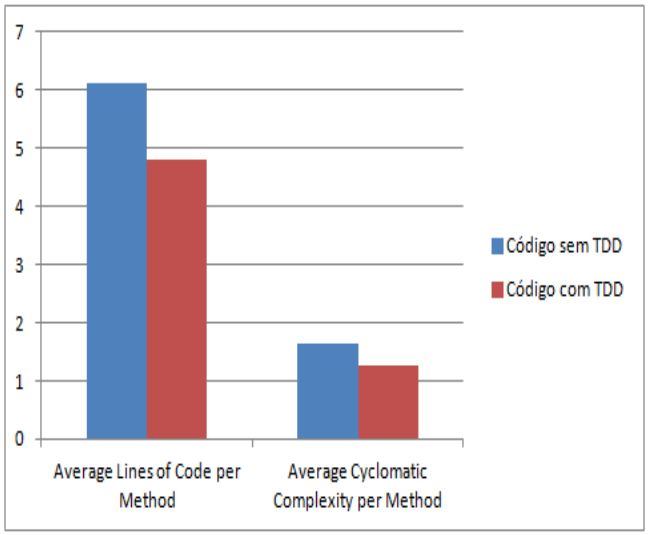

Fig. 3. Valor das Medidas *Average Cyclomatic Complexity* e *Average Lines of Code per Methods*

A segunda medida apresentada na Fig. 2 é *Efferent Couplings*, cujo valor mostra maior dependência de classes externas no código da versão legada. Essa medida indica dependência entre as classes de escopos distintos (outros pacotes/módulos). Na versão legada, a quantidade de abstrações foi menor, havendo maior redundância no código e vários componentes criados com trechos de código com execução similar. Na versão nova, a quantidade de classes abstratas foi ligeiramente maior, porém contribuiu para que muitos serviços e controladores fossem genéricos, diminuindo a quantidade de classes concretas. Com essa diminuição, a quantidade de dependências diminuiu, enquanto vários componentes possuem muitas dependências no código da versão legada. Por mais que as dependências externas caracterizem grau de acoplamento, elas são necessárias em certo momento. Durante o desenvolvimento da versão nova e conforme as responsabilidades das classes identificadas, a tendência é injetar dependências nos construtores para que um objeto possa delegar a outro objeto funções que ele não deve exercer. Assim, é natural que exista algum grau de dependência entre objetos de módulos diferentes, como "indica" essa medida.

A terceira medida apresentada na Fig. 2 é *Afferent Couplings*, cujo resultado obtido não se pode concluir algo, pois o comportamento é praticamente o mesmo no código das duas versões. Os números são semelhantes, pois os serviços desenvolvidos em ambas são os mesmos. Mesmo que os módulos tenham sido reduzidos com a utilização de TDD, os serviços em formas de funções se mantiveram, pois, a funcionalidade do sistema de software não foi alterada.

O grafo de dependência da versão legada e da versão nova é apresentado na Fig. 6 e na Fig. 7, respectivamente. Nesses grafos, é mostrada a relação entre os componentes do código das duas versões, deixando visíveis as diferentes concentrações entre elas. No primeiro grafo (Fig. 6), as dependências são mais dispersas, o que fortalece o fato apresentado anteriormente da possibilidade da existência de redundância do código, pois vários objetos semelhantes podem possuir dependências semelhantes, mas são implementados separadamente. No segundo grafo (Fig. 7), a concentração de dependências é maior na versão nova, o que caracteriza a reutilização de um módulo.

As medidas *Average Lines of Code per Method* e *Average Cyclomatic Complexity* estão relacionadas às médias aritméticas entre os componentes do software. A medida *Average Lines of Code per Method* considera a média de linhas por código. No entanto, deve-se considerar que, por causa da utilização do *framework* Hibernate, há a exigência de métodos acessores (métodos *get* e *set*), cuja quantidade de linhas de código é baixa e contribuem para a redução da média do tamanho dos métodos. Essa é uma característica comum do código das duas versões; as conclusões não sofrem interferências, pois as entidades envolvidas são as mesmas. A média do tamanho dos métodos da versão legada é maior, o que reforça a afirmação da diferença de responsabilidades entre as duas versões. Média menor no tamanho dos métodos na versão nova mostra que há responsabilidade menor, executando pequenas tarefas objetivas. As responsabilidades estão mais divididas na versão nova do que na versão legada. Nesse ponto, TDD influenciou na estrutura do software. O valor da medida *Average Cyclomatic Complexity* é menor na versão nova, por causa da constante utilização de polimorfismo. O fato de existir maior abstração no código faz com que existam menos ciclos, menor quantidade de estruturas condicionais que aumentam a complexidade ciclomática do código.

```
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
            pesquisa = pesquisa.substring( pesquisa.indexOf( FIM_LINK ) + FIM_LINK.length() );
            noticia.setManchete( tratarString( pesquisa.substring( 0, pesquisa.indexOf( FIM_MANCHETE ) ) ) );

            //pegar conteudo
            pesquisa = pesquisa.substring( pesquisa.indexOf( FIM_MANCHETE ) + FIM_MANCHETE.length() );
            noticia.setConteudo( tratarString( pesquisa.substring( 0, pesquisa.indexOf( FIM_CONTEUDO ) ) ) );

            noticia.setData( getDate( noticia.getManchete() ) );

            //adicionar noticia Ã  lista
            if( noticia.getManchete() != null && noticia.getConteudo() != null && noticia.getData() != null )
                if( noticia.getManchete().toLowerCase().contains( palavraChave.toLowerCase() ) )
                    noticias.add(noticia);

        }catch( StringIndexOutOfBoundsException stringIndexOutOfBoundsException ){
            System.out.println( stringIndexOutOfBoundsException.getMessage() );
        }
    }
    return noticias;
}
```

Fig. 4. Trecho de Código da Função "Filtrar Notícias" - Versão Legada

```
@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++;
        }
    }
```

Fig. 5. Trecho de Código da Função "Filtrar Notícias" - Versão Nova

Os resultados mostraram sensível diferença entre as duas versões, os quais foram condizentes com as expectativas ao utilizar TDD. O código da versão nova tem maior modularização dos componentes e maior distribuição das responsabilidades. Uma característica interessante foi o reúso de código. Os valores das medidas *Afferent Couplings* e *Efferent Couplings* combinados aos valores observados na medida *Weighted per Methods* mostrou que a versão nova possui métodos com pouca responsabilidade e a concentração das dependências foi maior. Essa concentração sugere que um componente é utilizado por vários recursos diferentes, deixando as dependências menos espalhadas. Isso é indício de reúso de código, pois o componente tem sua funcionalidade compartilhada em vários outros objetos.

Na versão legada, existe razoável reúso de código, o grafo de dependências mostra que os componentes têm alguma dispersão entre as funções. O fato de possuir valor das medidas maior não significa que o código seja de baixa qualidade. Contudo, esse grafo mostra ganho no aspecto de reúso de código com TDD. Observando o grafo de dependências, pode-se visualizar que TDD ofereceu melhoria no *design* do código e a medida *Average Lines of Code per Method* reforça essa melhoria proporcionada por TDD ao direcionar o desenvolvimento de maneira mais sucinta.

Observando apenas o resultado da medida *Average Lines of Code per Method*, não se pode garantir que o código obteve melhor *design*. Porém, a combinação desse valor com o valor das outras medidas utilizadas na análise pode levar a conclusão que o código possui responsabilidades bem definidas nas classes e cada método realmente faz o que deve fazer, sem fugir do seu escopo. TDD guiou o desenvolvimento para que cada método pudesse executar apenas a tarefa prevista no teste. Algum comportamento que não fizesse parte do objetivo de um teste, mas necessário naquele momento, era delegado a uma dependência que a classe testada possuísse; fato que caracteriza a redução de responsabilidade de um método. Essa redução de responsabilidades aliada ao nível de abstração acarretou em um valor menor para a medida *Average Cyclomatic Complexity*. O valor das medidas utilizadas na versão nova é relativamente menor quando comparado ao valor das medidas da versão legada.

A combinação do valor das cinco medidas selecionadas leva a constatação de alguns fatos interessantes, por exemplo, para a versão nova, o código é simples, o tamanho (linhas de código) dos métodos é menor e as classes possuem responsabilidades bem definidas. Tais fatos podem caracterizar alto grau da característica de qualidade de manutenibilidade no código da versão nova. Modificar um componente que tem responsabilidade definida tende a ser

menos árduo do que realizar manutenção em código com muitas ações. As medidas mostram que os métodos são "pequenos", fator que pode contribuir com a manutenibilidade da versão nova.

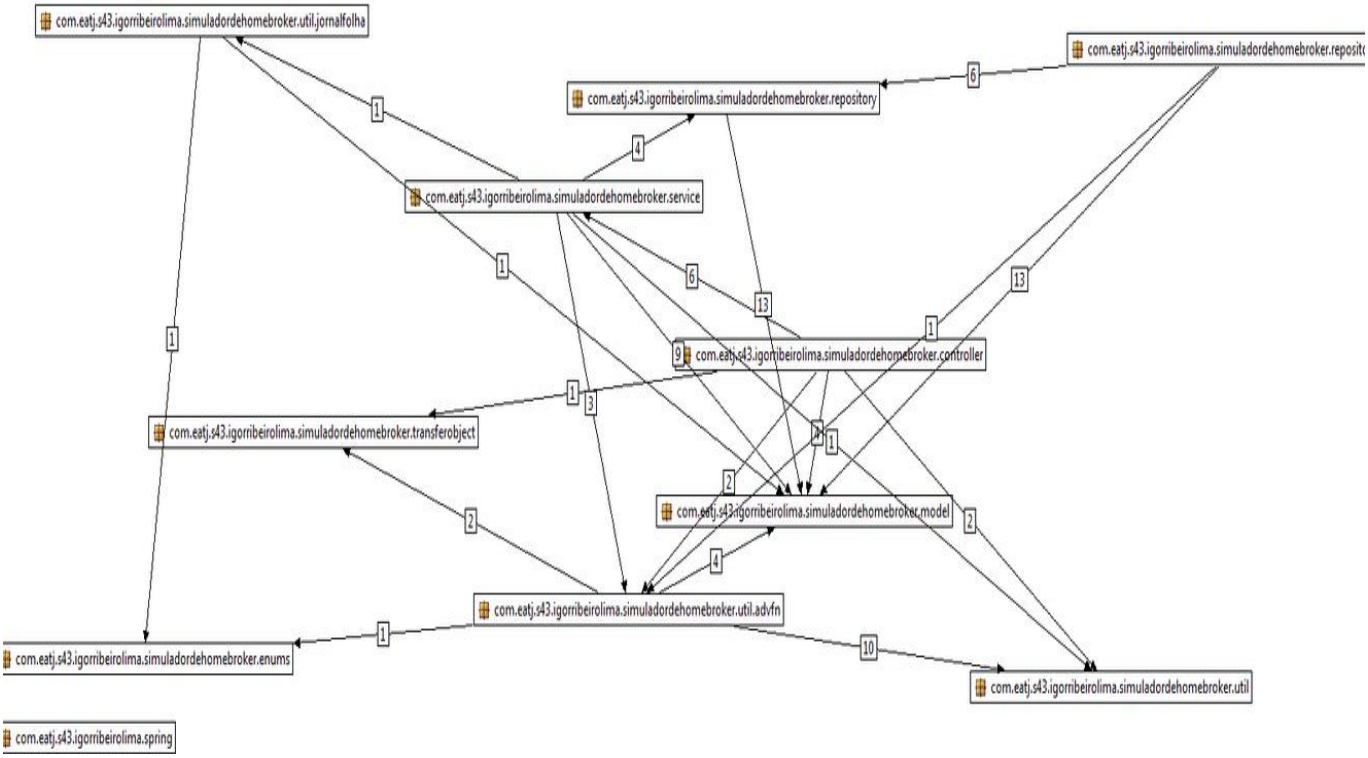

Fig. 6. Grafo de Dependências entre Componentes da Versão Legada

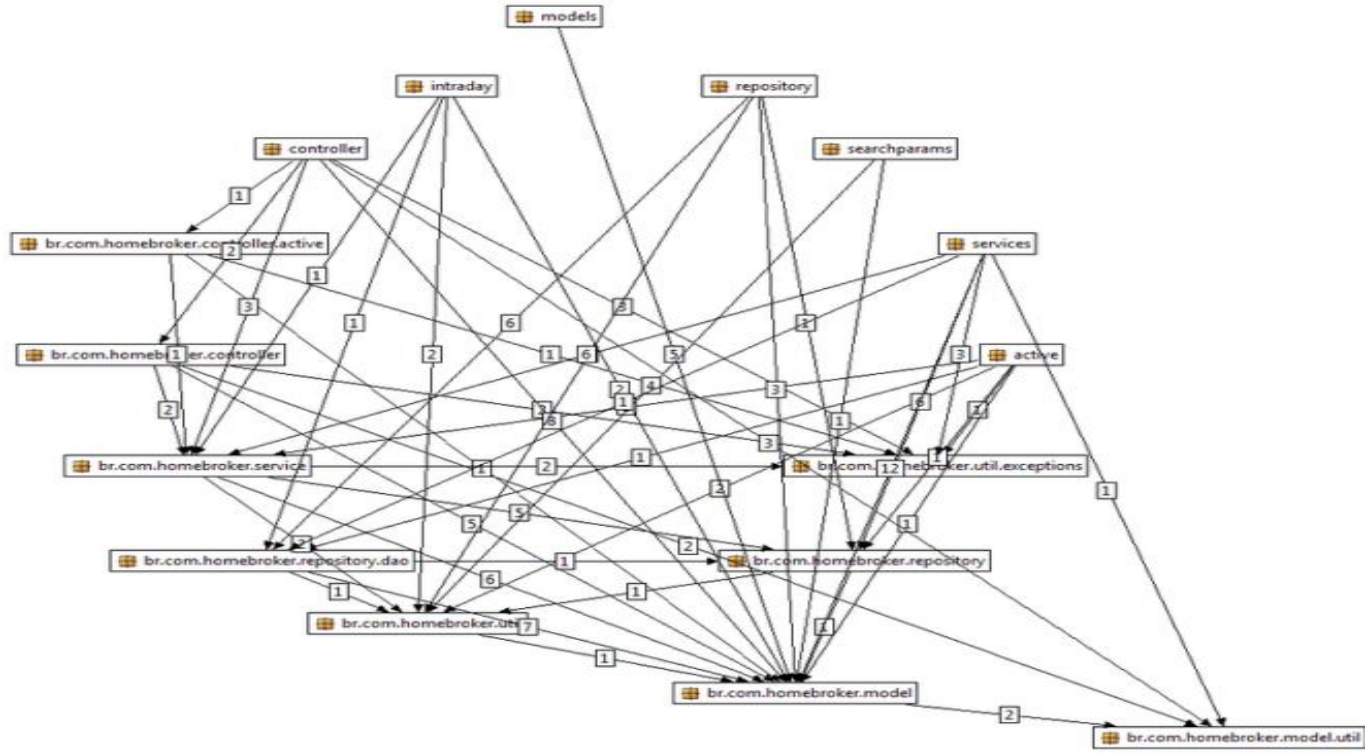

Fig. 7. Grafo de Dependências entre Componentes da Versão Nova

IV. TRABALHOS RELACIONADOS

O objetivo desse trabalho é agregar valor ao cenário da qualidade de software utilizando a técnica TDD no desenvolvimento de um software similar a um software real. Para isso, alguns fatores no código do software desenvolvido (versão nova) são observados, tais como, facilidade de modificação, de legibilidade e de compreensão. Profissionais e pesquisadores da academia e da indústria envolvidos no desenvolvimento de sistemas de software têm investido esforços para encontrar maneiras de diminuir custos financeiros e de tempo empregados na realização de atividade de manutenção desses sistemas. Nesse cenário, alguns trabalhos podem ser encontrados na literatura e são brevemente relatados a seguir.

Em um destes trabalhos [9], foi apresentado um estudo com alunos de graduação e de pós-graduação organizados em duas equipes. Esse estudo consistiu no desenvolvimento de um software utilizando 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 TDD melhora o *design* do código, realizando comparações dos resultados obtidos em ambas as abordagens. As equipes desenvolveram o mesmo sistema de software sem terem interação. Ao finalizar o experimento, aferições foram feitas utilizando medidas para analisar o código, por exemplo, complexidade ciclomática, quantidade 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 e seu acoplamento impactam na qualidade interna. A conclusão foi que TDD permite a construção de código mais coeso 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 e, consequentemente, facilidade de compreensão. O sistema de software foi construído com classes menores em relação às classes das equipes que utilizaram *test-after* e com mais capacidade de escrever testes para suas funções, ou seja, o sistema de software possuía mais possibilidades de verificação de suas funções o que garantia seu bom funcionamento.

Em outro trabalho [3], foram obtidos resultados semelhantes ao anterior. Foram utilizadas duas equipes independentes para avaliar a aplicação de TDD, uma equipe realizou testes antes (*test-first*) e outra equipe realizou testes depois (*test-after*). O conjunto de medidas utilizado foi similar ao trabalho anterior que consistiu, basicamente, na avaliação da complexidade do código fonte. A conclusão foi que TDD é uma alternativa à forma tradicional de desenvolvimento de software e favorece a construção de código manutenível. Além disso, esse trabalho afirma que o desempenho na detecção de falhas no código utilizando TDD é superior à técnica tradicional.

No terceiro trabalho [17], foram identificados outros valores relacionados a TDD. Aplicando métodos semelhantes aos trabalhos descritos anteriormente, foi mostrado que, em algumas equipes, TDD foi responsável por queda na produtividade, considerando a entrega das funções ao comparar as equipes que utilizaram a técnica tradicional de testes automatizados. O contraponto é TDD depender da capacidade e da preferência da equipe que a utiliza. Nesse estudo, as equipes não desenvolviam em apenas um estilo, ocorrendo revezamento. Foi possível observar que algumas equipes utilizaram TDD 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 [2]. Dessa forma, TDD sofre influência da equipe e o rendimento das pessoas envolvidas pode variar conforme seu perfil, o que pode dificultar a manutenção da simplicidade de código em relação ao obtido por meio de desenvolvimento tradicional.

Além da característica de qualidade manutenibilidade, existem melhorias relacionadas à comunicação entre os membros da equipe, como mostra o quarto trabalho analisado [4]. Esse trabalho reforça os resultados apresentados no primeiro trabalho, argumentando os benefícios de iniciar o desenvolvimento de sistemas de software por testes. O desenvolvedor é forçado a pensar mais sobre questões da funcionalidade desenvolvida antes de codificá-la, tornando o código mais objetivo. Essa abordagem gera 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 mais capacidade dos desenvolvedores em realizar tarefas de manutenção com menos lentidão. Além disso, a comunicação entre esses desenvolvedores tornou-se eficiente. Enquanto eles escreviam uma parte do código e, mais tarde, realizavam refatoração, havia troca intensa de informações, pois um desenvolvedor procurava saber o significado de um teste escrito por outro. Essa troca proporcionava mais agilidade na hora de entender os componentes a serem modificados ou complementados e melhorava a manutenibilidade do sistema de software. O trabalho mostrou que, além de mais facilidade na leitura do código, TDD influenciou outros níveis do desenvolvimento, por exemplo, troca de informações na equipe, o que contribuiu para mais produtividade no momento de realizar modificações e atualizações em um sistema.

Além desses fatores relacionados à legibilidade do código, pode-se pensar na capacidade em encontrar problemas no sistema de software [12]. Como apresentado em outro trabalho que argumenta melhor capacidade de verificar uma falha em um sistema de software. Observou-se que os desenvolvedores que utilizavam TDD realizavam menos *debug* no código, 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 de TDD pelo 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 de TDD para que esses benefícios sejam alcançados. Ao seguir esse ciclo, há forte tendência do desenvolvedor entregar funções com pequena quantidade de *bugs*, o que proporciona agilidade na hora de realizar manutenções.

Em outro trabalho [16], é analisada a simplicidade do código gerado com a utilização de TDD. Quando equipes na empresa Nokia Networks aplicaram essa técnica no desenvolvimento de sistemas de software, foi identificado um código simples (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. 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 e conseguia identificar redundâncias e aspectos que deixavam o código de difícil leitura. Durante o desenvolvimento, as melhorias aconteciam no momento da refatoração, removendo pequenas informações no código (chamadas *stuff*). Conseguiram retirar do sistema de software trechos que contribuíam para que ele se tornasse mais rebuscado. O *design* foi pensado e reformulado passo a passo, removendo os *stuffs*, e alcançaram código objetivo e simples.

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

V. Considerações Finais

Utilizar TDD para desenvolver sistemas de software não é trivial. Seguir os passos corretos para obter seus benefícios exige disciplina, pois altera a forma de desenvolver esses sistemas. Essa alteração no estilo de programar é o principal desafio no início da programação, mas seus benefícios são rapidamente percebidos. A cada passo durante a construção do código, pode-se visualizar e perceber o sistema de software assumindo *design* enxuto e objetivo, com código legível e fácil de entender. Essas percepções foram confirmadas ao fim do desenvolvimento do software (versão nova) utilizado como estudo de caso. Comparando o código das duas versões, foi observada melhor legibilidade no código da versão nova (utilizando TDD), melhorando sua manuteniblidade. O desenvolvimento com TDD proporcionou código com maior nível abstração, o qual foi ponto inicial para que outras vantagens pudessem ser alcançadas. Essa abstração fez com que trechos de código com lógica mais complexa ficassem encapsulados e escondidos de trechos de código que os utilizam. Se um componente não conhece a complexidade de outro componente em que ele depende, o código torna-se mais flexível por não existir a preocupação de sabe o "como" ele realiza sua função, mas em saber "qual" é a função.

Com base nos resultados coletados, o desenvolvedor pode optar por escolher TDD visando à construção de código 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 de sistemas de software e as manutenções podem ser feitas de maneira menos árdua pelas pessoas envolvidas na continuação desses sistemas. Como consequência dessa análise, foi realizada a reengenharia de um sistema de software existente, porém apenas no código, para identificar a funcionalidade da versão legada, pois sua documentação não existia. Com a funcionalidade identificada, pode-se elaborar os testes para o desenvolvimento da versão nova do software utilizando TDD.

Como sugestões de futuros trabalhos, pode-se utilizar TDD em um ambiente com mais desenvolvedores, analisar reúso de código alcançado na equipe com a utilização de componentes desenvolvidos separadamente, avaliar a capacidade da equipe em entregar sistemas de software com menor quantidade de falhas e utilizar outras medidas de software.

Referências

[1] Astels, D. Test-Driven Development: A Pratical Guide, 2003. 592 p.

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

[3] Canfora, G.; Visaggio, C. A. Measuring the Impact f Testing on Code Structure in Test Driven Development: Metrics and Empirical Analysis. In: International Symposium on Emerging Trends in Software Metrics, 2009.

[4] Crispin, L. Driving Software Quality: How Test-Driven Development Impacts Software Quality. IEEE Software, p. 70-71, 2006.

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

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

[7] 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.

[8] Java Developer Tools. Localizado em: <https://developers.google.com/java-dev-tools/codepro/doc/features/metrics/metrics_details>. Acessado em: 03/2015.

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

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

[11] 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, 2010. p. 455-463.

[12] Martin, R. C. Professionalism and Test-Driven Development. IEEE Software, p. 32-36, 2007.

[13] Meirelles, P. R. M. Levantamento de Métricas de Avaliação de Projetos de Software Livre. 78 p. 2008.

[14] Pressman, R. S. Software Engineering - A Practitioner's Approach. 860 p. 2010.

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

[16] Vodde, B.; Koskela, L. Learning Test-Driven Development by Counting Lines. IEEE Software, p. 74-79, 2007.

[17] 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.

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

[19] Wirfs-Brock, R. J. Design For Test. IEEE Software. p. 92-93, 2009.

[20] Shull, F.; Melnik, G.; Turhan, B.; Layman, L.; Diep, M.; Erdogmus, H. What do We Know About Test-Driven Development? In: IEEE Software. pp. 16-19. 2010.

[21] Kastenberg, H. Software Metrics as Class Graph Properties. Master's thesis, University of Twente. 131p. 2004.