ISSN 2316-2872
T.I.S. São Carlos, v. 4, n. 3, p. 260-267, set-dez 2015

Requisitos testáveis com *behaviour-driven development*

João Antonio Bulgareli, Ivan João Foschini

Resumo. Dentre as metodologias de desenvolvimento ágil, a Extreme Programming (XP) foi pioneira em incentivar a utilização de testes automatizados. Esse incentivo contribuiu para a criação do Test-DrivenDevelopment (TDD), um modelo de desenvolvimento no qual o teste é escrito antes do código. O desenvolvimento orientado a comportamentos (Behaviour-DrivenDevelopment - BDD) surgiu como uma evolução do TDD, com foco no comportamento do sistema, porém não deixando de lado as técnicas utilizadas pelo TDD. O BDD contribui com o desenvolvimento de software baseado nos requisitos previamente identificados através de histórias de usuário. Este trabalho apresenta as principais características do BDD, a sua relação com os valores ágeis e a aplicação prática da técnica com auxílio do framework SpecFlow, que irá auxiliar na transformação dos requisitos em testes.

Palavras-chave: Desenvolvimento orientado a comportamentos; Desenvolvimento ágil de software; Desenvolvimento orientado a testes;

Stable requirements with behavior-driven development

***Abstract**. Among the methodologies of agile development, Extreme Programming (XP) was the first to encourage the use of automated testing. This incentive has contributed to the creation of Test-Driven Development (TDD), a development model in which the test is written before the code. The Behavior-Driven Development (BDD) has emerged as an evolution of the TDD, focusing on system behavior, but not leaving behind the techniques used by TDD. BDD contributes to the software development based on the requirements previously identified through user´s stories. This paper presents BDD main features, its relationship with agile values and the practical application of the technique with the help of SpecFlow framework that will assist in transforming requirements in tests.*

***Keywords**: Behavior-driven development; Software Agile development; Test-driven development*

I. Introdução

Com o aumento da demanda de software, existe cada vez mais necessidade da entrega de produto de forma rápida e com qualidade (COSTA FILHO, 2006). O método típico para alcançar esses objetivos é o desenvolvimento de software ágil. Extreme Programming (XP) é uma das metodologias de desenvolvimento ágil, que como as demais, auxiliam no processo de desenvolvimento de software com requisitos superficiais e em constante mudança (BECK, 1999). Um dos princípios dessa metodologia é trabalho de alta qualidade, que é sintetizado através do desenvolvimento orientado a testes, do inglês Test-Driven Development (TDD).

O TDD, por sua parte, sugere que os testes de entrada e saída de valores do sistema, mais conhecidos como testes unitários, sejam escritos antes mesmo da codificação da funcionalidade. Desta forma, busca garantir que mais partes do sistema estejam com a cobertura de um teste e maior qualidade de escrita do software. Sua aplicação é simples, inicialmente se escreve o teste unitário sem a existência da lógica que o teste irá validar, apenas com o necessário para que o projeto possa ser compilado. Logo após, o teste é executado observando sua falha. Realiza-se a codificação para que o teste seja válido e, finalmente, o desenvolvedor busca identificar possibilidades de melhoria técnica no código (BECK, 2001). Nesse contexto, NORTH (2003) dá início a criação do desenvolvimento orientado a comportamentos, tradução do inglês de Behaviour-Driven Development (BDD), devido seu descontentamento com mal-entendidos do TDD. Quando se testa a estrutura interna de um objeto, o teste é do objeto ao invés do comportamento que deveria ter. Diferente disso, o BDD foca sobre o comportamento em geral do software, ou seja, objetos, serviços externos e até mesmo uma interface gráfica (CHELIMSKY et al., 2010).

É com estas características que se inicia a preocupação entre indivíduos envolvidos, os sistemas e o comportamento de objetos mais que a estrutura em que é escrita. A tríade do

Departamento de Computação - Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – 13.565-905 – São Carlos – SP – Brasil
Autor para correspondência: jabulgareli@gmail.com, ivan@dc.ufscar.br

BDD é o "Given, When and Then" ou em português "Dado, Quando e Então", essas palavras são utilizadas para descrever funcionalidades. É comum serem descritas como histórias de usuário em texto plano, que posteriormente serão convertidas em testes. Esses conceitos serão detalhados nas próximas seções do artigo.

Toma-se como objetivo deste trabalho o estudo prático da aplicação do BDD no desenvolvimento de funcionalidades de uma aplicação, a sua influência na criação de projetos, a relação com os valores ágeis e qualidade de código que resulta da sua aplicação.

Para este trabalho alcançar seus objetivos as metodologias a seguir serão adotadas: a) Desenvolver funcionalidades de um aplicativo utilizando BDD, TDD e o framework SpeckFlow na linguagem de programação C# com o Microsoft Visual Studio 2013 Community; b) Analisar como o BDD influencia na criação das funcionalidades em questões de qualidade de software; e c) Identificar sua influência na aplicação desenvolvida e relações com os valores ágeis.

II. Desenvolvimento orientado a comportamentos

Os métodos tradicionais de desenvolvimento costumam falhar devido ao planejamento precoce da aplicação, que resulta em entrega atrasada ou fora do orçamento, entregar um produto diferente do que era esperado pelo cliente, instabilidade em produção e alto custo de manutenção (CHELIMSKY et al., 2010).

Em contrapartida, a metodologia de desenvolvimento ágil busca amenizar os efeitos colaterais da utilização dos métodos tradicionais. Aplica maior ênfase na colaboração entre a equipe de desenvolvimento e o grupo de especialistas do negócio com resultado na melhoria da comunicação entre os envolvidos. Realiza ciclos de desenvolvimento pequenos com o intuito de entregar para o cliente funcionalidades que agregam valor ao seu negócio de forma frequente. As equipes ficam mais próximas e buscam se organizar de forma independente para resolver seus impedimentos, se tornando assim uma metodologia de trabalho no qual as inevitáveis mudanças dos requisitos não gerem uma crise na equipe. Esta metodologia é fundamentada por uma declaração denominada manifesto ágil que descreve através de seus valores e princípios, como qualquer método deve basicamente funcionar. (BECK et al., 2001).

Os quatro valores do manifesto ágil, são:

- Os indivíduos e suas interações acima de procedimentos e ferramentas;
- O funcionamento do software acima de documentação abrangente;
- A colaboração dos clientes acima da negociação de contratos;
- A capacidade de resposta à mudanças acima de um plano preestabelecido.

O TDD é um dos princípios da metodologia ágil XP, que foi pioneira no incentivo a automatização de testes. O TDD consiste, basicamente, na realização de um pequeno ciclo denominado vermelho-verde-refatora, que pode ser observado na figura abaixo:

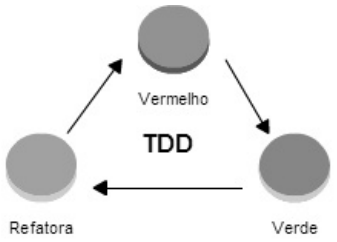

Figura 1. O ciclo vermelho-verde-refatora do TDD

No primeiro passo para se praticar o TDD, o círculo vermelho descrito na figura 1 se refere ao ato de escrever o teste para a funcionalidade desejada mesmo que a lógica não esteja codificada, para que haja a confirmação que o teste não será validado. O próximo passo, representado pelo círculo verde, o código para que o teste seja válido deve ser escrito da forma mais simples possível para observar a confirmação que o teste foi validado. E finalmente a terceira fase que é a refatoração, ou seja, analisar tudo que foi codificado para este teste até então, adequando aos padrões de desenvolvimento adotados pela equipe.

Segundo BECK (2003), que é considerado o criador do TDD, a técnica auxilia na confiança do código produzido junto com a melhoria no design do código criado, visto que escrever o código para que o teste não seja válido de imediato pode deixar o código mais claro e sem poluição quanto aos outros métodos de desenvolvimento.

O BDD surgiu na dificuldade que seu idealizador teve em ensinar TDD para seus alunos. NORTH (2003) afirma que seus alunos tinham dúvidas relacionadas a o que testar, quando testar, o que significa quando um teste falha, ou seja, dúvidas relacionadas ao teste em si, não levando em consideração a aplicação da técnica e os benefícios que ela traz. Desta forma, North teve a ideia de aplicar o TDD voltado ao negócio, com foco na produção de funcionalidades relevantes do produto a ser desenvolvido, nos valores do desenvolvimento ágil de software e remover a palavra teste, para que não exista confusão.

Um dos princípios do BDD é que a escrita dos nomes dos métodos sejam frases completas, com o máximo de informação possível de seu funcionamento. North observou esse comportamento com a utilização do aplicativo "Agiledox", criado por STEVENSON (2003). O aplicativo converte o nome dos testes unitários codificados com o auxílio do framework JUnit, os escrevendo em texto, removendo a palavra "test" caso haja. Na figura 2 abaixo, é demostrado um exemplo de classe de teste que utiliza os nomes dos métodos como o BDD sugere.

```
public class PedidoTest
{
    public void TestDeveConterPeloMenosUmItem()
    {
        //(..) código de teste omitido
    }

    public void TestPodeSerCanceladoDentroDoPeriodoPermitido()
    {
         //(..) código de teste omitido
    }

}
```

Figura 2. Exemplo de classe de teste com nomes completos

Com a utilização do aplicativo "Agiledox", a classe acima teria a palavra "test" removida e a documentação da classe seria gerada separando cada palavra, como por exemplo: Pedido deve conter pelo menos um "Item" ou "Pedido" pode ser cancelado dentro do período permitido.

Escrever os testes iniciando com a palavra “deve” sugere ao desenvolvedor qual o problema que será resolvido com o código que está ou será escrito, deixando-o focado nesta solução. NORTH (2003) sugere a adoção desta prática, com o intuito de criar classes e funcionalidades mais resumidas. Assim, o código fica mais simples e direto. O nome destinado ao teste é importante também quando ele falha. Um nome bem descritivo tende a ser mais fácil de identificar onde há uma possível falha no código. A figura 2, exibida acima, demonstra os testes com nomes maiores, porém bem descritivos quanto a funcionalidade que irão testar.

O BDD busca identificar o que deve ser testado na aplicação baseado no seu valor de negócio. O comportamento mais importante do software ainda não desenvolvido é sempre o próximo teste a ser criado. Para descrever estes comportamentos em forma de teste, uma linguagem única entre equipe técnica e de negócio deve ser utilizada, esta linguagem é denominada por EVANS (2003) como linguagem “ubíqua”, ou seja, todos os termos utilizados no negócio devem estar refletidos na codificação da aplicação.

Os comportamentos que serão desenvolvidos utilizando o BDD são previamente descritos utilizando o modelo de histórias de usuário. O padrão descrito por NORTH (2003) é demonstrado na figura 3, abaixo.

Como um contador
Eu quero calcular o imposto de uma nota fiscal
Para que o registro contábil esteja completo

Figura 3. Modelo de história de usuário sugerido pelo criador do BDD

Dado que a soma dos produtos de uma nota fiscal tem valor de R$ 550,90
Quando se aplica uma alíquota de ICMS no valor de 18%
Então o valor total do ICMS da nota fiscal é de R$ 99,16

Figura 4. Modelo de descrição de um critério de aceitação do BDD

Este modelo é utilizado pelo framework SpecFlow que será utilizado no exemplo prático do BDD. Com este modelo, o framework consegue identificar qual será o nome dos testes que ele irá criar automaticamente. Ele também fornece a capacidade de qualquer indivíduo relacionado com o projeto criar testes, sendo da equipe técnica ou não, pois a escrita é texto corrido, não necessitando nenhum conhecimento técnico para descrever o comportamento da aplicação e consequentemente seus testes. Desta forma, os critérios de aceitação se tornam executáveis e testáveis.

A palavra comportamento se torna mais útil que a palavra teste (NORTH, 2003). Com o foco no comportamento da aplicação, as dúvidas comuns entre os iniciantes no TDD são sanadas de forma inerente, como:

- Os testes devem ter o nome descrito de forma completa que possa refletir apenas o comportamento que será testado;
- Os principais comportamentos da aplicação devem ser testados, considerando também seus requisitos;
- O nome do teste é muito útil para sugerir onde está o problema em caso de uma falha.

III. Aplicação prática do BDD

Como exemplo prático da utilização do BDD, este trabalho descreve a solução hipotética para um requisito de um sistema contábil.

A) Preparação do ambiente

Para este trabalho foi utilizado a versão do Microsoft Visual Studio Community 2013 e o framework Specflow, os endereços e guia para instalação do ambiente podem ser encontrados abaixo:

-Microsoft Visual Studio Community 2013: https://www.visualstudio.com/en-us/news/vs2013-community-vs.aspx

-Integração do SpecFlow para o Visual Studio 2013: https://visualstudiogallery.msdn.microsoft.com/90ac3587-7466-4155-b591-2cd4cc4401bc

O código fonte completo do exemplo pode ser encontrado no endereço: https://github.com/jabulgareli/ExemploBDD

B) História de usuário para desenvolvimento

A história de usuário abaixo, descrita pela figura 5, ilustra a funcionalidade do sistema contábil onde o ator “analista do departamento do pessoal”, que é o responsável pela utilização da funcionalidade, deseja que seja desenvolvido o comportamento de cálculo do salário líquido de um funcionário.

Funcionalidade: Cálculo de salário líquido

Como um analista de departamento do pessoal
Desejo calcular o salário líquido de um funcionário
Para que o funcionário receba sua remuneração mensal

Figura 5. História de usuário para desenvolvimento

O cálculo do salário líquido consiste no valor do salário bruto deduzido o valor dos impostos: Instituto Nacional do Seguro Social (INSS) e do imposto de renda retido na fonte (IRRF).

O valor do INSS é constituído pela alíquota que o salário bruto se aplica. Para este trabalho, serão consideradas as alíquotas de INSS do ano corrente (2015) descrita na figura 6.

Salário de Contribuição (R$)	Alíquota (%)
Até 1.399,12	8
De 1.399,13 até 2.331,88	9
De 2.331,89 até 4.663,75	11

Figura 6. Alíquotas de INSS (PREVIDÊNCIA SOCIAL, 2015)

Para calcular o valor do IRRF, deve-se considerar o salário já abatido do INSS e aplicar a alíquota referente ao imposto. A parcela a deduzir deve ser subtraída do valor resultante deste

cálculo. As alíquotas de IRRF e parcela a deduzir são demonstradas na tabela 1.

Tabela 1. Alíquotas de IRRF
(GUIA TRABALHISTA, 2015)

Validade	Base de Cálculo (R$)	Alíquota (%)	Parcela a Deduzir do IR (R$)
VIGÊNCIA A PARTIR DE 01.04.2015	Até 1.903,98	-	-
	De 1.903,99 até 2.826,65	7,5	142,80
	De 2.826,66 até 3.751,05	15	354,80
	De 3.751,06 até 4.664,68	22,5	636,13
	Acima de 4.664,68	27,5	869,36

C)Criação do projeto

Um projeto do tipo "Unit Test Project" deve ser criado para este trabalho. Este projeto é utilizado para criação de testes unitários. O framework SpecFlow o utilizará como base para transformar os requisitos em testes.

O nome do projeto será "CalculoSalarioLiquidoTests". Para criar o projeto deve-se acessar o menu "File -> New Project" do Visual Studio 2013, acessar a seção "Templates -> C# -> Test -> Unit Test Project", como mostra a figura 7 abaixo:

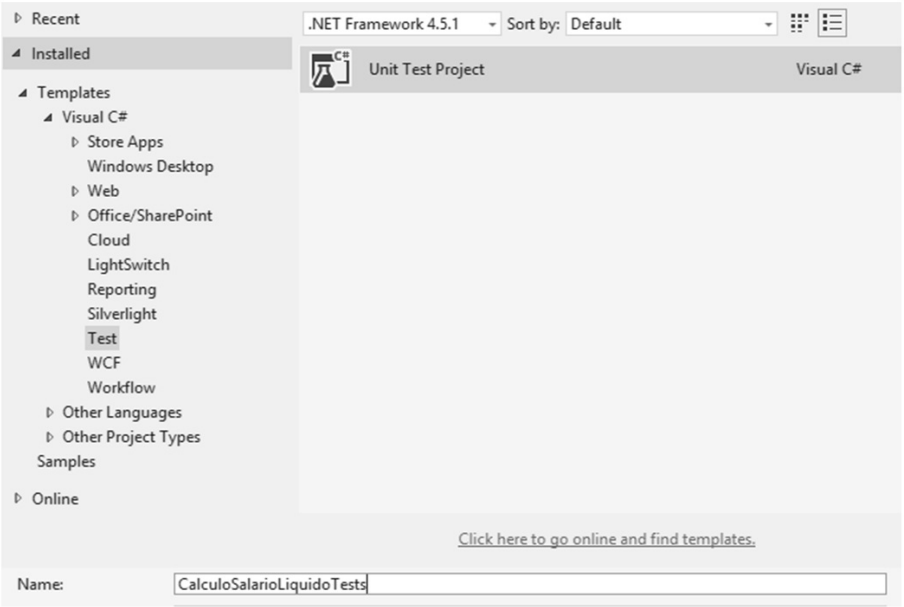

Figura 7. Criação do projeto

Neste momento, o SpecFlow já pode ser instalado no projeto. Para isso, deve-se seguir os procedimentos indicados no site do desenvolvedor do SpecFlow. As instruções podem ser encontradas neste endereço: http://www.specflow.org/getting-started/#InstallSetup.

D)Criação de cenários

A funcionalidade de cálculo e os cenários utilizando os conceitos do BDD são criados através de uma "SpecFlow feature file". Para adicioná-la no projeto, deve-se acessar com o botão direito do mouse no projeto, acessar o menu "Add -> New Item" e selecionar "SpecFlow feature file". O nome da funcionalidade deve ser "CalculoSalarioLiquido".

Um arquivo será aberto para edição da funcionalidade. Ele deve conter a definição da funcionalidade em si e os cenários que serão testados. Neste ponto é inserido texto plano, ou seja, qualquer pessoa envolvida com o projeto, sendo técnica ou não, poderá informar os dados. A funcionalidade proposta deve ser descrita conforme a figura 8 a seguir.

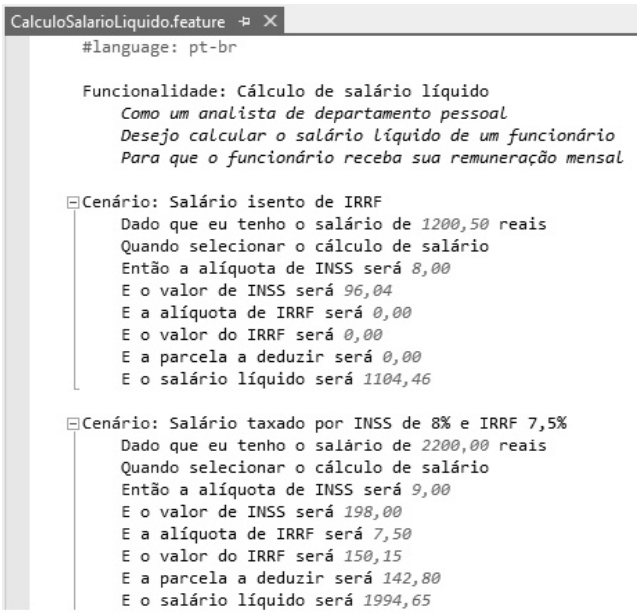

Figura 8. Funcionalidade e Cenários da funcionalidade

O Specflow permite a criação das funcionalidades e cenários em português. Para isso é necessário informar ao framework a linguagem "pt-br" conforme a primeira linha da figura 8. Logo após a funcionalidade é descrita conforme a história de usuário já demonstrada neste trabalho pela figura 5. Abaixo da funcionalidade é possível notar os cenários, que podem ser entendidos como os critérios de aceitação da funcionalidade.

Nesse ponto é possível notar a relação do BDD com um dos princípios ágeis. Os envolvidos no projeto, incluindo o cliente, podem criar os comportamentos da aplicação em texto plano, sem conhecimento técnico, aumentando assim a interação entre pessoas de negócio e a equipe técnica. A aplicação estará pronta para responder as mudanças de cenário, apenas alterando e adicionando comportamentos aos arquivos de funcionalidades. A aplicação estará apta a validar os comportamentos através dos testes unitários que serão gerados automaticamente.

A funcionalidade descrita dessa forma pode ser também utilizada como documentação do sistema, existe uma rica fonte de cenários que a funcionalidade atende e também a sua própria descrição. Como o arquivo da funcionalidade é utilizado para os testes unitários, ele deve ser atualizado de forma frequente para que os testes continuem válidos, sendo assim, a documentação se mantém sempre coerente com o que está codificado.

E) Execução dos testes

O SpecFlow permite a criação automática dos testes unitários. Para isso é necessário clicar com o botão direito em cima do arquivo da funcionalidade, neste caso "CalculoSalarioLiquido.feature", e acessar a opção "Run SpecFlow Scenarios". Os cenários não serão executados pela ausência dos testes unitários que os realizam, porém a mensagem de resultado da execução sugere o código para criação dos mesmos. A figura 9 ilustra o resultado da execução e o código que é sugerido para cada cenário.

Figura 9. Resultado da execução dos cenários sem os testes criados

O código sugerido deve ser adicionado em um arquivo do tipo "SpecFlow Step Definition". Esse arquivo é criado através do botão direito no projeto "Add -> New Item", ele deve ter o nome de "CalculoSalarioLiquidoStepDefinition". O arquivo ficará semelhante à figura 10.

```
[Given(@"que eu tenho o salário de (.*) reais")]
 public void DadoQueEuTenhoOSalarioDeReais(Decimal p0)
 {
     ScenarioContext.Current.Pending();
 }

 [When(@"selecionar o cálculo de salário")]
 public void QuandoSelecionarOCalculoDeSalario()
 {
     ScenarioContext.Current.Pending();
 }

 [Then(@"a alíquota de INSS será (.*)")]
 public void EntaoAAliquotaDeINSSSera(Decimal p0)
 {
     ScenarioContext.Current.Pending();
 }
                    (...)
```

Figura 10. Classe CalculoSalarioLiquidoStepDefinition.cs

Na figura 10 é notável a presença das palavras-chave do BDD "given-when-then" nas anotações acima de cada método responsável pelos testes da funcionalidade. Os testes não podem ser executados ainda, pois a chamada do método "ScenarioContext.Current.Pending()" está inserida em todos os métodos de teste. Este método é responsável por informar o framework de testes que a codificação ainda está ausente.

F) Codificação da funcionalidade

O funcionamento da aplicação já está descrito através da documentação da funcionalidade e seus cenários. Agora, há a necessidade de criar a codificação necessária para que o cálculo do salário líquido seja efetuado corretamente. Para isso é possível utilizar o TDD junto com o BDD, ou seja, seguir os princípios do TDD na codificação dos testes unitários gerados pelo framework a partir dos comportamentos baseados no BDD.

Considerando a utilização do TDD, deve-se criar a codificação necessária para que os testes sejam executados, porém sem a lógica do cálculo para observar sua falha. Este é o ciclo "vermelho" do TDD. A figura 11 apresenta a classe criada para cálculo somente com o necessário para ser compilada, neste caso, retorna valor zero para todos os impostos e suas alíquotas.

```
class CalculadoraSalario
{
    public void Calcular()...

    public void InformarSalario(decimal salario)...

    public decimal ObterAliquotaINSS()
    {
        return 0;
    }

    public decimal ObterValorINSS()...

    public decimal ObterAliquotaIRRF()...

    public decimal ObterValorIRRF()...

    public decimal ObterParcelaDeduzirIRRF()...

    public decimal ObterValorLiquido()...
}
```

Figura 11. Classe de cálculo do salário líquido

Esta classe é utilizada nos testes unitários gerados pelo SpecFlow. Abaixo, na figura 12, segue um trecho de sua utilização.

```
[Given(@"que eu tenho o salário de (.*) reais")]
 public void DadoQueEuTenhoOSalarioDeReais(Decimal p0)
{
    _salario = p0;
}

 [When(@"selecionar o cálculo de salário")]
 public void QuandoSelecionarOCalculoDeSalario()
 {
     _calculadoraSalario.InformarSalario(_salario);
     _calculadoraSalario.Calcular();
 }

 [Then(@"a alíquota de INSS será (.*)")]
 public void EntaoAAliquotaDeINSSSera(Decimal p0)
 {
     Assert.AreEqual(p0, _calculadoraSalario.ObterAliquotaINSS());
 }

 [Then(@"o valor de INSS será (.*)")]
 public void EntaoOValorDeINSSSera(Decimal p0)
 {
     Assert.AreEqual(p0, _calculadoraSalario.ObterValorINSS());
 }
```

Figura 12. Utilização da classe de cálculo nos testes unitários

Como é previsto, a execução dos testes unitários representados pela figura 12, resulta em falha. A figura 13 a seguir apresenta a falha dos testes com ícones vermelhos, que representam o ciclo do TDD que está sendo aplicado.

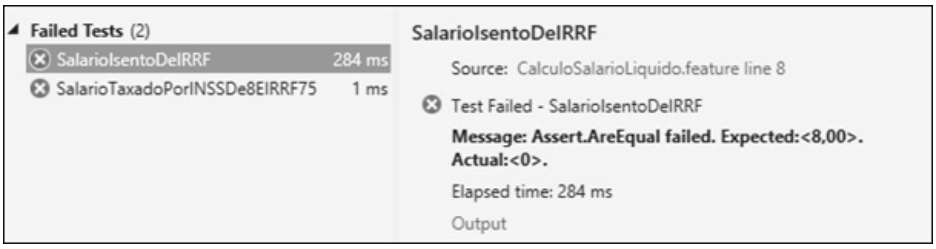

Figura 13. Falha dos testes unitários representando o ciclo vermelho do TDD

Nesse momento, é necessário passar para o ciclo "verde" do TDD. A classe "CalculadoraSalario" terá a codificação necessária para que os testes sejam válidos, conforme ilustra a figura 14.

```
class CalculadoraSalario
{
    private decimal _salario;
    private decimal _aliqInss;
    private decimal _aliqIrrf;
    private decimal _parcelaDeduzir;

    public void InformarSalario(decimal salario)
    {
        _salario = salario;
    }

    public void Calcular()
    {
        CalcularAliqINSS();
        CalcularAliqIRRF();
    }

    private void CalcularAliqINSS()
    {
        if (_salario <= 1399.12M)
        {
            _aliqInss = 8;
        }
        else if (_salario <= 2331.88M)
        {
            _aliqInss = 9;
        }
        else
        {
            _aliqInss = 11;
        }
    }
```

Figura 14. Parte da codificação necessária para os testes serem validados

O resultado da execução dos testes é uma saída com ícones verdes representando o ciclo atual do TDD, observado na figura 15.

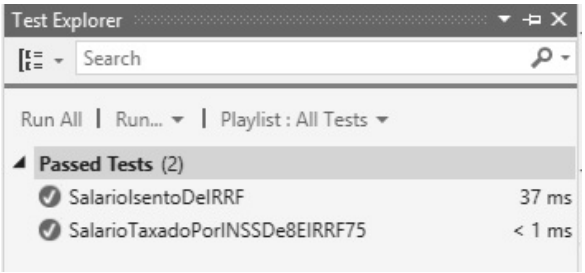

Figura 15. Testes validados para o ciclo verde do TDD

Só resta aplicar no exemplo o ciclo "Refatora" do TDD. A técnica de refatoração aplicada neste exemplo será a de manter apenas uma responsabilidade por classe e também por método. A classe "CalculadoraSalario" está com três responsabilidades, que são identificar as alíquotas de IRRF, INSS e efetuar o cálculo do salário. Para resolver esta situação, uma classe será criada para identificar as alíquotas de IRRF, uma outra classe para identificar as alíquotas de INSS e a classe "CalculadoraSalario" ficará somente com a responsabilidade de aplicar o cálculo do salário.

Na figura 16 é exibida a representação das classes criadas após mover as responsabilidades excessivas da classe "CalculadoraSalario".

```
public class CalculoIRRF
{
    private readonly decimal _salario;
    private decimal _aliqIrrf;
    private decimal _parcelaDeduzir;

    public CalculoIRRF(decimal salario)...

    private void CalcularValores()...

    public decimal ObterAliquotaIRRF()...

    public decimal ObterParecelaDeduzir()...
}

public class CalculoINSS
{
    private readonly decimal _salario;

    public CalculoINSS(decimal salario)...

    public decimal OberAliquotaINSS()...
}
```

Figura 16. Classes resultantes da refatoração

Após a separação descrita pela figura 16, por fim, é necessário utilizá-las no método "Calcular" da classe "CalculadoraSalário", conforme a figura 17.

```
public void Calcular()
{
    var calculoIRRF = new CalculoIRRF(_salario);
    _aliqIrrf = calculoIRRF.ObterAliquotaIRRF();
    _parcelaDeduzir = calculoIRRF.ObterParecelaDeduzir();

    var calculoINSS = new CalculoINSS(_salario);
    _aliqInss = calculoINSS.OberAliquotaINSS();
}
```

Figura 17. Utilização das classes resultantes da refatoração

Por fim, como um passo importante do BDD e do TDD, após a refatoração os testes unitários deverão ser executados novamente e caso haja falha, o ciclo do TDD deverá se repetir até que a funcionalidade esteja válida.

IV. Trabalhos relacionados

O trabalho postado no blog da empresa Lambda3 descreve um aplicativo semelhante com o desenvolvido neste trabalho, calculando alíquotas de IRRF e a utilização das interfaces gráficas nos testes, porém ainda não finalizado pelo autor BASSI (2009).

Um artigo completo sobre BDD e a utilização com o framework SpefFlow, chamado "BDD na prática com Specflow" que é baseado em funções de cálculo de impostos federais publicado pela .NET MAGAZINE (2013). No artigo, não há o estudo do relacionamento do BDD com os valores ágeis e o foco na produção de valor para o negócio. O TDD é somente apontado no momento da aplicação do BDD através de características semelhantes no processo de desenvolvimento, ao contrário deste trabalho que é construído com a utilização das duas técnicas sendo aplicadas simultaneamente.

O BDD também foi artefato de trabalho para a revista JAVA MAGAZINE (2011), com a criação de funcionalidades de um sistema financeiro, com controle de créditos e débitos, na linguagem de programação Java. O artigo publicado na

revista, não relaciona a integração do BDD com o TDD, apresentando brevemente o TDD como o predecessor do BDD.

V. Conclusões e trabalhos futuros

É possível observar que o BDD se mostrou um artefato de grande valor. Há muita relação da sua aplicação com os valores do desenvolvimento ágil. Suas funcionalidades podem ser escritas por qualquer indivíduo envolvido com o processo, como propõe o princípio ágil "A colaboração dos clientes acima da negociação de contratos". A proposta do BDD com associação ao framework SpecFlow auxilia em aumentar a qualidade do código e o correto funcionamento dos principais funcionalidades da aplicação, como propõe o princípio ágil "O funcionamento do software acima de documentação abrangente". As mudanças aplicadas em uma funcionalidade escrita com o BDD refletem diretamente nos testes unitários, garantindo que o sistema esteja preparado para identificar falhas, como propõe o princípio ágil "A capacidade de resposta a mudanças acima de um plano preestabelecido".

As funcionalidades escritas nos moldes do BDD podem ser utilizadas como documentação do sistema, pois estão escritas em formato de história de usuário.

Como demonstrado no exemplo "Cálculo de salário líquido", é possível utilizar BDD e TDD juntos, desta forma, é possível aumentar a quantidade de código sendo testado e há a possibilidade de melhoria técnica através da refatoração.

Os benefícios do BDD e TDD na criação de novos projetos são comentados com frequência, porém não se encontra muitas formas de utilização das tecnologias para códigos legados, sem padrões, com alto acoplamento e baixa coesão, sendo assim a criação de um modelo para utilizar essas metodologias de desenvolvimento em projetos legados é uma sugestão de trabalho futuro.

Referências

.NET MAGAZINE. DevMedia, Grajaú-RJ, edição 109, p. 27. 2013.

BASSI, G.. 2009. Disponível em <http://blog.lambda3.com.br/2011/05/bdd-com-net-parte-1/>. Acesso em: 06 set. 2015.

BECK, K. Extreme Programming Explained – Embrace Change. Addison-Wesley. 1999.

BECK, K. Test-Driven Development by Example. Addison Wesley - Vaseem, 2003.

BECK, K. et al. Manifesto for Agile Software development. 2001. Disponível em: <http://www.agilemanifesto.org/>. Acesso em: 15 jun. 2015.

COSTA FILHO, E. Integração de padrões organizacionais e de processo ao método ágil Scrum. 2006.

CHELIMSKY, D. The RSpec Book. behaviour-Driven Development with RSpec, Cucumber, and Friends. 2010.

EVANS, E. Domain-Driven Design: Tacklinkg Complexity in the Heart of Software. 2003.

GUIA TRABALHISTA, 2015. Disponível em: <http://www.guiatrabalhista.com.br/guia/tabela_irf.html>. Acesso em: 8 set. 2015.

JAVA MAGAZINE. DevMedia, Grajaú-RJ, edição 91, p. 65. 2011.

NORTH, D. Introducing BDD, 2003. Disponível em: <http://dannorth.net/introducing-bdd/>. Aceso em: 02 ago. 2015.

PREVIDÊNCIA SOCIAL, 2015. Disponível em: <http://www.previdencia.gov.br>. Acesso em: 06 set. 2015.

STEVENSON, C. agiledox, 2003. Disponível em: <http://agiledox.sourceforge.net/index.html>. Acesso em: 02 ago. 2015.