UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

WEVERTON CESAR PAULINO

TDD E BDD NO DESENVOLVIMENTO DE *SOFTWARE* EM .NET

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA
2015

WEVERTON CESAR PAULINO

TDD E BDD NO DESENVOLVIMENTO DE *SOFTWARE* EM .NET

Trabalho de Diplomação apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas – COADS – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Dr. Everton Coimbra de Araújo.

MEDIANEIRA

2015

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Graduação e Educação Profissional
Coordenação do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

TDD E BDD NO DESENVOLVIMENTO DE *SOFTWARE* EM .NET

Por

Weverton Cesar Paulino

Este Trabalho de Diplomação (TD) foi apresentado às 15:30 h do dia 08 de junho de 2015 como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná, *Campus* Medianeira. O candidato foi argüido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. Dr. Everton Coimbra de Araújo
UTFPR – *Campus* Medianeira
(Orientador)

Prof. Marcio Angelo Matté
UTFPR – *Campus* Medianeira
(Convidado)

Prof. Gloria Patricia Lopes Sepulveda
UTFPR – *Campus* Medianeira
(Convidado)

Prof. M.Eng. Juliano Rodrigo Lamb
UTFPR – *Campus* Medianeira
(Responsável pelas atividades de TCC)

RESUMO

PAULINO, C. Weverton. TDD e BDD no Desenvolvimento de *Software* em .NET. 2015. Trabalho de Diplomação (Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas), Universidade Tecnológica Federal do Paraná - PR. Medianeira, 2015.

As consequências causadas por um *software* não testado podem ser desastrosas, tendo em vista que *softwares* estão em todos os lugares. Porém, existem possíveis soluções para isso, chamadas de práticas ágeis de desenvolvimento de *software*. Sendo assim, este trabalho apresenta um estudo sobre as vantagens proporcionadas por Test Driven Development e Behavior Driven Development no desenvolvimento de aplicações utilizando *framework* .NET e linguagem C#. Também é parte deste trabalho a ilustração do desenvolvimento de uma aplicação utilizando estas técnicas por meio de um estudo experimental.

Palavras-chave: Metodologias ágeis, teste de *software*.

ABSTRACT

PAULINO, C. Weverton. TDD and BDD at the Software Development in .NET. 2015. Trabalho de Diplomação (Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas), Universidade Tecnológica Federal do Paraná - PR. Medianeira, 2015.

The consequences caused by an untested software can be disastrous, considering that software is everywhere. However, there are possible solutions to this, called agile practices for software development. Therefore, this work presents a study on the benefits conferred by Test Driven Development and Behavior Driven Development in application development using .NET framework and the C # language. Also part of this work is the development illustration of an application using these techniques through an experimental study.

Keywords: Agile methodologies, software testing.

LISTA DE FIGURAS

LISTA DE SIGLAS

BDD	*Behavior Driven Development*
CLR	*Common Language Runtime*
FCL	*Framework Class Library*
GC	*Garbage Collector*
IDE	*Integrated Development Environment*
IR	Imposto de Renda
TDD	*Test Driven Development*
XP	*Extreme Programming*

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Ao longo dos anos, a indústria de desenvolvimento de *software* passou a se preocupar com a qualidade de seus produtos não mais como uma ferramenta de *marketing*, mas sim como um requisito essencial para permanecer no mercado competitivo. Com isso, uma das saídas encontradas por muitas empresas é a aplicação de testes automatizados no *software* produzido, trazendo benefícios não somente ao cliente, mas também a toda equipe de desenvolvimento.

O TDD (*Test Driven Development*) ou Desenvolvimento Guiado por Testes, é uma prática de desenvolvimento ágil que, como o nome já diz, faz uso de testes automatizados a fim de auxiliar os desenvolvedores ainda na fase de implementação do *software*. Mas, ao contrário dos testes que a maioria dos desenvolvedores está acostumada, com TDD os testes são escritos antes mesmo do código de produção. Desta maneira, o desenvolvedor garante que boa parte do seu sistema tenha um teste que assegure o seu funcionamento. Além disso, muitos desenvolvedores também afirmam que os testes os guiam no projeto de classes do sistema (ANICHE, 2014).

Apesar dos benefícios do TDD e dos testes automatizados no desenvolvimento de *software*, um dos problemas mais corriqueiros de todo desenvolvedor é, sem dúvida, conseguir explicar para leigos como seu trabalho funciona. Isso, aliado à rapidez com que os serviços se propagam pela *Internet*, criou uma falsa impressão de que os *softwares* são fáceis de serem construídos.

Com isso em mente, Dan North, em 2003, deu origem ao BDD (*Behavior Driven Development*) ou Desenvolvimento Guiado por Comportamento, adotado atualmente como uma prática de desenvolvimento ágil de *software*. Esta técnica, apontada por muitos como uma evolução do TDD, visa estimular a colaboração entre os vários participantes de um projeto de maneira que as más interpretações e falhas de comunicação sejam minimizadas, além de integrar regras de negócios especificadas pelo cliente com a linguagem de programação (NORTH, 2006).

Este trabalho tem por objetivo aplicar as práticas de TDD e BDD no desenvolvimento de *software* utilizando a plataforma .NET a fim de apontar as principais vantagens da utilização destas práticas.

1.1 OBJETIVO GERAL

Apresentar as principais vantagens na utilização de TDD e BDD, por meio de um estudo experimental, aliando estas técnicas no desenvolvimento de uma aplicação em .NET.

1.2 OBJETIVOS ESPECÍFICOS

- Descrever as consequências no processo de desenvolvimento de *software* sem testes (ou com testes inadequados);
- Abordar, por meio de referencial teórico, os conceitos de metodologias ágeis no desenvolvimento de *software*;
- Desenvolver uma aplicação que demonstre, de maneira prática, como TDD e BDD podem auxiliar no processo de desenvolvimento de *software*.

1.3 JUSTIFICATIVA

Como todo e qualquer produto, os *softwares* também necessitam ser testados, a fim de garantir que aquilo que está sendo oferecido realmente funciona. Entretanto, as empresas que produzem estes softwares muitas vezes decidem por realizar testes somente quando tudo já está terminado, tendo como consequência uma série de problemas que acabam atrasando a entrega do produto final.

O TDD (ou Desenvolvimento Guiado por Testes) é uma prática que busca resolver esse tipo de cenário, trazendo os testes para o início do processo de desenvolvimento, antes mesmo de qualquer funcionalidade ser implementada. Assim, o desenvolvedor garante uma ampla cobertura de testes ao longo do caminho, obtendo *feedbacks* rápidos que o guiarão da maneira mais correta.

Como uma evolução do TDD, têm-se o BDD (ou Desenvolvimento Guiado por Comportamento), ambos com uma característica em comum: o código de produção fica pra depois. No entanto, ao invés de escrever testes para verificar se um método faz o esperado, no

BDD o desenvolvedor escreve especificações, e estas descrevem o comportamento desejado pela funcionalidade (ASTELS, 2006).

Sendo assim, a utilização de TDD e BDD pode ser facilmente demonstrada no desenvolvimento de uma aplicação, utilizando a plataforma .NET em conjunto com a linguagem de programação C#. Podendo então, desta maneira, favorecer ainda mais a adoção destas práticas no processo de desenvolvimento de *software*.

1.4 ESTRUTURA DO TRABALHO

O presente trabalho divide-se em cinco capítulos. O primeiro capítulo trata da contextualização do tema a ser estudado, bem como os objetivos do trabalho e a justificativa pela escolha do tema.

No segundo capítulo é feito o embasamento teórico sobre os assuntos que circundam o tema principal. O capítulo de número três apresenta um estudo experimental, a fim de ilustrar o desenvolvimento deste tema.

Por fim, os capítulos quatro e cinco, apresentam, respectivamente, os resultados e discussões, bem como as considerações finais do trabalho.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo tem como objetivo apresentar os conceitos relacionados ao tema central do trabalho, bem como abordar os tópicos necessários para o desenvolvimento da aplicação.

2.1 LINGUAGEM C#

O C# ou C Sharp (em português lê-se "cê charp") é uma linguagem de programação moderna, fortemente tipada e orientada a objetos. Embora sua sintaxe seja baseada no C++, existem muitas influências de outras linguagens de programação, como Object Pascal e Java.

A linguagem apresenta algumas características principais que incluem:

- **Simplicidade:** costuma-se dizer que essa linguagem é tão poderosa quanto o C++ e tão simples quanto o Visual Basic;
- **Completamente orientada a objetos:** para toda variável em C#, existe uma classe definida;
- **Fortemente tipada:** erros por manipulação imprópria de tipos e atribuições incorretas são evitados;
- **Tudo é objeto:** System.Object é a classe base de tipos em C#;
- **Linguagem gerenciada:** em programas desenvolvidos em C#, todo o gerenciamento de memória é feito pelo *runtime* via GC[1] (*Garbage Collector*).

No entendimento de Alves (2014) a linguagem C# define um conjunto de dados primitivos utilizado para representar as informações básicas de uma aplicação, sejam números, textos, caracteres ou dados lógicos.

A Figura 1 apresenta os tipos de dados primitivos presentes na linguagem C#, bem como seus tamanhos e valores possíveis.

[1] GC, do inglês *Garbage Collector* ou Coletor de Lixo, é um processo que gerencia o uso da memória em um aplicativo de forma automatizada, sendo a solução para problemas de gerenciamento manual de memória.

Tipo	Tamanho	Valores Possíveis
bool	1 byte	true e false
byte	1 byte	0 a 255
sbyte	1 byte	-128 a 127
short	2 bytes	-32768 a 32767
ushort	2 bytes	0 a 65535
int	4 bytes	-2147483648 a 2147483647
uint	4 bytes	0 to 4294967295
long	8 bytes	–9223372036854775808L to 9223372036854775807L
ulong	8 bytes	0 a 18446744073709551615
float	4 bytes	Números até 10 elevado a 38. Exemplo: 10.0f, 12.5f
double	8 bytes	Números até 10 elevado a 308. Exemplo: 10.0, 12.33
decimal	16 bytes	números com até 28 casas decimais. Exemplo 10.991m, 33.333m
char	2 bytes	Caracteres delimitados por aspas simples. Exemplo: 'a', 'ç', 'o'

Figura 1 - Tipos de dados primitivos em C#
Fonte: (CAELUM)

2.2 .NET *FRAMEWORK*

Microsoft .NET *Framework* ou simplesmente .NET é uma plataforma de desenvolvimento gerenciado que visa a criação de aplicações e serviços de uma maneira mais rápida e eficiente (SANTOS, 2014).

Essa poderosa ferramenta foi uma iniciativa da Microsoft, criada para atender os seguintes objetivos:

- Fornecer um ambiente de programação consistente e orientado a objetos;
- Fornecer um ambiente de execução que minimize conflitos de versão em publicações;
- Fornecer um ambiente de execução que promova a execução segura do código criado por desconhecidos ou código de terceiros;
- Fornecer um ambiente de execução que elimine os problemas de desempenho dos ambientes interpretados ou com *scripts*;

- Tornar a experiência do desenvolvedor consistente, por meio dos aplicativos baseados no Windows e na Web;
- Executar toda a comunicação usando padrões da indústria, garantindo que códigos baseados em .NET possam se integrar a qualquer outro código.

Duas partes compõem o .NET *Framework*:

1. **CLR (*Common Language Runtime*):** é o alicerce do .NET *Framework*, é quem fornece serviços que auxiliam no gerenciamento e execução das aplicações;
2. **FCL (*Framework Class Library*):** é a biblioteca de classes do *framework*, onde estão contidas inúmeras classes e métodos que possibilitam tornar a programação mais fácil e rápida.

A Figura 2 ilustra o contexto do .NET *Framework*, onde é possível visualizar o relacionamento do CLR (*Runtime*) e da FCL (*Class library*) para com seus aplicativos e para com o sistema em geral. A ilustração também mostra como os aplicativos gerenciados (*Managed applications*) operam dentro de uma arquitetura maior.

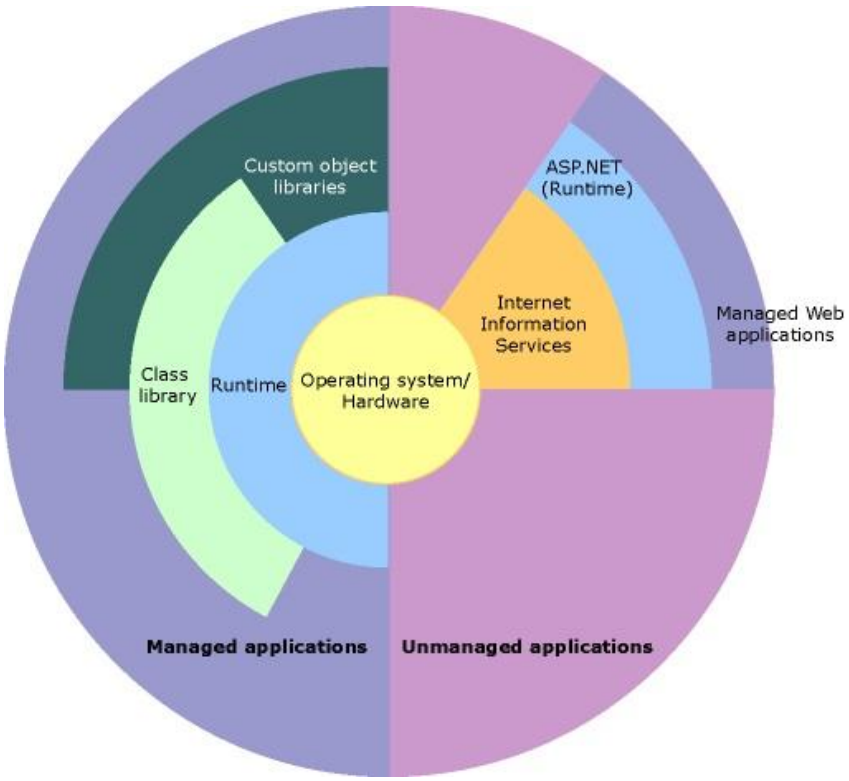


Figura 2 - Contexto do .NET *Framework*
Fonte: MSDN

Santos (2014) afirma ainda que, essas duas partes, consideradas a base do *framework*, fornecem aos desenvolvedores um modelo de programação consistente e simplificado, de

maneira que os desenvolvedores não necessitam mais entender de registros, tampouco alocar ou desalocar memória.

2.3 DESENVOLVIMENTO ÁGIL

O termo "Desenvolvimento Ágil" é fruto da constatação realizada, de maneira independente, por diversos profissionais da área de engenharia de *software*, que decidiram reunir-se no início de 2001 para discutirem maneiras de minimizar os riscos associados ao desenvolvimento de *software*. Como resultado dessa reunião, surgiu o Manifesto de Desenvolvimento Ágil de *Software*, popularmente conhecido como Manifesto Ágil.

Com base nesse manifesto valorizam-se:

- **Indivíduos e interação entre eles** mais que processos e ferramentas;
- ***Software* em funcionamento** mais que documentação abrangente;
- **Colaboração com o cliente** mais que negociação de contratos;
- **Responder a mudanças** mais que seguir um plano (BECK, 2001).

O Manifesto Ágil não rejeita os itens à direita, apenas mostra que eles têm importância secundária quando comparados com os itens à esquerda.

Além de valores, Beck (2001) também aponta doze princípios contidos no Manifesto Ágil:

1. Satisfazer o cliente, por meio da entrega adiantada e contínua de *software* de valor;
2. Aceitar mudanças de requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adaptam a mudanças, para que o cliente possa tirar vantagens competitivas;
3. Entregar *software* funcionando com frequência, na escala de semanas até meses, com preferência aos períodos mais curtos;
4. Pessoas relacionadas a negócios e desenvolvedores devem trabalhar em conjunto durante todo o curso do projeto;
5. Construir projetos ao redor de indivíduos motivados. Dando a eles o ambiente e suporte necessário e, confiar que farão seu trabalho;

6. O Método mais eficaz de se transmitir informações para, e por dentro de um time de desenvolvimento, é por meio de uma conversa cara a cara;
7. *Software* funcional é a medida primária de progresso;
8. Processos ágeis promovem um ambiente sustentável. Os patrocinadores, desenvolvedores e usuários, devem ser capazes de manter indefinidamente, passos constantes;
9. Contínua atenção a excelência técnica e bom *design* aumenta a agilidade;
10. Simplicidade: a arte de maximizar a quantidade de trabalho que não precisou ser feito;
11. As melhores arquiteturas, requisitos e *designs* emergem de times auto-organizáveis;
12. Em intervalos regulares, o time reflete em como ficar mais efetivo, então, se ajustam e aperfeiçoam seu comportamento de acordo (BECK, 2001).

Por meio desses valores e princípios pode-se notar que as metodologias ágeis não são dependentes de documentação, tampouco se preocupam apenas com a codificação. Além disso, elas são adaptativas ao invés de preditivas, ou seja, se adaptam a novos fatores decorrentes do desenvolvimento do projeto ao invés de analisarem previamente tudo o que pode acontecer no decorrer do desenvolvimento (SOARES, 2004).

Embora existam várias metodologias classificadas como ágeis, a mais conhecida entre todas é a Programação Extrema (do inglês *eXtreme Programming)* ou simplemente XP.

2.3.1 *Extreme Programming* (XP)

Extreme Programming (Programação Extrema ou XP) é uma metodologia ágil voltada para pequenas e médias equipes de desenvolvimento que baseiam seus *softwares* em requisitos vagos e que sofrem constantes mudanças (BECK, 1999).

No entendimento de Beck (2000) citado por Fagundes (2005) "XP segue um conjunto de valores, princípios e práticas básicas, adotado durante o processo de desenvolvimento por uma equipe dividida em papéis específicos, visando alcançar eficiência e efetividade".

Teles (2006) menciona os valores adotados pela metodologia XP:

- **Comunicação**: Embora existam inúmeras maneiras de se comunicar, algumas são mais eficazes que outras. Conscientes disso, equipes XP priorizam o uso do

diálogo presencial, a fim de garantir maior compreensão entre todas as partes envolvidas em um projeto;

- **Coragem**: Equipes de desenvolvimento precisam lidar com mudanças no projeto de maneira constante. Para isso, é necessário ter coragem, o que em XP se traduz na confiança em suas práticas e em seus mecanismos de proteção;
- ***Feedback***: Normalmente, quanto mais cedo um problema é descoberto, menos prejuízos ele pode causar e maiores são as chances de resolvê-lo de maneira barata. Com base nisso, projetos XP possibilitam encurtar ao máximo a defasagem de tempo entre a execução de uma ação e o seu resultado;
- **Respeito**: É o mais básico de todos os valores, além de dar sustentação a todos os demais. Assim, para o sucesso de um projeto é preciso que haja compreensão e respeito entre todos os membros de uma equipe;
- **Simplicidade**: XP utiliza o conceito de simplicidade com o objetivo de assegurar que a equipe se concentre em entregar, antes de tudo, aquilo que é claramente necessário para o cliente, evitando o desperdício de tempo em algo que ainda não se provou essencial (TELES, 2006).

Da mesma maneira que são apontados valores em XP, Teles (2006) evidencia algumas práticas que incluem:

- ***Design* Incremental**: O design de uma aplicação surge de maneira iterativa e incremental. As funcionalidades de cada iteração são implementadas com a solução mais simples possível, e os esforços da equipe são concentrados naquilo que será realmente necessário;
- **Programação em Par**: É uma das práticas mais conhecidas e mais polêmicas utilizadas por XP. Nela é sugerido que todo e qualquer código seja implementado por duas pessoas, diante do mesmo computador, revezando-se no teclado. Embora pareça ser uma prática fadada ao fracasso e ao desperdício, acredita-se que a chance de duas pessoas errarem seja significativamente menor que a chance de uma única pessoa cometer erro;
- **Código Coletivo**: Em um projeto XP, todos têm acesso e autorização para editar qualquer parte do código da aplicação. Isso se dá por meio da troca constante de pessoas entre as equipes ou pares. Desta maneira, há uma maior disseminação do conhecimento, além da frequente refatoração em áreas do código que necessitam de melhorias;

- **Desenvolvimento Orientado a Testes**: Esta prática tem como um de seus objetivos, antecipar a identificação e correção de falhas durante o desenvolvimento por meio de testes unitários criados antes mesmo do código de produção. Em TDD, quando se termina uma implementação dificilmente é preciso retornar ao código para corrigir falhas, pois possíveis falhas já foram detectadas e corrigidas durante a confecção dos testes (TELES, 2006).

2.4 QUALIDADE DE *SOFTWARE*

Conceituar qualidade de *software* é de fato uma tarefa complexa, apesar disso, ela é definida por Pressman (2011) como "uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam" (PRESSMAN, 2011, p. 360).

A premissa da qualidade era, até pouco tempo, uma forma de *marketing* para se vender um produto (MATTÉ, 2011). No entanto, devido a globalização e ao acirramento da competição entre as empresas, tornou-se obrigatória para a sobrevivência no mercado de *software*. Desta maneira, para que uma empresa se destaque neste mercado é preciso não apenas produzir *software* de qualidade, mas também que os clientes percebam a diferença em seus produtos.

Pressman (2011, p. 361) menciona a proposta criada por McCall, Richards e Walters (1977) que visa a categorização dos fatores responsáveis por influenciar na qualidade de *software*. Esses fatores, agrupados e representados na Figura 3, enfatizam três relevantes aspectos de um produto de *software*: características operacionais, habilidade de suportar mudanças e adaptabilidade a novos ambientes.

Figura 3 - Fatores de qualidade de *software* de McCall
Fonte: (PRESSMAN, 2011, p. 362)

Acerca desses fatores, Pressman (2011 apud MCCALL; RICHARDS; WALTERS, 1997) faz as seguintes definições:

- **Correção**: O quanto um programa satisfaz o que foi lhe especificado e cumpre com os objetivos definidos pelo cliente;
- **Confiabilidade**: O quanto se anseia que um programa realize as tarefas solicitadas com a precisão exigida;
- **Eficiência**: A quantidade de recursos computacionais e de código exigidos por um programa para executar sua tarefa;
- **Integridade**: O quanto é possível controlar o acesso ao *software* ou aos dados por parte de pessoas não autorizadas;
- **Usabilidade**: Esforço necessário para aprender, operar, preparar a entrada de dados e interpretar a saída gerada por um programa;
- **Facilidade de manutenção**: Esforço necessário na localização e correção de defeitos provenientes de um sistema;
- **Flexibilidade**: Quanto esforço é preciso para modificar um programa em funcionamento;
- **Testabilidade**: Refere-se ao esforço desempenhado para testar um programa de maneira que ele realize a função destinada;
- **Portabilidade**: Esforço empregado na transferência da estrutura de um *hardware* e/ou *software* para outra;

- **Reusabilidade**: O quanto é possível reutilizar um programa (ou parte de um programa) em outras aplicações;
- **Interoperabilidade**: Capacidade que um programa tem de interagir ou integrar-se a outro (PRESSMAN, 2011, p. 362).

Embora não seja tarefa fácil, medir a qualidade de uma aplicação utilizando os fatores de qualidade de McCall possibilitará uma sólida indicação da qualidade de um *software*.

2.5 TESTE DE *SOFTWARE*

O mundo moderno encontra-se cada vez mais dependente das tecnologias, sendo que muitos dos produtos ou processos do cotidiano possuem um *software* embutido ou são controlados por um, como em celulares, carros, aviões, redes de transmissão de energia, redes de comunicação telefônica, sistemas financeiros e bancários, entre outros.

Conforme Garcia (2013, p. 61), uma pesquisa do ano 2000 constatou perdas financeiras estimadas em 60 bilhões de dólares decorrentes de falhas de *software*. Essa mesma pesquisa revelou ainda que, caso fossem realizados mais testes e estes fossem mais abrangentes e eficientes, tais perdas poderiam ser reduzidas em mais de 50%.

Garcia (2013, p. 62) relata ainda que algumas falhas clássicas na área de *software* ficaram lembradas por ocasionarem consequências desastrosas, tanto em perdas financeiras quanto em perdas de vidas humanas:

- Choque da Estação Climática enviada para pesquisa em Marte – Perda de US$ 165 milhões;
- Avião comercial Airbus A320 abatido por engano – 290 vítimas fatais;
- Máquinas de radiação em tratamento de câncer – dezenas de vítimas fatais;
- Sistema de ambulância (SAMU) de Londres em 1992 – várias vítimas fatais;
- Queda de avião comercial A300 em 1994 – 264 vítimas fatais;
- Míssil Scud na Guerra do Golfo – 30 vítimas fatais;
- Vários colapsos de telefonia – milhões de pessoas atingidas (GARCIA, 2013, p. 62).

Falhas dessas proporções colocam em evidência a necessidade de desenvolvimento de *software* com mais cautela e responsabilidade, bem como a importância de testes de *software* mais completos e precisos.

Teste de *Software*, por sua vez, pode ser compreendido como o processo de execução de um produto a fim de apontar se ele atingiu suas especificações e funcionou de maneira adequada no ambiente para o qual foi projetado, assim, o seu objetivo é encontrar falhas e possibilitar que estas sejam identificadas ainda na fase de desenvolvimento do *software*.

Pressman (2011) sugere que uma estratégia para teste de *software* pode ser vista no conceito de espiral, partindo do centro para a extremidade, conforme ilustra a Figura 4.

Figura 4 - Estratégia de teste de *software*
Fonte: (PRESSMAN, 2011, p. 404)

O teste de unidade, ao centro da espiral, se concentra em cada unidade do *software* individualmente. Em seguida, o teste avança em direção ao exterior da espiral, passando pelo teste de integração, onde o produto deve ser testado a cada nova versão, assegurando que novas funcionalidades não danifiquem o que foi implementado. Após a fase de integração, o *software* é submetido a uma série de outros testes, como o teste de validação, que visa garantir a satisfação de todos os requisitos do sistema. Por fim, o teste de sistema verifica se todos os elementos do sistema, como *hardware* e base de dados, combinam entre si, materializando o produto final de *software*.

Considerando a estratégia de teste sob a perspectiva procedimental, quatro etapas são executadas sequencialmente. A Figura 5 apresenta a ideia proposta por Pressman (2011).

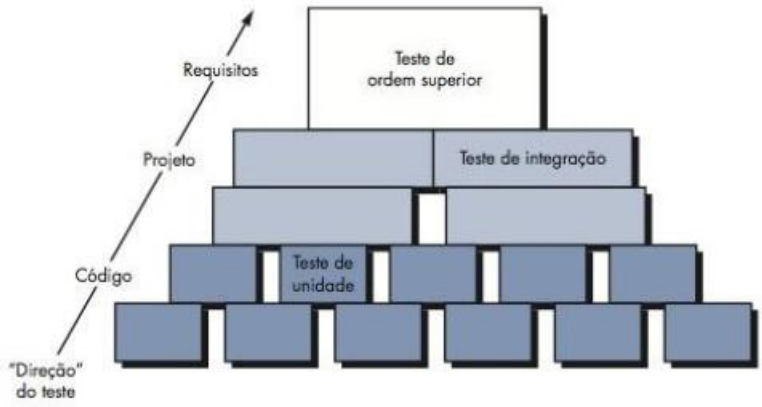


Figura 5 - Etapas de teste de *software*
Fonte: (PRESSMAN, 2011, p. 405)

O Teste de *Software* torna-se, então, umas das atividades mais custosas do processo de desenvolvimento de *software*, todavia também é fator crucial para a Qualidade de *Software*, visto que a satisfação do cliente está diretamente relacionada com o correto funcionamento do produto.

2.6 *TEST DRIVEN DEVELOPMENT* (TDD)

O TDD (*Test Driven Development*) ou Desenvolvimento Guiado por Testes é a prática de desenvolvimento de *software* que preza por antecipar os testes automatizados em uma aplicação, deixando a codificação para depois e a refatoração sempre que necessária. Com isso, ao final do processo o desenvolvedor terá uma aplicação funcional, com uma estrutura bem definida e amplamente testável.

"Código limpo que funciona", é a frase que Ron Jeffries cunhou para descrever o objetivo de TDD. No entanto, muitas forças impedem que o desenvolvedor produza código limpo, ou mesmo código que funciona. Com a proposta de contornar esta situação, Beck (2011) enfatiza duas regras básicas que devem ser seguidas no TDD, são elas:

- Escrever código novo somente quando um teste automatizado falhar;
- Eliminar código duplicado.

Essas regras, apesar de simples, implicam em uma ordem para as tarefas de programação (BECK, 2011):

1. **Vermelho** - Escrever um pequeno teste que falhe e que nem ao menos compile para começar;

2. **Verde** - Fazer rapidamente o teste funcionar, porém, escrevendo o suficiente para isso;
3. **Refatorar** - Eliminar todo o código duplicado, criado somente para fazer o teste funcionar.

A prática desse ciclo, também conhecido como ciclo **vermelho-verde-refatorar** (ou *red-green-rafactor*), traz algumas vantagens para o processo de desenvolvimento, tais como (ANICHE, 2013, p. 28):

- **Foco no teste e não na implementação**. Ao iniciar pelo teste, o desenvolvedor se mantém focado apenas no que a classe deve fazer e, por um instante, deixa de lado a implementação. Isso o ajuda a pensar em melhores cenários de teste para a classe que está sendo desenvolvida;
- **Código nasce testado**. Se o desenvolvedor pratica o ciclo corretamente, consequentemente ele terá em todo o código escrito ao menos um teste de unidade que assegure o seu correto funcionamento;
- **Simplicidade**. Por meio da constante busca por código mais simples, o desenvolvedor acaba afastando-se de soluções complexas ou desnecessárias, comuns em diversos sistemas;
- **Melhor reflexão sobre o *design* da classe**. No modelo tradicional, onde os testes são escritos depois, a falta de coesão ou o excesso de acoplamento é, muitas vezes, causado pelo desenvolvedor que se preocupa apenas com a implementação e esquece como a classe deve se comportar perante as outras. Ao começar pelo teste, este passa a atuar como o primeiro cliente da classe sob desenvolvimento, e nele o desenvolvedor decide, entre outras coisas, qual o nome que será dado à classe, os seus métodos e parâmetros. No fim, todas são decisões de *design* que contribuem para o aumento da qualidade do *design* de classes.

Tanto os desenvolvedores que praticam TDD, quanto os que utilizam a abordagem tradicional, podem obter as mesmas vantagens. Todavia, a diferença está na quantidade de *feedbacks* que cada um recebe.

Na Figura 6 é possível observar que o praticante de TDD escreve um pouco de testes, um pouco de implementação e recebe *feedback*. Isso acontece de maneira constante ao longo do desenvolvimento. Por outro lado, o desenvolvedor que não pratica TDD precisa esperar um tempo (às vezes muito longo) para receber o mesmo *feedback* (ANICHE, 2013, p. 29).

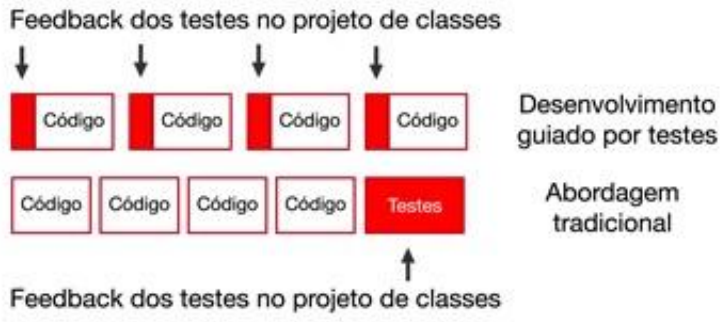


Figura 6 - *Feedback* dos testes no projeto de classes
Fonte: (ANICHE, 2013, p. 29)

No entendimento de Aniche (2013), quanto mais cedo o desenvolvedor recebe *feedbacks*, menores são os custos ao corrigir defeitos e melhor é a qualidade do código escrito. Sendo assim, o que TDD faz é maximizar a quantidade de *feedback* sobre o código que está sendo produzido, fazendo com que o desenvolvedor perceba os problemas antecipadamente e, por consequência, diminuindo os custos de manutenção e melhorando o código.

2.7 *BEHAVIOR DRIVEN DEVELOPMENT* (BDD)

Behavior Driven Development (Desenvolvimento Guiado por Comportamento ou BDD) é uma prática de desenvolvimento ágil que busca integrar regras de negócio com linguagem de programação. Nela, a colaboração entre os diversos membros de um projeto de software é largamente encorajada. Contudo, a participação e contribuição efetiva do cliente é requisito essencial para o sucesso no uso de BDD.

BDD pode ser definido como a união de várias práticas consideradas ágeis e úteis no desenvolvimento de *software*, cuja ênfase está na linguagem e nas interações utilizadas durante o processo de desenvolvimento, o que permite ao desenvolvedor escrever testes em sua linguagem nativa combinada a linguagem ubíqua[2] (*Ubiquitous Language*), possibilitando assim uma comunicação eficiente entre desenvolvedores e testadores (SOARES, 2011).

[2] Conceito de linguagem proposto por Eric Evans (2004), cuja intenção é estabelecer uma visão unificada e compartilhada entre todos os envolvidos no processo de desenvolvimento de *software*. Essa visão deve se traduzir tanto nas conversas entre desenvolvedores e clientes, quanto nos conceitos e nomenclaturas adotadas na implementação do *software*.

Sanchez (2006) menciona que BDD surgiu como tentativa de desassociar TDD ao conceito de testes, visto que a principal vantagem desta abordagem não é a validação de código e sim o seu uso para projetá-lo. Mesmo assim, alguns *frameworks* como JUnit continuaram utilizando termos como *TestCase* e *TestSuite*, forçando o desenvolvedor a nomear seus métodos com o prefixo "*test*".

Por conta disso, em 2003, Dan North empregou esforço na escrita do primeiro *framework* para trabalhar com BDD, o JBehave, cujo propósito era sobrescrever o JUnit. A nova ferramenta removeu qualquer referência a "testes", substituindo-as por um vocabulário construído a partir da verificação de "comportamento" (NORTH, 2006).

Alguns exemplos desse novo vocabulário são:

- ***Context*** ao invés de *TestCase*;
- ***Specification*** ao invés de *Test*;
- ***Should*** ao invés de *Assert* (SANCHEZ, 2006).

Soarez (2011) apud Dan North (2006) ressalta que o propósito de BDD não é anular as práticas de TDD, e sim adicionar a elas uma série de outros benefícios que incluem:

- **Comunicação entre equipes**. Na maioria das organizações, equipes de desenvolvimento e testes encontram dificuldades em trabalhar juntos para atingirem um objetivo comum. BDD permite esta integração de maneira que testadores escrevam os cenários de testes, e estes sejam implementados pelos desenvolvedores;
- **Compartilhamento de conhecimento**. O trabalho em conjunto possibilita que desenvolvedores e testadores transfiram conhecimentos uns aos outros, tornando assim a equipe multifuncional;
- **Documentação dinâmica**. Algumas equipes ágeis relatam não documentarem seus *softwares* devido a custosa manutenção de seus artefatos. *Frameworks* de BDD, porém, auxiliam na geração destes artefatos de maneira dinâmica e sem qualquer esforço adicional;
- **Visão do todo**. Um dos motivos pelo qual projetos de *software* chegam ao fracasso, é que os objetivos do projeto não são bem definidos e/ou os envolvidos não são identificados (O'CONNELL, 1999 apud SOARES, 2011). Em vista disso, BDD sugere que analistas/testadores escrevam cenários antes mesmo dos testes serem codificados, permitindo que os desenvolvedores tenham uma visão geral do objetivo do projeto antes de implementá-lo.

Enquanto em TDD foca-se no *design* do código, em BDD foca-se no comportamento do sistema. Dessa maneira, TDD é utilizado para criar parte de uma funcionalidade descrita por meio de um cenário de BDD.

Em uma analogia feita por Ribeiro (2012), pode-se dizer que:

> BDD é usado para descrever que um carro deve acelerar e para isso é necessário acionar o pedal do acelerador, o que fará com que o carro acelere a certa velocidade, o velocímetro aumente e o indicador de giro do motor exiba o giro. Por outro lado, para montar o motor é preciso descrever com TDD tudo o que está dentro dele. Adaptado de (RIBEIRO, 2012).

Em BDD, um comportamento é descrito por meio de uma história, cujo modelo é:

- **Como um** [pessoa ou papel desempenhado],
- **Eu quero** [alguma funcionalidade],
- **Para que** [benefício ou valor da funcionalidade].

Nesse contexto, a seguinte estrutura é definida para descrever um cenário:

- **Dado** (*Given*) algum contexto inicial,
- **Quando** (*When*) um evento ocorre,
- **Então** (*Then*) assegure alguns resultados.

Logo, em **dado** será definido tudo o que for preciso antes, para **quando** o evento ocorrer, **então** ser verificado o resultado.

3 MATERIAL E MÉTODOS

Este capítulo visa apresentar as ferramentas e tecnologias utilizadas no desenvolvimento do estudo experimental proposto.

3.1 FERRAMENTAS E TECNOLOGIAS UTILIZADAS

As ferramentas e tecnologias usadas no desenvolvimento do estudo experimental incluem:

- Linguagem C#;
- Ambiente Integrado de Desenvolvimento (IDE) Microsoft Visual Studio 2013; e
- Os *frameworks* NUnit, SpecFlow e WatiN.

A linguagem C# foi escolhida por ser de alto nível, robusta em performance e também por conter um vasto conteúdo de material disponível em livros e na Internet.

O Microsoft Visual Studio 2013 foi selecionado por ser da própria empresa mantedora da linguagem e suprir facilmente as necessidades encontradas ao longo do desenvolvimento.

O *framework* NUnit, utilizado para a criação dos testes unitários, encontra-se na versão 2.6.4, sendo esta a sétima versão desta ferramenta para a plataforma .NET.

Como o BDD é uma forma diferente de desenvolvimento, o SpecFlow é um *framework* que auxilia nesse processo. Sendo assim, por meio desta ferramenta os requisitos para teste podem ser automatizados.

O WatiN, na sua versão 2.1.0, foi a ferramenta utilizada para a automação da interface *web*. Essa ferramenta possui uma API fácil e pode ser compreendida até mesmo por iniciantes em desenvolvimento *web*.

Os três *frameworks* aqui citados podem ser facilmente instalados por meio do Package Manager Console, presente no Visual Studio 2013. Nele é possível executar comandos para realizar operações de instalação, atualização, configuração e remoção de pacotes.

3.2 ESTUDO EXPERIMENTAL

A estudo experimental proposto se baseia no cálculo do Imposto de Renda (IR), conforme mostra a Tabela 1. Dessa maneira, a aplicação deverá retornar o valor do imposto a ser pago de acordo com o valor do salário que lhe for informado.

Tabela 1 - Cálculo mensal do Imposto de Renda

Base de cálculo mensal em R$	Alíquota %	Parcela a deduzir do imposto em R$
Até 1.499,15	-	-
De 1.499,16 até 2.246,75	7,5	112,43
De 2.246,76 até 2.995,70	15,0	280,94
De 2.995,71 até 3.743,19	22,5	505,62
Acima de 3.743,19	27,5	692,78

Fonte: (RECEITA FEDERAL DO BRASIL, 2015)

A primeira etapa do desenvolvimento é criar, na IDE Visual Studio, um projeto C# Class Library, que será utilizado como o projeto de testes da aplicação. Em seguida, por meio do Package Manager Console, devem ser instalados os *frameworks* mencionados na seção anterior. A Figura 7 demonstra o comando usado para a instalação do *framework* NUnit, assim, o mesmo procedimento foi usado para instalar os demais *frameworks* necessários.

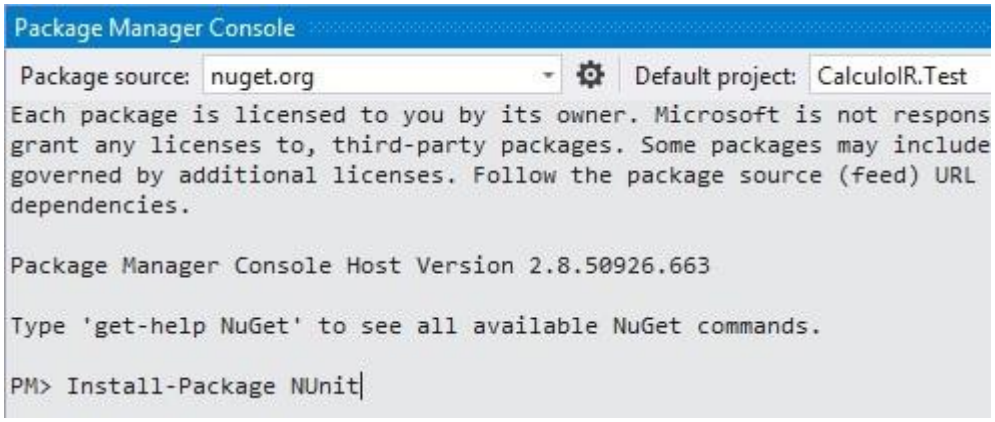


Figura 7 - Instalação do NUnit com Package Manager Console
Fonte: Autoria própria.

Na Figura 8 é possível visualizar o projeto de testes já criado e com seus *frameworks* devidamente instalados.

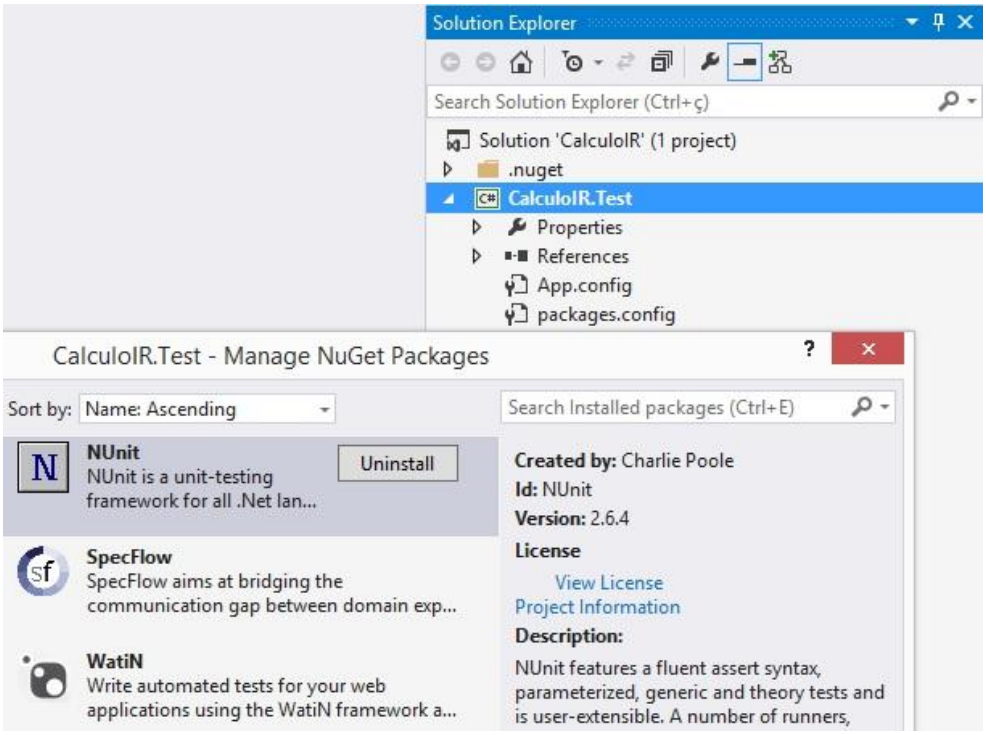


Figura 8 - Projeto de testes da aplicação
Fonte: Autoria própria.

Com o projeto de teste criado, o próximo passo é adicionar um novo arquivo para representar uma funcionalidade da aplicação. O Quadro 1 exibe o arquivo **ValorIR.feature** com as características de comportamento e cenário implementado.

```
#language: pt-br

Funcionalidade: Valor do IR
       Como um contribuinte
       Eu quero calcular o valor do IR
       Para que eu saiba quanto devo pagar de imposto

Cenário: Salário de 1000 reais ficará isento de imposto
       Dado que estou na página do IR
       E preencho o campo Salário com o valor de 1000,00
       Quando clico no botão Calcular
       Então vejo a mensagem 'Imposto: 0,00'
```

Quadro 1 - Conteúdo do arquivo ValorIR.feature
Fonte: Autoria própria.

O SpecFlow permite a geração do código utilizado para implementar os passos que executarão cada frase do arquivo de funcionalidades. Para isso, basta acrescentar um arquivo de passos chamado de **Step Definition** e, em seguida, basta copiar e colar o código gerado pelo arquivo de funcionalidade (*feature*), conforme Quadro 2.

```
[Binding]
public sealed class IRSteps
{
    [Given(@"que estou na página do IR")]
    public void DadoQueEstouNaPaginaDoIR()
    {
        ScenarioContext.Current.Pending();
    }

    [Given(@"preencho o campo Salário com o valor de (.*)")]
    public void DadoPreenchoOCampoSalarioComOValorDe(Decimal p0)
    {
        ScenarioContext.Current.Pending();
    }

    [When(@"clico no botão Calcular")]
    public void QuandoClicoNoBotaoCalcular()
    {
        ScenarioContext.Current.Pending();
    }

    [Then(@"vejo a mensagem '(.*)'")]
    public void EntaoVejoAMensagem(string p0)
    {
        ScenarioContext.Current.Pending();
    }
}
```

Quadro 2 - Conteúdo da classe IRSteps
Fonte: Autoria própria.

O código acima corresponde, exatamente, com a *feature* contida no Quadro 1, porém, ainda é preciso implementar o trabalho real. Para isso, utilizando o WatiN é possível implementar as funções que irão chamar o navegador, preencher os campos, clicar em botões e avaliar os resultados. Dessa maneira, a implementação da classe **IRSteps** pode ser visualizada no Quadro 3.

```
[Binding]
public sealed class IRSteps
{
    private IE _browser;

    [Given(@"que estou na página do IR")]
    public void DadoQueEstouNaPaginaDoIR()
    {
        _browser = new IE("http://localhost:32494/IR", true);
    }
```

```
        [Given(@"preencho o campo Salário com o valor de (.*)")]
        public void DadoPreenchoOCampoSalarioComOValorDe(Double salario)
        {
            _browser.TextField(Find.ByName("salario"))
                .TypeText(salario.ToString());
        }

        [When(@"clico no botão Calcular")]
        public void QuandoClicoNoBotaoCalcular()
        {
            _browser.Button(Find.ByValue("Calcular")).Click();
        }

        [Then(@"vejo a mensagem '(.*)'")]
        public void EntaoVejoAMensagem(string mensagem)
        {
            Assert.IsTrue(_browser.ContainsText(mensagem));
        }
    }
```

Quadro 3 - Classe IRSteps implementada
Fonte: Autoria própria.

Ao ser executada, a especificação acima falhará, visto que a página IR ainda não existe, de maneira que será preciso implementá-la, ou seja, criar o *controller* e as classes de domínio.

Todavia, uma das premissas de TDD é que os testes devem ser escritos antes do código de produção. Sendo assim, o Quadro 4 apresenta o primeiro caso de teste criado, cujo propósito é verificar se a *view* retornada pelo *controller* é, de fato, a página do IR.

```
    [TestFixture]
    public class IRControllerTest
    {
        [TestCase]
        public void ir_controller_should_show_index_view()
        {
            var controller = new IRController();
            var result = controller.Index() as ViewResult;

            Assert.AreEqual("Index", result.ViewName);
        }
    }
```

Quadro 4 - Primeiro caso de teste em IRControllerTest
Fonte: Autoria própria.

Com base no teste que acaba de ser criado, é possível observar que a classe **IRController**, bem como o método **Index(...)** ainda não foram implementados e, dessa

maneira, o projeto nem ao menos compila. Esta fase dentro do ciclo de TDD é chamada de *red*, ou vermelho, conforme mencionado no capítulo dois deste documento.

O próximo passo é escrever o código necessário apenas para fazer o teste passar. Para isso, criou-se um projeto ASP.NET MVC e nele foi implementado o *controller* correspondente ao IR, como mostra o Quadro 5.

```
public class IRController : Controller
{
    [HttpGet]
    public ActionResult Index()
    {
        return View("Index");
    }
}
```

Quadro 5 - Método Index da classe IRController
Fonte: Autoria própria.

O código escrito no método **Index(...)** já é o suficiente para fazer o teste passar, o que conclui a segunda etapa do ciclo de TDD conhecida como *green*, ou verde. A Figura 9 mostra o teste bem sucedido, contudo, o cenário encontra-se incompleto devido a funcionalidade que, por hora, não foi completamente atendida.

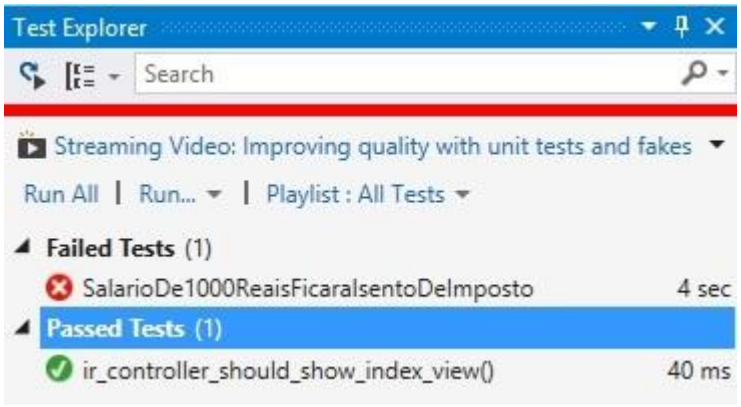


Figura 9 - Primeiro caso de teste passando
Fonte: Autoria própria.

Com o teste validado, o trabalho seguinte é a refatoração, que no ciclo de TDD é chamado de *refactor*, ou refatorar. No entanto, o código implementado no método **Index(...)** já cumpre à sua obrigação e não há necessidade de melhorias no código.

O próximo caso de teste visa garantir que, com base no valor do salário informado, a aplicação retorne o valor do imposto adequado. Dessa maneira, o Quadro 6 apresenta o segundo caso de teste criado na classe **IRControllerTest**.

```
[TestCase(1000, 0)]
public void ir_controller_should_show_calculed_result
    (double salario, double imposto)
{
    var controller = new IRController();
    var result = controller.Calculate(salario) as ViewResult;
    var calc = result.ViewData.Model;

    Assert.AreEqual(imposto, calc);
}
```

Quadro 6 - Segundo caso de teste em IRControllerTest
Fonte: Autoria própria.

Esse último teste, assim como o primeiro, inicialmente falhará, visto que o método **Calculate(...)** ainda não foi implementado. No Quadro 7 é possível visualizar a implementação inicial desse método, que apenas retorna o valor 0 esperado pelo teste.

```
[HttpPost]
public ViewResult Calculate(double salario)
{
    return View("Index", 0);
}
```

Quadro 7 - Método Calculate da classe IRController
Fonte: Autoria própria.

Na Figura 10 é possível visualizar os casos de teste passando. Enquanto isso, o cenário permanece incompleto.

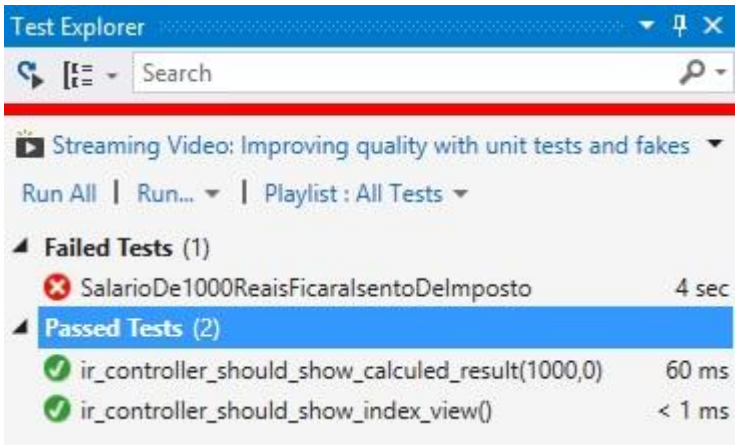


Figura 10 - Segundo caso de teste passando
Fonte: Autoria própria.

Na terceira etapa do ciclo de TDD que corresponde à refatoração, o método **Calculate(...)** foi reescrito e, assim, incluída e lógica para o cálculo do IR. O resultado pode ser visto no Quadro 8.

```
[HttpPost]
public ViewResult Calculate(double salario)
{
    double result = 0;

    if (salario <= 1499.15)
        result = 0;
    if (salario >= 1499.16 && salario <= 2246.75)
        result = 112.43;
    if (salario >= 2246.76 && salario <= 2995.70)
        result = 280.94;
    if (salario >= 2995.71 && salario <= 3743.19)
        result = 505.62;
    if (salario > 3743.19)
        result = 692.78;

    return View(result);
}
```

Quadro 8 - Método Calculate refatorado
Fonte: Autoria própria.

Após executar o teste e vê-lo ser novamente validado, o trabalho seguinte é abstrair essa lógica para outra classe, onde seja possível reutilizá-la futuramente. Entretanto, nenhum código de produção deve ser escrito sem antes haver um teste que garanta o seu correto funcionamento. Sendo assim, criou-se um terceiro teste de unidade e sua implementação pode ser vista no Quadro 9.

```
[TestCase(1000, 0)]
public void ir_model_should_calc_ir_from_salary
    (double salario, double imposto)
{
    var irModel = new IRModel(salario);
    Assert.AreEqual(imposto, irModel.ValorImposto);
}
```

Quadro 9 - Primeiro caso de teste em IRModelTest
Fonte: Autoria própria.

Com a execução do teste é possível vê-lo falhar, pois a classe **IRModel** precisa ser criada e devidamente implementada. Assim, o resultado da sua implementação é exibido no Quadro 10.

```
public class IRModel
{
    public IRModel(double valorSalario)
    {
        _valorSalario = valorSalario;
        _calculate();
    }

    private double _valorSalario;
    private double _valorImposto;

    public double ValorImposto
    {
        get { return _valorImposto; }
    }

    private void _calculate()
    {
        if (_valorSalario <= 1499.15)
            _valorImposto = 0;
        if (_valorSalario >= 1499.16 && _valorSalario <= 2246.75)
            _valorImposto = 112.43;
        if (_valorSalario >= 2246.76 && _valorSalario <= 2995.70)
            _valorImposto = 280.94;
        if (_valorSalario >= 2995.71 && _valorSalario <= 3743.19)
            _valorImposto = 505.62;
        if (_valorSalario > 3743.19)
            _valorImposto = 692.78;
    }
}
```

Quadro 10 - Conteúdo da classe IRModel
Fonte: Autoria própria.

Com base no Quadro 10 é possível observar que a lógica para calcular o imposto passou a pertencer ao *model*, deixando para o *controller* a tarefa de exibir o resultado na *view*, apenas. O Quadro 11 demonstra o resultado do método **Calculate(...)** após essa alteração.

```
[HttpPost]
public ViewResult Calculate(double salario)
{
    var irModel = new IRModel(salario);
    ViewBag.Mensagem = String.Format("Imposto: {0:N2}", irModel.ValorImposto);

    return View("Index", irModel);
}
```

Quadro 11 - Resultado final do método Calculate
Fonte: Autoria própria.

Pode-se dizer que, o objeto modelo, exibido no Quadro 10, ao receber a lógica do *controller* tornou-se imutável (sem *setters*), de maneira que o valor do salário tem referência direta ao valor do imposto.

Na Figura 11 é possível visualizar todos os testes passando com sucesso, inclusive o cenário, que atendeu à sua última especificação ao exibir o valor do imposto na *view*.

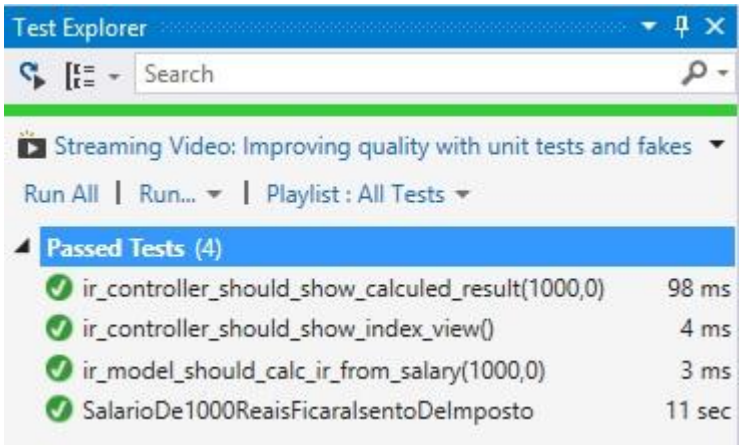


Figura 11 - Casos de teste e cenário passando
Fonte: Autoria própria.

Conforme o cenário de teste é executado, o WatiN passa a realizar todas as ações diretamente na *view* de maneira automatizada e o resultado referente ao cálculo do IR é exibido na própria *view*, conforme Figura 12.

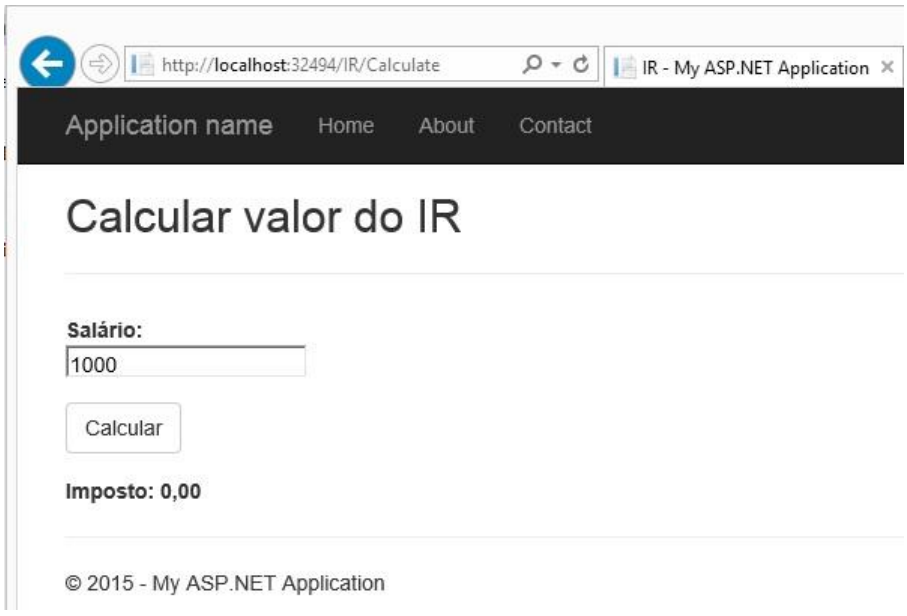


Figura 12 - *View* correspondente ao *controller* IR
Fonte: Autoria própria.

4 RESULTADOS E DISCUSSÕES

Este capítulo visa apresentar os resultados obtidos com a adição de quatro novos casos de teste, bem como determinar qual proporção do código do projeto está sendo testada de fato por testes codificados, como os testes de unidade.

4.1 EXECUÇÃO DE TESTES

Com o objetivo de testar todos os valores possíveis que a aplicação pode retornar, foram incluídos quatro novos casos de teste, conforme Quadro 12. Dessa maneira, os valores que serão comparados por meio do *Assert* correspondem, exatamente, aos valores exibidos na Tabela 1.

```
[TestFixture]
public class IRModelTest
{
    [TestCase(1000, 0)]
    [TestCase(2000, 112.43)]
    [TestCase(2500, 280.94)]
    [TestCase(3000, 505.62)]
    [TestCase(4000, 692.78)]
    public void ir_model_should_calc_ir_from_salary
        (double salario, double imposto)
    {
        var irModel = new IRModel(salario);
        Assert.AreEqual(imposto, irModel.ValorImposto);
    }
}
```

Quadro 12 - IRControllerTest com novos casos de teste
Fonte: Autoria própria.

A seguir, a Figura 12 permite visualizar os novos casos de teste sendo validados, inclusive o cenário, concluindo, dessa maneira, os ciclos de TDD e BDD.

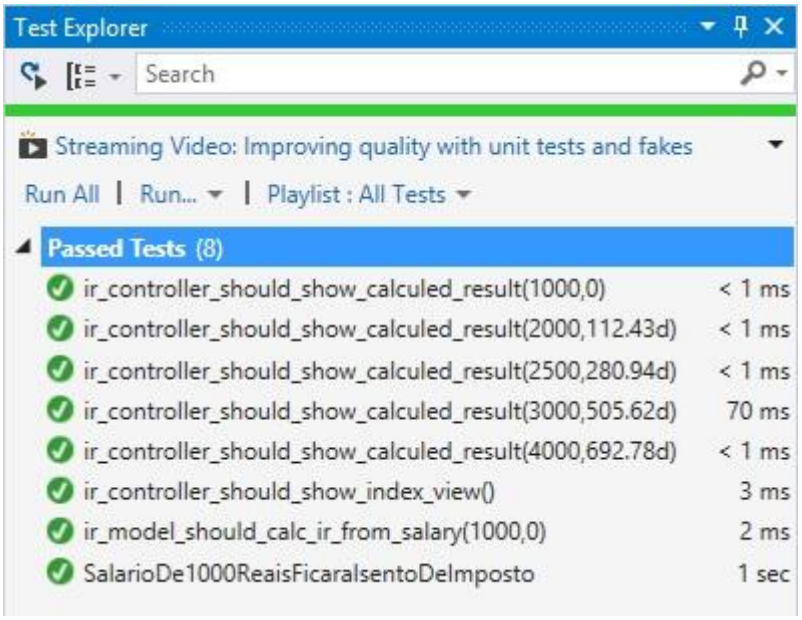
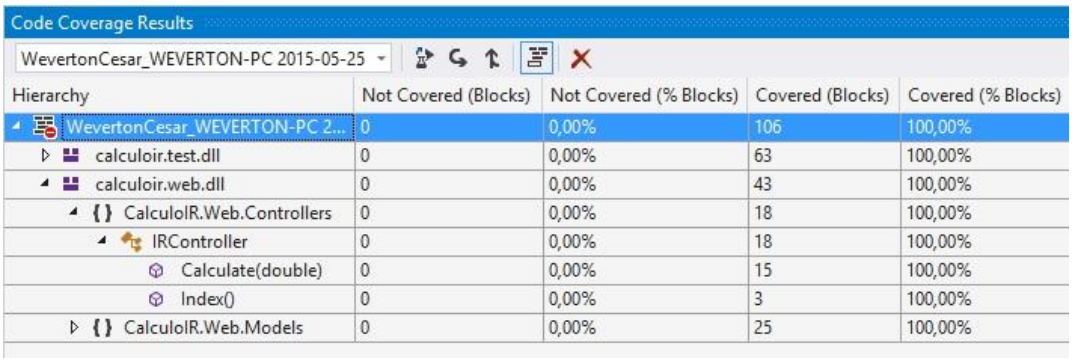


Figura 12 - Novos casos de teste passando
Fonte: Autoria própria.

4.2 COBERTURA DE CÓDIGO

A ferramenta de cobertura de código (*Code Coverage*) é uma opção quando se executam métodos de teste usando o Gerenciador de Teste do Visual Studio 2013. Esse recurso permite determinar que proporção do código da aplicação está sendo testada de fato por testes automatizados, como os testes de unidade.

A tabela de resultados, exibida na Figura 13, indica a porcentagem do código que foi executada em cada *assembly*, classe e método.



Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
WevertonCesar_WEVERTON-PC 2...	0	0,00%	106	100,00%
calculoir.test.dll	0	0,00%	63	100,00%
calculoir.web.dll	0	0,00%	43	100,00%
{ } CalculoIR.Web.Controllers	0	0,00%	18	100,00%
IRController	0	0,00%	18	100,00%
Calculate(double)	0	0,00%	15	100,00%
Index()	0	0,00%	3	100,00%
{ } CalculoIR.Web.Models	0	0,00%	25	100,00%

Figura 13 - *Code Coverage Results*
Fonte: Autoria própria.

A cobertura de código é contada em blocos, sendo que um bloco é uma parte do código com exatamente um ponto de entrada e saída. Dessa maneira, um bloco só é considerado coberto quando o fluxo do programa passar por ele durante a execução do teste.

Como pôde ser visto na Figura 13, o resultado da análise realizada mostra que cada método teve 100% de seu código coberto por testes, garantindo, assim, uma aplicação segura, altamente testável e de fácil manutenção.

5 CONSIDERAÇÕES FINAIS

O presente capítulo visa apresentar as considerações finais do trabalho bem como apontar os trabalhos futuros acerca do tema apresentado.

5.1 CONCLUSÃO

Neste estudo experimental, constatou-se a importância das metodologias de teste no desenvolvimento de *software*, sobretudo em aplicações que utilizam a plataforma .NET, que além de possibilitar a construção de aplicações de maneira ágil, auxilia o desenvolvedor na codificação de testes de unidade.

Com o uso de TDD como prática de teste, a codificação tornou-se mais simples, visto que um método pôde ser quebrado em várias partes, separando as responsabilidades e ajudando a entender melhor o código produzido.

A aplicação de BDD foi de suma importância para a clara compreensão do *software* que viria a ser escrito, de maneira que ao aplicar esta técnica no processo de desenvolvimento, torna fácil a aproximação de desenvolvedores e testadores, que muitas vezes são desencorajados a trabalharem juntos para atingirem um objetivo comum.

É importante ressaltar ainda que a aplicação desenvolvida, apesar de simples, demonstrou de maneira eficiente como é possível escrever uma aplicação com um grau de confiabilidade maior no código produzido.

Sendo assim, este trabalho alcançou seu objeto ao aliar as técnicas de TDD e BDD e, com isso, produzir uma aplicação confiável, de fácil manutenção e com suas classes e métodos totalmente cobertas por testes de unidade.

5.2 TRABALHOS FUTUROS/CONTINUAÇÃO DO TRABALHO

Este trabalho foi o primeiro passo no estudo de metodologias ágeis, servindo como estímulo para colocá-las em prática no dia-a-dia.

Contudo, há ainda uma vasta gama de conteúdo a ser visto, o que permitirá, por meio de um estudo futuro, explorar as técnicas de TDD e BDD em outros *frameworks* e linguagens que proporcionem a criação e execução de testes automatizados.

REFERÊNCIAS BIBLIOGRÁFICAS

ALVES, U. **Uma Visão Geral sobre Tipos de Dados em C#**, 2014. Disponível em: <http://csprogrammer.net/visao-geral-sobre-tipos-de-dados-em-csharp/>. Acesso em: 16 Setembro 2014.

ANICHE, M. **Test-Driven Development: Test e Deign no Mundo Real com .NET**. São Paulo: Casa do Código, 2013.

ANICHE, M. **Mais uma vez... TDD não é bala de prata!**, 2014. Disponível em: <http://blog.caelum.com.br/mais-uma-vez-tdd-nao-e-bala-de-prata/>. Acesso em: 05 Agosto 2014.

ASTELS, D. **A New Look at Test Driven Development**, 2006. Disponível em: <http://blog.daveastels.com.s3-website-us-west-2.amazonaws.com/2014/09/29/a-new-look-at-test-driven-development.html>. Acesso em: 27 Agosto 2014.

BECK, K. **Programação Extrema Explicada**. Bookman, 1999.

BECK, K. **Manifesto for Agile Software Development**, 2001. Disponível em: <http://www.agilemanifesto.org/>. Acesso em: 15 Outubro 2014.

BECK, K. **TDD: Desenvolvimento Guiado por Testes**. Porto Alegre: Bookman, 2010.

CAELUM. **Variáveis e Tipos Primitivos**, 2000. Disponível em: <http://www.caelum.com.br/apostila-csharp-orientacao-objetos/variaveis-e-tipos-primitivos/>. Acesso em: 10 Fevereiro 2015.

EVANS, E. **Domain Driven Design - Tackling Complexity int the Heart of Software**, 2004.

FAGUNDES, P. B. **Framework para Comparação e Análise de Métodos Ágeis**, 2005. Disponível em: <https://repositorio.ufsc.br/bitstream/handle/123456789/101860/220977.pdf>. Acesso em: 18 Novembro 2014.

GARCIA, L. F. F. **Engenharia de Software I**. Canoas: Ulbra, 2013.

KUHN, G. R. Apresentando XP. Encante seus clientes com Extreme Programming. **Javafree**, 2009. Disponível em: <http://javafree.uol.com.br/artigo/871447/Apresentando-XP-Encante-seus-clientes-com-Extreme-Programming.html>. Acesso em: 17 Dezembro 2014.

MATTÉ, M. A. **Teste de Software: Uma Abordagem da Atividade de Teste de Software em Metodologias Ágeis Aplicando a Técnica Behavior Driven Development em um Estudo Experimental**, 2011.

MSDN, **Visão Geral do .NET Framework**. Disponível em: <https://msdn.microsoft.com/pt-br/library/zw4w595w(v=vs.110).aspx>. Acesso em: 03 Março 2015.

NORTH, D. **Introducing BDD**, 2006. Disponível em: <http://dannorth.net/introducing-bdd/>. Acesso em: 08 Agosto 2014.

PRESSMAN, R. S. **Engenharia de Software - Uma Abordagem Profissional**. 7a. ed. Porto Alegre: Bookman, 2011.

RIBEIRO, C. **Entendendo BDD com Cucumber – Parte I. The Bug Bang Theory**, 2012. Disponível em: <http://www.bugbang.com.br/entendendo-bdd-com-cucumber-parte-i/>. Acesso em: 25 Fevereiro 2015.

SANCHEZ, I. Apresentando Behaviour Driven Development (BDD). **Coding Dojo Floripa**, 2006. Disponível em: <https://dojofloripa.wordpress.com/2006/10/28/apresentando-behaviour-driven-development-bdd/>. Acesso em: 23 Fevereiro 2015.

SANTOS, W. **Introdução ao .Net Framework - Parte I**, 2014. Disponível em: <https://wenndersantos.wordpress.com/2014/03/21/48/>. Acesso em: 02 Setembro 2014.

SOARES, I. Desenvolvimento Orientado por Comportamento (BDD). **Devmedia**, 2011. Disponível em: <http://www.devmedia.com.br/space/ismael-soares>. Acesso em: 12 Janeiro 2015.

SOARES, M. S. **Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software**, 2004. Disponível em: <http://www.dcc.ufla.br/infocomp/artigos/v3.2/art02.pdf>. Acesso em: 19 Setembro 2014.

TELES, V. M. **Extreme Programming**, 2006. Disponível em: <http://www.desenvolvimentoagil.com.br/xp/>. Acesso em: 27 Outubro 2014.