

Luiz Cezer Marrone Filho

**Desenvolvimento de Software utilizando
abordagem do Behavior Driven
Development**

Monografia apresentada no curso de Pós-Graduação do Centro Universitário Católica de Santa Catarina como requisito parcial para obtenção do certificado do curso.

Joinville
2015

Luiz Cezer Marrone Filho

Desenvolvimento de Software utilizando abordagem do Behavior Driven Development

Monografia apresentada no curso de Pós-Graduação do Centro Universitário Católica de Santa Catarina como requisito parcial para obtenção do certificado do curso.

Área de Concentração: Pós Graduação em Engenharia de Software

Orientador: Maurício Henning

Joinville
2015

Filho, Luiz Cezer Marrone

Desenvolvimento de Software utilizando abordagem do Behavior Driven Development. Joinville, 2015.

Monografia - Centro Universitário Católica de Santa Catarina.

1. Desenvolvimento de Software 2. Teste de Software 3. BDD I. Centro Universitário Católica de Santa Catarina. Curso de Bacharelado em Sistemas de Informação. Curso de Pós-Graduação em Engenharia de Software.

Sumário

Sumário	i
Lista de Figuras	iii
Lista de Tabelas	v
Resumo	vi
Abstract	vii
 Capítulo 1	
Introdução	1
1.1 Objetivo Geral	2
1.2 Objetivos Específicos	2
1.3 Justificativa do Trabalho	2
1.4 Estrutura do Trabalho	3
 Capítulo 2	
Referencial Teórico	4
2.1 Falhas em projetos de software	4
2.2 Metodologias Ágeis	5
2.3 Extreme Programming (XP)	6
2.4 Scrum	8
2.4.1 O time do Scrum	9
2.4.2 Artefatos do Scrum	10
2.4.3 Eventos do Scrum	10

2.5	Qualidade de Software	11
2.6	Testes de Software	13
2.7	Test Driven Development (TDD)	15
2.8	Behavior Driven Development (BDD)	18
2.9	Linguagem Ruby	21
 Capítulo 3		
	Materiais e Métodos	23
3.1	Tecnologias Utilizadas	23
3.2	Estudo Experimental	24
 Capítulo 4		
	Resultados e Discussões	38
 Capítulo 5		
	Considerações Finais	39
	Referências Bibliográficas	40

Lista de Figuras

2.1	Visão geral de uma <i>Sprint</i> dentro do <i>Scrum</i>	9
2.2	A Pirâmide de qualidade de <i>software</i>	13
2.3	O espiral dos testes de <i>software</i>	15
2.4	O ciclo do TDD	16
2.5	<i>Feedback</i> do código utilizando TDD	18
2.6	O ciclo do BDD	20
2.7	Exemplo de código Ruby	22
3.1	Estrutura inicial da aplicação	25
3.2	Configuração dos <i>frameworks</i> de teste	26
3.3	Criando primeiro teste de aceitação	27
3.4	Executando primeiro teste de aceitação	27
3.5	Passos do teste de aceitação ainda pendentes	28
3.6	Criando os testes de aceitação reais	29
3.7	Executando a especificação	30
3.8	Criando primeiro teste unitário que irá falhar	31
3.9	Executando testes unitários para a classe <i>SalaryCalculator</i>	31
3.10	Codificação mínima para a classe <i>SalaryCalculator</i>	32
3.11	Executando testes unitários, testes funcionando corretamente	32
3.12	Criando teste unitários de acordo com especificação	33

3.13	Executando testes unitários com base na especificação	34
3.14	Criando código mínimo para que os testes passem	34
3.15	Executando novamente os testes unitários	35
3.16	Reescrevendo o código da classe <i>SalaryCalculator</i>	36
3.17	Executando novamente os testes unitários, código continua funcionando após a refatoração	37
3.18	Executando a especificação, código funcionando corretamente	37

Lista de Tabelas

3.1	Tabela do Cálculo do Imposto de Renda - 2015	24
3.2	Tabela do Cálculo do INSS - 2015	24

Resumo

Utilizar testes de *software* no seu desenvolvimento irá proporcionar um código de melhor qualidade, confiança e segurança. O uso de TDD irá ajudar ao desenvolvedor a ter um código melhor e mais simples e o BDD irá ajudar na validação de uma especificação de funcionalidade. Esse trabalho tem como objetivo mostrar através de um referencial teórico e um estudo prático, como a utilização de técnicas de TDD e BDD podem ajudar a equipes ágeis a chegar nesse objetivo.

Palavras-chave: Desenvolvimento de Software, Teste de Software, BDD, TDD

Abstract

Missing...

Keywords: Software Development, Software Test, BDD, TDD

Capítulo 1

Introdução

Desenvolver *software* é uma tarefa complicada, devido aos vários fatores que estão envolvidos na sua construção e devem ser avaliados antes mesmo do projeto ser iniciado. Porém nos dias atuais, entregar um *software* funcionando não basta, é preciso que o produto entregue tenha qualidade, segurança e atenda a real necessidade para o qual ele se propõe.

Entre as técnicas que tentam melhorar a acurácia e a qualidade do *software* entregue, surge o TDD. O TDD (*Test Driven Development* ou Desenvolvimento Guiado por Testes) visa minimizar os problemas do desenvolvimento e perda de tempo com *debug*, adotando a técnica de criar testes antes de criar código. (CORBUCCI; ANICHE, 2013)

Com essa troca de hábito o desenvolvedor irá primeiro criar um código que irá falhar, após isso irá criar um código o mais simples possível apenas para que o teste antes falhe, funcione. Na última etapa o desenvolvedor irá codificar a melhor solução para resolver aquele teste, sempre focando na simplicidade. Ao final desse processo, já é possível ter um código simples, que resolva o problema testado e feito da melhor maneira possível. (CORBUCCI; ANICHE, 2013)

O TDD foca no código e nas suas melhores práticas, porém para validar se a funcionalidade entregue realmente cumpre seu papel é preciso uma outra abordagem. O BDD surge então como um complemento ao TDD, onde ao invés de focar no código, irá focar no comportamento e tentará validar se o código entregue consegue garantir que o comportamento esperado pelo *software* esteja correto. (NORTH, 2006)

Além disso o BDD permite uma forma de integrar equipes de negócio e desenvolvimento, fazendo com que as duas falem a mesma linguagem e tenham um objetivo único, entregar *software* com qualidade e que atenda os requisitos esperados pelo cliente. (SAN-

CHEZ, 2006) (BARAÚNA, 2013)

Esse trabalho tem como foco conceitualizar sobre o uso das técnicas de TDD e BDD no desenvolvimento ágil de *software* e através de um estudo experimental avaliar os benefícios da utilização dessas abordagens.

1.1 Objetivo Geral

Apresentar por meio de um estudo experimental as vantagens do desenvolvimento de *software* utilizando a abordagem do BDD.

1.2 Objetivos Específicos

1. Contextualizar por meio de referencial teórico sobre métodos ágeis, testes e qualidade do *software*;
2. Contextualizar por meio de referencial teórico as práticas e como desenvolver *software* utilizando abordagem do TDD e do BDD;
3. Desenvolver um exemplo prático utilizando BDD para demonstrar como essa abordagem auxilia no desenvolvimento de *software*;

1.3 Justificativa do Trabalho

Os riscos de se desenvolver *software* sem o mínimo de testes é muito grande. Esses riscos vão desde atrasos no projeto, demora na resolução de *bugs*, rigidez na hora de executar uma alteração no projeto. Além desses problemas estruturais como código mal feito também podem surgir problemas na hora de validar se o código entregue realmente atende a necessidade para o qual ele se propõe.

A utilização da técnica de TDD pelos desenvolvedores irá causar uma mudança na forma como o desenvolver irá escrever seu código. Abordando essa técnica, o desenvolver irá escrever sempre ao menos um teste antes de criar determinado código, fazendo essa troca na ordem no seu dia a dia o desenvolvedor irá se forçar a sempre focar em uma solução simples que irá resolver o problema da melhor maneira e obtendo um *feedback* constante de seu trabalho. Dessa forma o código ficará mais enxuto, simples e com um

mínimo de cobertura de testes. (CORBUCCI; ANICHE, 2013)

Porém para equipes ágeis que utilizam *Scrum* apenas código entregue com qualidade não basta, é preciso um modo para validar se o código que foi escrito, realmente irá atender a necessidade de negócio para o qual ele foi proposto. Com a utilização do BDD é possível integrar equipes de áreas diferentes para escrever especificações de *software* executáveis, através das *User Stories* do *Scrum* e assim conseguir validar a funcionalidade entregue. (BARAÚNA, 2013)

Dessa forma, o uso de TDD e BDD no processo de desenvolvimento de *software*, pode ser demonstrado de forma simples em aplicações *Ruby*, que além de agilizar o desenvolvimento também oferece vários *frameworks* para aplicação do TDD e BDD.

1.4 Estrutura do Trabalho

O trabalho se divide em seis capítulos, sendo o primeiro capítulo responsável pela contextualização do tema, definindo os objetivos e a justificativa pela escolha do trabalho.

O segundo capítulo trata de todo o embasamento teórico, onde serão tratadas todos os conceitos para o claro entendimento do trabalho. O terceiro capítulo apresenta o estudo experimental, onde será construída uma aplicação utilizando as técnicas de teste que foram conceituadas no capítulo anterior e ao final do desenvolvimento apontar as principais vantagens e conclusões sobre o estudo desenvolvido. O quarto capítulo apresenta as discussões sobre os resultados obtidos.

O quinto capítulo apresentará as considerações finais do trabalho. E no sexto e último capítulo serão apresentadas as referências bibliográficas sobre os assuntos abordados no trabalho.

Capítulo 2

Referencial Teórico

Neste capítulo serão apresentados os principais conceitos que serão abordados durante o trabalho.

2.1 Falhas em projetos de software

Desenvolvimento de *software* é uma tarefa complicada para se praticar. Muitas questões devem ser levadas em conta antes mesmo de iniciar a codificação, por exemplo, é preciso estudar sobre o problema que deve ser solucionado, escolher a melhor tecnologia para o projeto, estudar a viabilidade e planejar uma metodologia que consiga organizar e gerenciar o projeto da melhor forma.

Além desses fatores, existem outros que devem ser seguidos, como boas práticas de desenvolvimento de *software*, padrões de desenvolvimento pela equipe para que todos possam trabalhar da melhor forma em conjunto. Entre os pontos importantes dentro do processo, existe o teste de *software*, que deve ser feito para garantir que o que foi codificado está funcionando e também irá suprir a necessidade do cliente. (PRESSMAN, 2011) (BECK, 2001)

Testar *software* de maneira manual, abrindo todas as telas do projeto e clicando componente por componente é uma tarefa complicada, demorada e fica inviável a medida que projeto cresce. Além do fato que uma nova linha de código adicionada no projeto pode fazer com que todo o *software* pare de funcionar. (CORBUCCI; ANICHE, 2013)

Para tentar sanar esses problemas e viabilizar que o *software* ainda seja testado é preciso mudar a abordagem para uma abordagem que facilite a forma como o desenvolvedor

irá testar seu código, duas dessas abordagens comumente usadas por equipes ágeis são: TDD e BDD.

2.2 Metodologias Ágeis

Visando minimizar problemas do desenvolvimento de *software* sem algum planejamento é preciso adotar uma metodologia que ajude no processo de planejamento e na solução a ser desenvolvida.

Segundo (FOWLER, 2003) citado por (PASSUELO, 2005) "Metologias impõem um processo disciplinado no desenvolvimento de *software*, com objetivo de torná-lo mais previsível e mais eficiente".

Porém o problema de muitas metodologias é a burocrácia. Esas metodologias são focadas em documentação e gerenciamento rígido do projeto, muitas vezes acabam sendo vistas como um grande atraso para o projeto, já que a grande maioria do tempo do projeto será gasto levantando requisitos e documentando todas as ações do sistema, antes mesmo de ele ser codificado. (FOWLER, 2003)

Então nos anos noventa, surge uma reação a essas metodologias mais rígidas, são as chamadas metodologias ágeis no processo de desenvolvimento de *software*.

Criado por Kent Beck, com uma equipe de desenvolvedores experientes, o Manifesto Ágil é o documento que reúne todas as práticas e princípios dos métodos ágeis dentro do processo de desenvolvimento (BECK, 2001). O manifesto Ágil possui valores, que são eles:

- **Indivíduos e interações** mais que processos e ferramentas;
- **Software em funcionamento** mais que documentação detalhada;
- **Colaboração com cliente** mais que negócios e contratos;
- **Responder a mudança** mais que seguir um plano;

Ou seja, mesmo havendo valor nos itens a direita, valorizamos mais os itens a esquerda. (ÁGIL, 2001)

Além dos valores, o Manifesto Ágil se baseia em alguns princípios:

- Satisfazer o cliente por meio de entregas adiantadas e contínuas do *software*;

- Aceitar as mudanças de requisito a qualquer momento dentro do projeto;
- Entrega de *software* em funcionamento com frequência semanal ou mensal;
- Pessoas que entendem do negócio e desenvolvedores devem estar sempre trabalhando juntas durante o processo de desenvolvimento;
- Trabalhar com pessoas motivadas no projeto, dando a eles um bom ambiente de trabalho e confiar que farão aquele que foi designado a eles;
- A melhor maneira de se transmitir informação é por meio de uma conversa pessoal;
- *Software* funcional é a medida primária de progresso;
- Processos ágeis promovem um ambiente sustentável;
- Contínua atenção a excelência técnica e bom design;
- Simplicidade;
- As melhores arquiteturas, requisitos e designs emergem de times auto-organizáveis;
- Em intervalos regulares, o time reflete em como ficar mais efetivo, então, se ajustam e aperfeiçoam seu comportamento de acordo.

Por meio desses princípios e valores pode-se notar que as metodologias ágeis deixam de lado a documentação pesada, processos de extração e elaboração de requisitos e passam a valorizar mais o *software* funcionando, liberando pequenas versões à medida que as mesmas ficam prontas, trazem o cliente para dentro do processo de desenvolvimento fazendo o mesmo participar e acompanhar o crescimento do projeto, enfatiza interação entre pessoas sejam elas do time de desenvolvimento ou pessoas que entendem do negócio e abraçam mudanças no *software* a qualquer momento (FOWLER, 2003) (BECK, 2001).

2.3 Extreme Programming (XP)

A *Extreme Programming* mais conhecida com XP é uma metodologia de desenvolvimento que tem como finalidade entregar *software* de qualidade e com as necessidades que o cliente realmente precisa.

É uma metodologia voltada para pequenos e médios times de desenvolvimento que trabalham em projetos orientados a objetos e que seus requisitos podem sofrer constantes mudanças. (KUHN, 2008)

A satisfação do cliente é o que faz a metodologia ser bem sucedida. O XP busca o máximo desempenho da equipe mantendo um ritmo saudável de trabalho onde horas extra só são realizadas se forem agregar algo de valor ao produto final.

O cliente obterá seu produto em pequenos lançamentos, podendo utilizá-lo, aprender com o mesmo e avaliar se o que foi desenvolvido era o desejado. (ÁGIL, 2013a)

(ÁGIL, 2013a) menciona os valores seguidos pelo XP:

- **Feedback:** Quanto mais cedo um problema for descoberto, menos prejuízo e atraso ao *software* ele dará e maiores são chances de ele ser resolvido rapidamente;
- **Comunicação:** Das várias formas de se comunicar existentes, times que utilizam XP prezam por uma conversa presencial para melhorar a compreensão e entendimento entre os membros;
- **Simplicidade:** Embora muitas vezes no desenvolvimento é um hábito desenvolver pensando em futuras implementações, no XP o time é focado em desenvolver o que é necessário para agora, entregando o que realmente o cliente necessita para o momento, deixando o que vier depois, para depois;
- **Coragem:** Equipes de desenvolvimento precisam encarar mudanças constantes nos projetos e para isso é necessário coragem e confiança em suas práticas;
- **Respeito** É o valor mais básico e serve de base a todos, pois dará confiança e respeito a toda equipe durante o processo.

Aliado aos valores e princípios a metodologia XP também possui uma série de práticas para descrever quais atitudes a equipe deve tomar no ambiente de trabalho e como se deve ocorrer o processo de desenvolvimento pela equipe (ÁGIL, 2013a).

- **Design simples e incremental:** O desenvolvimento da aplicação será de maneira iterativa e incremental, ou seja, a equipe se focará em desenvolver os *software* do modo mais simples possível, dando valor e codificando o que é realmente necessário para o cliente e permitindo o *software* crescer aos poucos;

- **Programação em pares:** É uma prática que visa nivelar o trabalho da equipe, sugerindo que dois programadores trabalhem juntos de forma colaborativa na mesma tarefa, enquanto um foca sua atenção na criação do código o outro terá o trabalho de revisar e manter o primeiro focado na solução mais simples;
- **Código de posse coletiva:** Em um projeto todos tem responsabilidade e liberdade de alterar todo e qualquer código encontrado, o código não pertence a ninguém e sim a todo mundo. Isso ocorre quando equipes trocam de membro de forma constante e abordando o código dessa maneira é mais fácil transmitir o conhecimento do projeto, sem atrelar determinados códigos ou funcionalidades a determinados desenvolvedores;
- **Desenvolvimento guiado por testes:** É a prática que propõem a criação dos testes antes mesmo do código de produção ser implementado, tem como objetivo maior detectar e solucionar todas as possíveis falhas do desenvolvimento por meio dos testes. Dessa forma, quando um código é terminado não é preciso voltar até ele para testá-lo, pois ele já estará testado e pronto.

2.4 Scrum

Após a criação do Manifesto Ágil, surgiram várias metodologias que visavam tirar proveito dos valores e princípios, em 1990 surge o *Scrum* que segue a seguinte definição:

”*Scrum* é um *framework* Ágil, simples e leve, utilizado para a gestão do desenvolvimento de produtos complexos imersos em ambientes complexos. *Scrum* é embasado no empirismo e utiliza uma abordagem iterativa e incremental para entregar valor com frequência e, assim, reduzir os riscos do projeto”. (SABBAGH, 2013)

Embora o *Scrum* vise ser menos burocrático que outras metodologias, ele trás consigo uma série de definições e papéis para melhor organizar sua equipe e seus processos. Porém pelo fato de ser definido como um *framework*, não há um processo rígido fixo e sim um processo que pode ser adaptado para cada projeto e cada equipe.

Em sua forma mais básica o *Scrum* é composto por um time, seus artefatos e seus eventos. O processo de desenvolvimento é baseado em *Sprints*, onde o time irá interagir e trabalhar no projeto. (ÁGIL, 2013b) (SABBAGH, 2013)

A visão geral do processo de *Sprint* pode ser vista na Figura 2.1 abaixo:

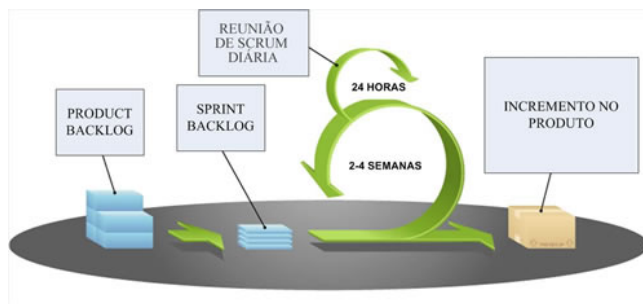


Figura 2.1: Visão geral de uma *Sprint* dentro do *Scrum* (JAMIL, 2013)

2.4.1 O time do Scrum

Segundo (SABBAGH, 2013) dentro do *Scrum* existem três papéis definidos: o Time de desenvolvimento, *Scrum Master*.

- **Time de desenvolvimento:** Geralmente é uma equipe composta de 3 a 10 pessoas e são os responsáveis pela entrega do produto. Formam um time auto-organizado, multidisciplinar, motivado e com foco em qualidade. Dentro do time não há divisões entre programadores, *designers*, engenheiros ou analistas;
- **Product Owner:** É o responsável pela visão do produto, por tomar as decisões de negócio, definir e priorizar as atividades do *Product Backlog*, ele representa a necessidade do cliente em relação ao produto que está sendo desenvolvido;
- **Scrum Master:** É o responsável por garantir o funcionamento e execução do *Scrum* no decorrer de uma *Sprint*. Possui além de conhecimentos técnicos, habilidade de gerenciar pessoas, técnicas para facilitar o trabalho e está sempre tentando remover qualquer bloqueio que possa impedir o time de desenvolvimento.

Embora existam essas três separações, toda equipe é igualmente responsável e responsabilizada pelo resultados obtidos, isso faz com que todos se comprometam com o projeto inteiramente.

Todos os membros juntos, formam o chamado Time do *Scrum*, todos visam trabalhar juntos de forma colaborativa, para que como equipe todos consigam alcançar um único objetivo coletivo. (SABBAGH, 2013) (SCHWABER, 2013)

2.4.2 Artefatos do Scrum

Os artefatos do *Scrum* servem como ferramentas de apoio a todo o time, afim de que todos possam ter um acompanhamento geral dos trabalho durante a *Sprint* e uma visão geral do andamento do produto que está sendo feito.

Conforme (ÁGIL, 2013b) e (ÁGIL, 2013b) os artefatos mais comuns são:

- **Product Backlog:** É lista de funcionalidades desejadas para o produto que está sendo desenvolvido. É responsabilidade do *Produto Owner* adicionar, remover, priorizar todas as tarefas dentro do *Product Backlog*;
- **Sprint Backlog:** É lista de funcionalidades extraídas de dentro do *Product Backlog* que serão desenvolvidas durante uma *Sprint* pelo time de desenvolvimento. Os itens são extraídos de acordo com a priorização do *Product Owner* e a equipe tem a liberdade de escolher os itens que são inclusos;
- **User Story:** É descrição de uma real necessidade do projeto, ou seja, uma funcionalidade esperada no sistema. É escrita do ponto de vista do usuário e ele deverá descrever o que deseja fazer e como deseja fazer.
 - **Como um** - papel dentro do *software*;
 - **Eu quero** - funcionalidade que espera executar;
 - **Para que** - objetivo para o qual essa funcionalidade será criada.

2.4.3 Eventos do Scrum

Eventos do *Scrum*, são na verdade o próprio ciclo de desenvolvimento, denominado *Sprint*. Durante o período da *Sprint* a equipe faz um série de cerimônias com objetivo de atualizar todos os membros sobre o andamento do trabalho e também ajuda a detectar algum possível impedimento para o escopo da *Sprint*. (SCHWABER, 2013) (SABBAGH, 2013)

Segundo (SCHWABER, 2013) e (SABBAGH, 2013) o principais eventos dentro do *Scrum*, são:

- **Sprint:** É um mini-projeto, que tem duração entre 1 e 4 semanas e é durante esse intervalo de tempo que o Time de desenvolvimento faz o incremento do produto. O

objetivo maior da *Sprint* é que ao seu final uma nova parte utilizável do software seja entregue.

- ***Sprint Planning:*** É uma reunião com participação de todos os envolvidos onde o *Product Owner* mostra quais as atividades de maior prioridade naquele momento e a equipe irá selecionar quais dessas atividades entram na *Sprint* para gerar um novo incremento no *software*;
- ***Dayli Scrum:*** É uma reunião curta e realizada todos os dias pela equipe de desenvolvimento, geralmente ocorre sempre no mesmo horário e tem uma duração de aproximadamente 15 minutos. Durante os *Dayli Scrums* todos os membros da equipe se focam em responder três perguntas:
 - O que eu fiz ontem ?;
 - O que eu farei hoje ?;
 - Algum impedimento para o que eu fiz ontem ou terei que fazer hoje ?
- ***Sprint Review:*** é uma reunião feita com toda a equipe ao final de uma *Sprint*. Tem como objetivo avaliar tudo que foi feito ao longo da *Sprint*, se o que foi feito está de acordo com o que foi combinado na reunião de *Sprint Planning* e se irá entrega um novo incremento ao *software*;
- ***Sprint Retrospective:*** É uma análise feita entre o a *Sprint Review* e a próxima *Sprint Planning*, serve para avaliar o que foi feito na última *Sprint*, quais pontos do processo foram falhos, quais ajudaram a equipe a ter uma melhor performance e quais pontos podem ser melhorados para otimização da equipe.

2.5 Qualidade de Software

Segundo (PRESSMAN, 2011)

Conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo *software* profissionalmente desenvolvido.

A qualidade de *software* não pode estar ligada somente ao *software* fazer algo certo, como foi pedido pelo cliente e também não se restringe ao *software* final criado. Qualidade

de *software* está ligada também a qualidade do código, reusabilidade, manuteabilidade, aos processos adotados durante o desenvolvimento como análise de requisitos e documentação.

Existem fatores para determinar a qualidade de *software* (PRESSMAN, 2011):

- **Corretitude:** O *software* satisfaz as especificações determinadas e cumpre os objetivos desejados pelo cliente;
- **Confiabilidade:** O *software* executar determinada funcionalidade com a precisão esperada;
- **Eficiência:** Quantidade de recursos computacionais e de código exigida pelo *software* para executar uma funcionalidade;
- **Integridade:** Controlar acesso ao *software* ou a dados a pessoas não autorizadas;
- **Usabilidade:** Esforço para aprender, operar e preparar a entrada e interpretar uma saída do *software*;
- **Manuteabilidade:** Esforço para localizar e reparar erros no *software*;
- **Flexibilidade:** Esforço para modificar um programa operacional;
- **Testabilidade:** Esforço exigido para testar um *software* a fim de garantir que ele execute uma funcionalidade pretendida;
- **Portabilidade:** Esforço exigido para transferir o programa de um ambiente de sistema de *hardware/software* para outro;
- **Reusabilidade:** À medida que um *software* cresce possibilita algumas de suas partes serem reutilizadas em outros ou até no mesmo *software*;
- **Interoperabilidade:** Esforço exigido para se acoplar um sistema a outro.

Essas são os fatores propostos por McCall, Richards e Walters (1977) e citados por (PRESSMAN, 2011). Esses fatores podem ser agrupados e representados conforme a Figura 2.2.



Figura 2.2: A Pirâmide de qualidade de *software*
(CAMPOS, 2012)

2.6 Testes de Software

A evolução tecnológica vivenciada por nós nos dias atuais nos traz uma comodidade muito grande, o uso de *notebooks* e *smartphones* facilita cada vez mais as atividades e nossa comunicação. Meios de transporte como carros e aviões também estão cada vez mais dependentes de tecnologia e *softwares* para facilitar a atividade e trazer uma maior segurança, além desses, sistemas bancários e telefônicos possuem grandes *softwares* e necessitam estar sempre em funcionamento.

O lado negativo de todo esse avanço, é que nos torna refens da tecnologia e uma falha gerada por algum desses dispositivos pode ser catastrófica e trazer grandes prejuízos financeiros.

Ao longo dos anos a indústria de *software* acumula algumas grandes falhas, citadas por (COSTA, 2012) e (TI, 2012) :

- **Apagão nos EUA:** Um incidente causado por uma falha no sistema de alarmes deixou mais de 50 milhões de pessoas sem luz e causou 11 mortes;
- **Recall da Honda:** uma falha de programação no sistema de *airbag* dos carros, fez com que a *Honda*, fosse forçada a fazer um *recall* de mais de 2 milhões de automóveis

e custou milhões a empresa japonesa;

- **Radiação em excesso:** Entre os anos de 1985 e 1987, hospitais dos EUA utilizavam um aparelho chamado *Therac-25* para o tratamento com radiação contra o câncer, um erro de programação no *software* fazia com que a máquina aplicasse uma dosagem até 100 vezes maior do que a indicada para os pacientes, foram registradas 6 mortes por conta dessa falha;
- **Foguete Mariner:** uma fórmula matemática falha introduzida por um programador fez com que o foguete fosse para fora do seu curso, causando um prejuízo de 18 milhões de dólares;
- **Míssil na guerra do Golfo:** durante a guerra, um sistema de mísseis americanos falhou na interceptação de um míssil inimigo e o acampamento dos soldados foi destruído causando 28 mortes;

Essas falhas colocam como cada vez mais prioritária a necessidade de se garantir a qualidade e funcionamento do *softwares* através de testes bem feitos, melhor elaborados e mais precisos.

Teste de *software* pode ser definido como um processo de execução de um produto, para determinar se o mesmo atingiu suas necessidades e se seu funcionamento ocorreu de forma correta e segura. O maior objetivo do processo de testes é detectar falhas e preveni-las ainda durante o desenvolvimento, para evitar que essas falhas sejam descobertas por usuários e não causem danos e nem prejuízos. (PRESSMAN, 2011)

Segundo (PRESSMAN, 2011) testes de *software* devem abordar todos os níveis do projeto e podem ser classificados como:

- **Testes de Unidade:** Focam em testar cada pequeno componente do sistema, garantindo sua funcionalidade e também garantem que cada pequeno componente consiga funcionar de forma isolada;
- **Teste de Integração:** Visa testar o fluxo do sistema como um todo e garantir que todas as interações funcionem entre si;
- **Teste de Validação:** Visa testar se os requisitos estão funcionando da mesma forma no *software* de como foram levantados na análise;

- **Teste de Sistema:** Foca em testar a combinação do *software* com outros elementos de *hardware*, banco de dados e usuários reais. Também visa garantir que o projeto atendeu a necessidade para o qual ele foi criado.

(PRESSMAN, 2011) ainda sugere que os teste podem ser vistos em forma de espiral conforme a Figura 2.3, partindo da menor unidade do *software* até sua integração total e seu uso.



Figura 2.3: O espiral dos testes de *software* (PRESSMAN, 2011)

Dessa forma é possível notar que a atividade de testar *software* deve se tornar cada vez mais importante nos projetos, para garantir não só sua qualidade, mas também eliminar riscos de falhas do projeto e garantir a satisfação do cliente.

2.7 Test Driven Development (TDD)

O TDD (*Test Driven Development* ou Desenvolvimento Guiado por Testes) é a prática de desenvolvimento de *software* onde os testes automatizados são criados antes de qualquer código real seja implementado, deixando a codificação real para um segundo momento, forçando o desenvolvedor a focar sempre na simplicidade, refatorando seu código sempre que possível e buscando deixar o código sempre enxuto. (CORBUCCI; ANICHE, 2013)

A utilização dessa abordagem possibilita ao desenvolvedor tenha ao final do processo uma aplicação funcional, com uma estrutura bem feita e altamente testável. (CORBUCCI; ANICHE, 2013)

(BECK, 2001) enfatiza duas regras principais que devem ser seguidas quando se utiliza

TDD:

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

Segundo (BECK, 2001) a prática do TDD segue um ciclo chamado vermelho-verde-refatorar (*red-green-refactor*) como mostra a Figura 2.4 e pode ser definido da seguinte forma:

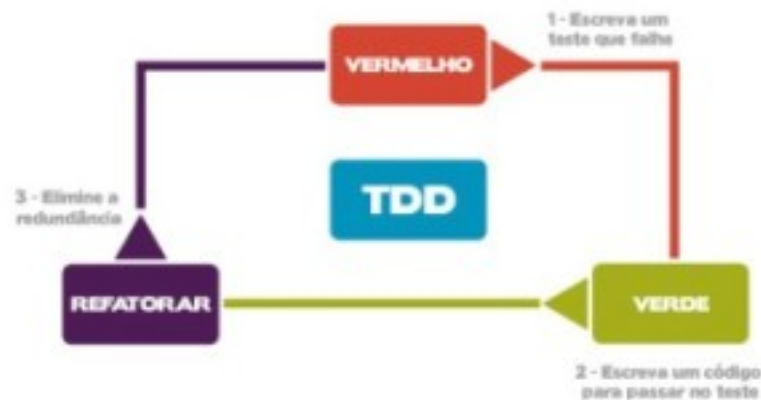


Figura 2.4: O ciclo do TDD
(ROCHA, 2013)

1. **Vermelho:** Desenvolvedor escreve um código que falhe;
2. **Verde:** Desenvolvedor escreve o mínimo de código necessário para que o teste passe, sem se preocupar com a qualidade do que foi codificado, apenas foca que o teste passe;
3. **Refatorar:** Como o teste já está verde, agora o desenvolvedor irá focar na melhoria do código escrito anteriormente.

Segundo (CORBUCCI; ANICHE, 2013) a utilização dessa prática de desenvolvimento trará vantagens ao desenvolvedor, sendo elas:

- **Focar no teste e não na sua implementação:** Nessa abordagem, o desenvolvedor irá focar em codificar apenas aquilo que o teste necessita, dessa forma ele irá focar apenas naquela classe ou trecho de código que está testando, deixando de lado um tempo a sua implementação;

- **Código nasce testado:** Quando o desenvolvedor segue esse ciclo, ele irá garantir que cada pedaço do código final tenha ao menos um teste sobre ele para assegurar seu funcionamento;
- **Simplicidade:** Quando o desenvolvedor aplica o ciclo ele sempre irá se preocupar em fazer o teste passar e irá focar seus esforços em um teste por vez, isso fará com que ele codifique sempre da forma mais simples, deixando de lado soluções mirabolantes ou complexas;
- **Melhor compreensão sobre o *design* das classes:** Como o desenvolvedor irá focar sempre em um teste e uma classe por vez, irá deixar de lado por um momento a integração geral das classes e irá se preocupar em manter simples a classe na qual está trabalhando, aumentando a sua coesão e diminuindo seu acoplamento.

(CORBUCCI; ANICHE, 2013) e (BECK, 2001) também dizem que os testes podem refletir sobre a saúde do código, quando um teste é complicado de se escrever é um sinal que o código pode estar ruim ou não estar fazendo aquilo que realmente deveria fazer.

Dentro das práticas do TDD ainda existe uma prática chamada *baby steps* que pode ser definida com a prática onde o código cresce aos poucos de forma incremental, forçando o desenvolvedor a focar em um método por vez, praticando o ciclo do TDD em cada um dos métodos e classes do projeto.

Segundo (CORBUCCI; ANICHE, 2013) a prática do *baby steps* aliado ao TDD irá proporcionar ao desenvolvedor receber um *feedback* mais rápido e constante do seu código, podendo prevenir e arrumar falhas de codificação de forma imediata diminuindo os custos e prevenindo futuras manutenções no código.

A Figura 2.5 mostra como é a forma de recebimento do *feedback* do código pelo desenvolvedor que pratica o TDD e um desenvolvedor que não pratica TDD.

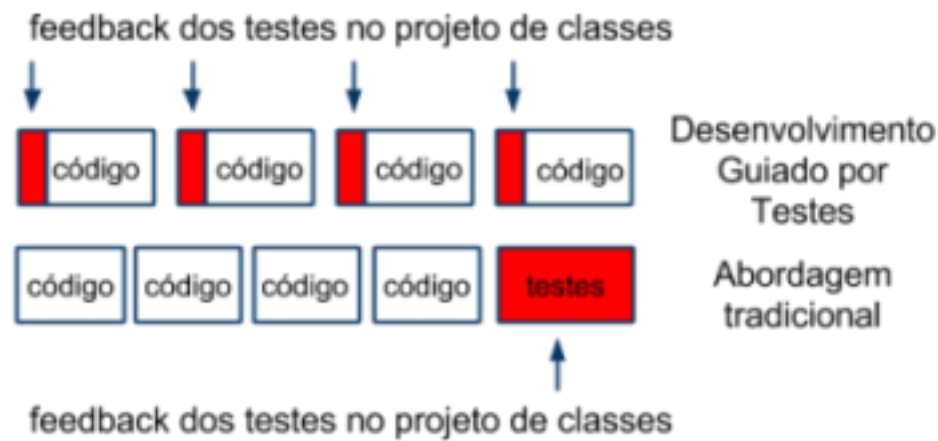


Figura 2.5: *Feedback* do código utilizando TDD
(CORBUCCI; ANICHE, 2013)

Dessa forma é possível notar que a prática do TDD além de garantir um melhor código, criar uma base sólida de testes de unidade ao *software* também irá ajudar na prevenção de erros e quando erros surgirem, o desenvolvedor poderá corrigi-lo mais rapidamente, evitando que esses erros possam se propagar e prejudicar todo o projeto.

2.8 Behavior Driven Development (BDD)

O BDD (*Behavior Driven Development* ou Desenvolvimento Guiado por Comportamento) é uma prática de desenvolvimento de *software* que tenta gerar uma maior integração entre equipes de negócio e desenvolvimento. (BARAÚNA, 2013)

Nessa prática as equipes irão se focar em descrever um comportamento esperado pelo *software* para após isso desenvolver o código necessário que irá solucionar o cenário desse comportamento, dessa forma pode-se validar se de fato o código criado irá atender a necessidade de negócio e se comportar de modo assertivo. (BARAÚNA, 2013)

O BDD pode ser dito como uma evolução sobre o TDD, visto que o TDD foca na construção do código, BDD se preocupa em saber se o código criado irá se comportar de forma esperada e se consegue entregar a implementação da funcionalidade de forma esperada.

O BDD fornece uma maneira com que equipes de áreas diferentes consigam trabalhar e escrever os cenários de teste juntas, já que a prática utiliza uma Linguagem Ubíqua na descrição dos seus cenários. (BARAÚNA, 2013) (SOARES, 2011)

O surgimento do BDD começou quando Dan North sentiu dificuldades de ensinar TDD para seus alunos por conta de sua nomenclatura e da utilização da palavra teste. Seus alunos sentiam certa dificuldade em saber por onde começar a especificação. Dan North então se focou em escrever o primeiro *framework* para de utilizar BDD, surgiu então o *JBehave*. Essa nova ferramenta tinha como foco tirar a referência da palavra teste e substituir por um vocabulário onde o foco fosse o comportamento. (BARAÚNA, 2013) (NORTH, 2006)

(SANCHEZ, 2006) cita as principais alterações feitas por Dan North, na tentativa de remover o conceito de teste e focar na descrição do comportamento:

- Uso de **Context** ao invés de *TestCase*;
- Uso de **Specification** ao invés de *Test*;
- Uso de **Should** ao invés de *Assert*.

Além dessas, também foram introduzidas novas formas de criar cenários e validar seus critérios de aceitação (BARAÚNA, 2013):

- **Feature** que servirá para especificar e testar uma funcionalidade;
- **Scenario** irá descrever o cenário para qual o teste deve cobrir e quais seus critérios de aceitação;
- **Steps** um cenário é composto por vários *steps*, esses *steps* irão descrever o que se espera do *software* e servirão de critério de aceitação.

Uma especificação de comportamento no BDD é descrita de forma muito similar a uma *User Story* do *Scrum* e por isso é uma prática dentro de times *Scrum* a utilização do BDD. Outro ponto importante da utilização do BDD no processo de desenvolvimento é que as especificações ao longo do tempo, acabam se tornando uma documentação viva do sistema. (SANCHEZ, 2006)

De forma simples, uma especificação é composta da seguinte maneira:

- **Como um** - papel dentro do *software*;
- **Eu quero** - funcionalidade que espera executar;
- **Para que** - objetivo para o qual essa funcionalidade será criada.

E para a descrição do cenário do comportamento:

- **Dado** - é o contexto inicial;
- **Quando** - ocorre a execução de alguma atividade;
- **Então** - descreve qual o resultado esperado para essa execução.

A abordagem de desenvolvimento utilizando BDD propõe que o *software* seja desenvolvido especificando as várias camadas do comportamento com testes automatizados seguindo uma abordagem chamada de *outside-in development*, ou seja, deve-se começar com a especificação de um comportamento de uma funcionalidade, após isso codificar as classes e métodos necessárias para satisfazer esse comportamento. (BARAÚNA, 2013)

Na Figura 2.6 visa ilustrar a forma de desenvolvimento proposta pelo BDD:

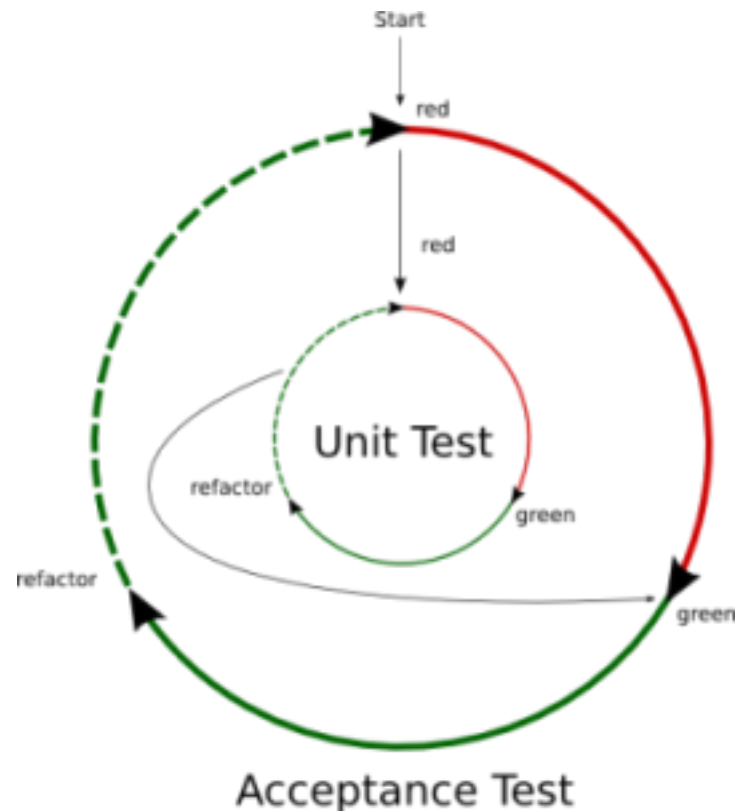


Figura 2.6: O ciclo do BDD
(SOFTWARE, 2013)

Segundo (BARAÚNA, 2013) e (SANCHEZ, 2006) as etapas do processo de desenvolvimento seguindo o BDD ocorrem da seguinte forma:

- Na primeira etapa do ciclo a equipe deve escrever um novo comportamento esperado pelo *software* e ao executar essa especificação ela deve falhar;
- Na segunda etapa do ciclo a equipe entra no ciclo do TDD, escrevendo um teste de unidade que também deverá falhar;
- Na terceira etapa do ciclo deve-se codificar o mínimo necessário de código para que o teste feito na segunda etapa passe;
- Na quarta etapa o código escrito na terceira etapa deverá ser refatorado e após a refatoração é necessário certificar-se de que o teste de unidade escrito na etapa dois, continue passando;
- Na quinta e última etapa do ciclo, após o código ter sido codificado e refatorado é necessário executar novamente a especificação e verificar que o comportamento está sendo executado corretamente.

Com o desenvolvimento seguindo essa abordagem é possível verificar como o BDD utiliza o TDD, porém eleva o nível do teste para o nível de especificação, saindo do escopo de apenas estar testando código. Permite então, que a equipe realmente veja que o código feito atende a necessidade, funciona corretamente e implementa a funcionalidade de forma esperada.

2.9 Linguagem Ruby

A linguagem foi criada no ano de 1995 no Japão Yukihiro "Matz" Matsumoto, é uma linguagem orientada a objeto interpretada, de tipagem forte e dinâmica. Foi criada baseada no que havia de melhor em outras linguagens como: *Smalltalk*, *Lisp* e *Ada*. Tudo em *Ruby* é objeto e sua sintaxe simples visa facilitar o desenvolvedor tanto na hora de escrever código como também na hora de ler código de outros desenvolvedores.

É *open-source* o que possibilita aos desenvolvedores abrirem e melhorem o código fonte sempre que julgarem necessário, o que possibilita uma constante na correção de erros, facilitando melhorias e inclusões de novas funcionalidades a linguagem. (RUBY; THOMAS; HANSSON, 2013)

Na Figura 2.7 é possível ver um exemplo de código Ruby:


```

2.2.2 :003 > number = 10
=> 10
2.2.2 :004 > name = 'Luiz'
=> "Luiz"
2.2.2 :005 > number.class
=> Fixnum
2.2.2 :006 > name.class
=> String
2.2.2 :007 > number + class
SyntaxError: (irb):7: syntax error, unexpected end-of-input
      from /home/cezinha/.rvm/rubies/ruby-2.2.2/bin/irb:11:in `<main>'
2.2.2 :008 > number + 10
=> 20

```

Figura 2.7: Exemplo de código Ruby
 Autoria Própria

Na primeira linha da Figura 2.7 está sendo criado uma nova variável chamada *number* e sendo atribuído um valor 10, como *Ruby* possui uma tipagem dinâmica e 10 é um valor de tipo número inteiro, o interpretador irá entender que a variável *number* será do tipo *Fixnum*.

O mesmo ocorre para a variável *name*, que foi criada recebendo uma *string* "Luiz", o interpretador irá entender que *name* é um valor do tipo *String*

Como em Ruby tudo é um objeto é possível fazer a checagem de qual tipo de objeto as variáveis *number* e *name* pertencem, nesse caso *Fixnum* e *String* respectivamente.

Como *Ruby* possui uma tipagem forte, ele não irá permitir que as duas variáveis, um *Fixnum* e uma *String*, executem a operação de soma, exibindo um erro.

Capítulo 3

Materiais e Métodos

Este capítulo tem como objetivo descrever as ferramentas e tecnologias utilizadas no desenvolvimento da aplicação experimental.

3.1 Tecnologias Utilizadas

Para o desenvolvimento do estudo experimental foram utilizadas as seguintes tecnologias e ferramentas:

- Linguagem *Ruby*;
- *Framework RSpec*;
- *Framework Cucumber*;
- *Framework Web Sinatra*;
- Editor de texto *Sublime Text*;

A linguagem *Ruby* utilizada no estudo experimental se encontra na versão 2.2.2, sendo essa a sua última versão estável.

O *RSpec* é um *framework* para criação de testes unitários que segue a filosofia do BDD.

O *Cucumber* é um *framework* para automatizar a criação de testes de aceitação e validar o comportamento do *software*, com ele é possível escrever especificações executáveis de do ponto de vista do usuário.

O editor *Sublime Text* foi escolhido por ser um editor simples, leve e permite a instalação de diversos *plugins* para diversas linguagens, para facilitar o seu uso.

E o *Sinatra* é um *micro-framework web* escrito em *Ruby* que tem como grande vantagem ser leve e possibilitar um desenvolvimento flexível e rápido para aplicações *web* de pequeno porte.

3.2 Estudo Experimental

Nesta seção será descrito o procedimento da aplicação experimental para validar a utilização do BDD no processo de desenvolvimento.

O estudo experimental proposto, será uma aplicação que irá realizar o cálculo do salário líquido, conforme Tabela 3.1 do Imposto de Renda (IR) e Tabela 3.2 referente ao Instituto Nacional do Seguro Social (INSS).

Tabela 3.1: Tabela do Cálculo do Imposto de Renda - 2015

Base de cálculo	Alíquota em %	Parcela a deduzir em R\$
Até 1.903,98	-	-
De 1.903,99 até 2.826,65	7,5	142,80
De 2.826,66 até 3.751,05	15,0	354,80
De 3.751,06 até 4.664,68	22,5	636,13
Acima de 4.664,68	27,5	869,36

Tabela 3.2: Tabela do Cálculo do INSS - 2015

Base de cálculo	Alíquota em %
Até 1.399,12	8
De 1.399,13 até 2.331,88	9
De 2.331,89 até 4.663,75	11

O propósito da aplicação é que dado um valor de salário líquido, a aplicação deverá retornar a alíquota e o valor em reais referente ao IR e ao INSS do valor informado.

Na primeira etapa é necessário criar a estrutura inicial da aplicação *web* conforme Figura 3.1.

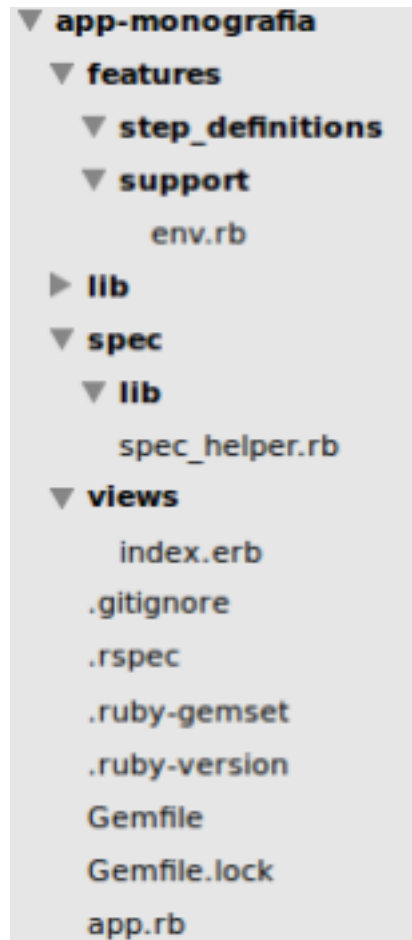


Figura 3.1: Estrutura inicial da aplicação
Autoria Própria

Os principais itens dessa estrutura são:

- **features:** É o diretório onde estarão os testes de aceitação;
- **features/step-definitions:** É o diretório onde serão escritos os passos para validação do teste de aceitação;
- **features/support:** É o diretório onde serão escritos códigos que poderão ser utilizados de forma à auxiliar os testes de aceitação;
- **lib:** É o diretório onde está o código da aplicação;
- **spec/lib:** É o diretório responsável por armazenar os testes de unidade dos códigos escritos dentro de *lib*;

- **views:** É o diretório onde as telas do projeto estarão armazenadas;
- **app.rb:** É o arquivo que inicia a aplicação;

O passo seguinte é configurar os *frameworks* de testes que serão utilizados, conforme Figura 3.2

```
1 source 'https://rubygems.org'
2
3 gem 'sinatra'
4
5 group :test do
6   gem 'rspec'
7   gem 'shoulda-matchers'
8   gem 'cucumber'
9 end
```

Figura 3.2: Configuração dos *frameworks* de teste
Autoria Própria

Com a aplicação devidamente configurada, é possível iniciar o desenvolvimento.

O *framework Cucumber*, que será utilizado para a criação das especificações possui um padrão de linguagem chamado *Gherkin* que possui um vocabulário próprio e permite que as especificações sejam escritas na linguagem nativa da equipe ou desenvolvedor.

O primeiro passo para o desenvolvimento é criar um primeiro teste de aceitação, para validar os cálculos que serão efetuados pela aplicação, conforme visto na Figura 3.3:

```

1  # language: pt
2
3  Funcionalidade: Calcular salário líquido
4  Como um usuário
5  Quero poder calcular meu salário líquido
6  Para saber qual será meu ganho real
7
8  Cenário: Salário R$ 1.200,00 reais é isento de IRRF e 8% de INSS.
9  Dado que informo meu salário bruto de '1200.00'
10 E cliço para calcular o meu salário líquido
11 Então vejo a mensagem IRRF - Aliquota (%): 'Isento'
12 Então vejo a mensagem IRRF - Valor: R$ '0.00'
13 Então vejo a mensagem INSS - Aliquota (%): '8%%'
14 Então vejo a mensagem INSS - Valor: R$ '96.00'
15 Então vejo a mensagem Salário Líquido: R$ '1104.00'

```

Figura 3.3: Criando primeiro teste de aceitação
Autoria Própria

Após criado o arquivo de especificação é possível executá-lo, com o comando *bundle exec cucumber*, porém como nesse momento só existe o código da especificação, não será possível validar a aplicação e ele apenas irá informar que a especificação ainda não foi escrita, conforme a Figura 3.4.

```

→ app-monografia git:(master) ✕ bundle exec cucumber
# language: pt
Funcionalidade: Calcular salário líquido
  Como um usuário
  Quero poder calcular meu salário líquido
  Para saber qual será meu ganho real

  Cenário: Salário R$ 1.200,00 reais é isento de IRRF e 8% de INSS.
    Dado que informo meu salário bruto de '1200.00'
    E cliço para calcular o meu salário líquido
    Então vejo a mensagem IRRF - Aliquota (%): 'Isento'
    Então vejo a mensagem IRRF - Valor: R$ '0.00'
    Então vejo a mensagem INSS - Aliquota (%): '8%%'
    Então vejo a mensagem INSS - Valor: R$ '96.00'
    Então vejo a mensagem Salário Líquido: R$ '1104.00'

1 cenário (1 undefined)
7 steps (7 undefined)
0m0.017s

```

Figura 3.4: Executando primerio teste de aceitação
Autoria Própria

A saída do comando também irá sugerir que sejam copiados os passos do teste sugeridos e crie-se um arquivo *Ruby* no diretório *features/step-definitions* contendo todos os passos da especificação, conforme a Figura 3.5.

```

You can implement step definitions for undefined steps with these snippets:

Dado(/^que informo meu salário bruto de '(\d+)\.(\d+) '$/) do |arg1, arg2|
  pending # Write code here that turns the phrase above into concrete actions
end

Dado(/^clico para calcular o meu salário líquido$/ do
  pending # Write code here that turns the phrase above into concrete actions
end

Então(/^vejo a mensagem IRRF \- Aliquota \(%\): 'Isento'$/ do
  pending # Write code here that turns the phrase above into concrete actions
end

Então(/^vejo a mensagem IRRF \- Valor: R\$ '(\d+)\.(\d+) '$/) do |arg1, arg2|
  pending # Write code here that turns the phrase above into concrete actions
end

Então(/^vejo a mensagem INSS \- Aliquota \(%\): '(\d+)\%'$/ do |arg1|
  pending # Write code here that turns the phrase above into concrete actions
end

Então(/^vejo a mensagem INSS \- Valor: R\$ '(\d+)\.(\d+) '$/) do |arg1, arg2|
  pending # Write code here that turns the phrase above into concrete actions
end

Então(/^vejo a mensagem Salário Líquido: R\$ '(\d+)\.(\d+) '$/) do |arg1, arg2|
  pending # Write code here that turns the phrase above into concrete actions
end

```

Figura 3.5: Passos do teste de aceitação ainda pendentes
 Autoria Própria

Todos os arquivos contidos no diretório *features/step-definitions* são os arquivos que farão a conexão entre o arquivo de especificação e as classes *Ruby* com o código real da aplicação. Esses arquivos entendem e conseguem receber dados dos arquivos de teste de aceitação e também conseguem se conectar com os arquivos de código da aplicação.

É esse ponto que o uso de BDD visa facilitar a integração entre as equipes, pois essas especificações podem ser escritas em conjunto pela equipe e ao final do processo para garantir o correto funcionamento do *software* basta a equipe toda se focar na execução e validação dessas especificações.

Após copiar e colar em um arquivo *Ruby* os passos sugeridos pelo *Cucumber* para o teste, é preciso iniciar o desenvolvimento desses passos, utilizando os códigos reais da aplicação. Conforme Figura 3.6 os passos que antes eram pendentes agora irão possuir código que será utilizado para a validação.

```

1  Dado(/^que informo meu salário bruto de '(.)'$/ do |total_salary|
2    total_salary = total_salary.to_f
3    @salary_calculator = SalaryCalculator.new(total_salary)
4  end
5
6  Dado(/^clico para calcular o meu salário líquido$/ do
7    @salary_calculator.calculate
8  end
9
10 Então(/^vejo a mensagem IRRF \- Aliquota \(%\): '(.)'$/ do |expected_result|
11   expect(@salary_calculator.result[:irrf_tax]).to eq(expected_result)
12 end
13
14 Então(/^vejo a mensagem IRRF \- Valor: R\$ '(.)'$/ do |expected_result|
15   expect(@salary_calculator.result[:irrf_value]).to eq(expected_result.to_f)
16 end
17
18 Então(/^vejo a mensagem INSS \- Aliquota \(%\): '(.)'$/ do |expected_result|
19   expect(@salary_calculator.result[:inss_tax]).to eql(expected_result)
20 end
21
22 Então(/^vejo a mensagem INSS \- Valor: R\$ '(.)'$/ do |expected_result|
23   expect(@salary_calculator.result[:inss_value]).to eql(expected_result.to_f)
24 end
25
26 Então(/^vejo a mensagem Salário Líquido: R\$ '(.)'$/ do |expected_result|
27   expect(@salary_calculator.result[:real_salary]).to eq(expected_result.to_f)
28 end
29

```

Figura 3.6: Criando os testes de aceitação reais
Autoria Própria

Ao executar esses testes de aceitação eles devem falhar pois ainda não foi feito criado código *Ruby* na aplicação para que os testes passem. Na Figura 3.7 é possível observar a execução de todos os passos através do comando *bundle exec cucumber* e verificar que todos os passos estão falhando.

```

→ app-monografia git:(master) ✖ bundle exec cucumber
# language: pt
Funcionalidade: Calcular salário líquido
  Como um usuário
  Quero poder calcular meu salário líquido
  Para saber qual será meu ganho real

  Cenário: Salário R$ 1.200,00 reais é isento de IRRF e 8% de INSS.
    Dado que informo meu salário bruto de '1200.00'
      uninitialized constant SalaryCalculator (NameError)
      ./features/step_definitions/calcular_salario_1000.rb:3:in `/^q
      features/calcular_salario_1000.feature:9:in `Dado que informo
    E clico para calcular o meu salário líquido
    Então vejo a mensagem Salário Líquido: '1.200,00'
    Então vejo a mensagem IRRF - Aliquota (%): 'Isento'
    Então vejo a mensagem IRRF - Valor: R$ '0,00'
    Então vejo a mensagem INSS - Aliquota (%): '8%'
    Então vejo a mensagem INSS - Valor: R$ '96.00'

Failing Scenarios:
cucumber features/calcular_salario_1000.feature:8 # Cenário: Salário

1 scenario (1 failed)
7 steps (1 failed, 6 skipped)
0m0.018s

```

Figura 3.7: Executando a especificação
Autoria Própria

Dessa forma é encerrada a primeira etapa no ciclo do BDD, onde foi criada uma especificação que falha por não haver nenhum código de execução na aplicação, agora é possível iniciar o desenvolvimento e entrar na segunda etapa do ciclo do BDD e iniciar a codificação utilizando TDD.

Conforme visto na Figura 3.7 os testes de aceitação falham pois a classe *SalaryCalculator* ainda não existe. Obedecendo o ciclo do TDD o primeiro passo é criar um teste unitário que falhe, essa é a fase do vermelho (*red*) dentro do TDD.

Na Figura 3.8 está sendo criado o primeiro teste unitário, esse teste deverá falhar pois ainda não há código dentro *SalaryCalculator* apenas a sua definição.

```

1  require 'spec_helper'
2  require_relative '../lib/salary_calculator.rb'
3
4  describe SalaryCalculator do
5    describe 'respond to methods' do
6      let(:value) { 1000.00 }
7      subject { described_class.new(value) }
8
9      it { expect(subject).to respond_to(:calculate) }
10    end
11  end

```

Figura 3.8: Criando primeiro teste unitário que irá falhar
 Autoria Própria

Na Figura 3.9 está sendo executado o primeiro teste unitário para a classe *SalaryCalculator* e é possível observar que ele está falhando.

```

➔ app-monografia git:(master) ✗ bundle exec rspec

SalaryCalculator
  respond to methods
    example at ./spec/lib/salary_calculator_spec.rb:9 (FAILED - 1)

Failures:

  1) SalaryCalculator respond to methods
     Failure/Error: subject { described_class.new(value) }
     ArgumentError:
       wrong number of arguments (1 for 0)
     # ./spec/lib/salary_calculator_spec.rb:7:in 'initialize'
     # ./spec/lib/salary_calculator_spec.rb:7:in 'new'
     # ./spec/lib/salary_calculator_spec.rb:7:in 'block (3 levels)'
     # ./spec/lib/salary_calculator_spec.rb:9:in 'block (3 levels)'

Finished in 0.0007 seconds (files took 0.0797 seconds to load)
1 example, 1 failure

```

Figura 3.9: Executando testes unitários para a classe *SalaryCalculator*
 Autoria Própria

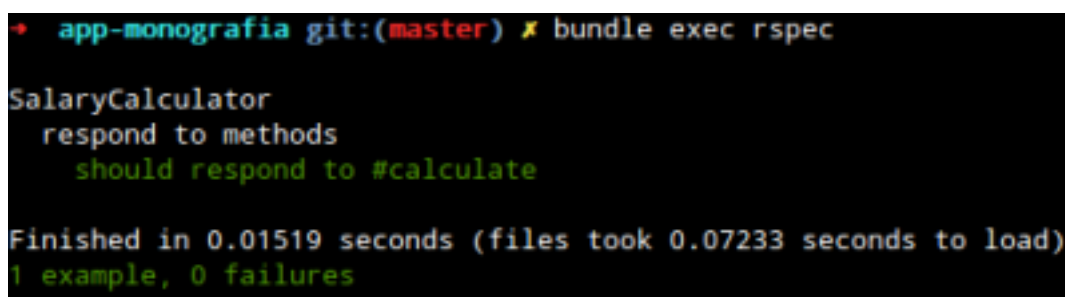
Seguindo na segunda etapa do TDD, conhecida como verde (*green*) é preciso criar um código da forma mais simples possível, apenas para garantir que o teste passe de vermelho para verde. A codificação simplificada é vista na Figura 3.10 e a execução do teste retornando verde é vista na 3.11.

```

1  class SalaryCalculator
2    def initialize(total_salary)
3      @total_salary = total_salary
4    end
5
6    def calculate; end
7  end

```

Figura 3.10: Codificação mínima para a classe *SalaryCalculator*
 Autoria Própria



```

+ app-monografia git:(master) ✗ bundle exec rspec

SalaryCalculator
  respond to methods
    should respond to #calculate

Finished in 0.01519 seconds (files took 0.07233 seconds to load)
1 example, 0 failures

```

Figura 3.11: Executando testes unitários, testes funcionando corretamente
 Autoria Própria

Nesse ponto a terceira etapa do ciclo do BDD foi cumprida, pois o código foi criado utilizando TDD e agora está no estado verde, ou seja, o código atual está funcionando, porém o ciclo não termina até a especificação também estar funcionando corretamente.

O código atual feito na Figura 3.10 está simples o suficiente e está no estado verde. O próximo passo agora seria efetuar a refatoração (*refactor*) para completar o ciclo do TDD, porém o código atual já é o suficiente, não havendo necessidade de refatoração e podendo seguir para o próximo teste.

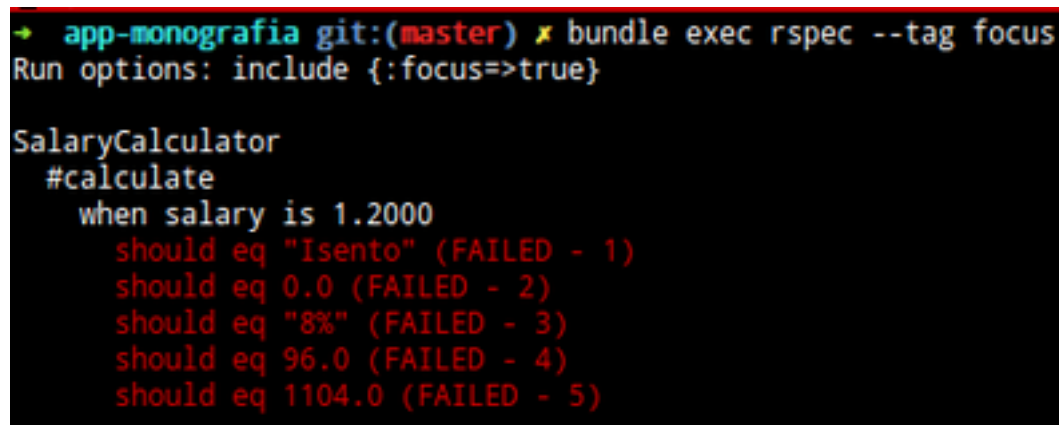
O próximo caso de teste unitário, serão os testes que dado um salário será preciso efetuar os cálculos referentes ao valores de IRRF, INSS e retornar o salário líquido final.

Tomando como base a especificação escrita na Figura 3.3, os testes unitários ficaram da seguinte forma, como visto na Figura 3.12.

```
12 describe '#calculate' do
13   context 'when salary is 1.2000' do
14     let(:value) { 1200.00 }
15     subject { described_class.new(value) }
16
17     before { subject.calculate }
18
19     it { expect(subject.result[:irrf_tax]).to eq('Isento') }
20
21     it { expect(subject.result[:irrf_value]).to eq(0.0) }
22
23     it { expect(subject.result[:inss_tax]).to eq('8%') }
24
25     it { expect(subject.result[:inss_value]).to eq(96.00) }
26
27     it { expect(subject.result[:real_salary]).to eq(1104.00) }
28   end
29 end
```

Figura 3.12: Criando teste unitários de acordo com especificação
Autoria Própria

Na execução dos testes, presentes na Figura 3.13, todos estão falhando, seguindo o princípio do TDD que todos os testes devem falhar em primeiro momento.



```

→ app-monografia git:(master) x bundle exec rspec --tag focus
Run options: include {:focus=>true}

SalaryCalculator
  #calculate
    when salary is 1.2000
      should eq "Isento" (FAILED - 1)
      should eq 0.0 (FAILED - 2)
      should eq "8%" (FAILED - 3)
      should eq 96.0 (FAILED - 4)
      should eq 1104.0 (FAILED - 5)

```

Figura 3.13: Executando testes unitários com base na especificação
Autoria Própria

Na etapa seguinte é preciso realizar a codificação mínima do código para que todos os testes passem. A codificação é vista na Figura 3.14.



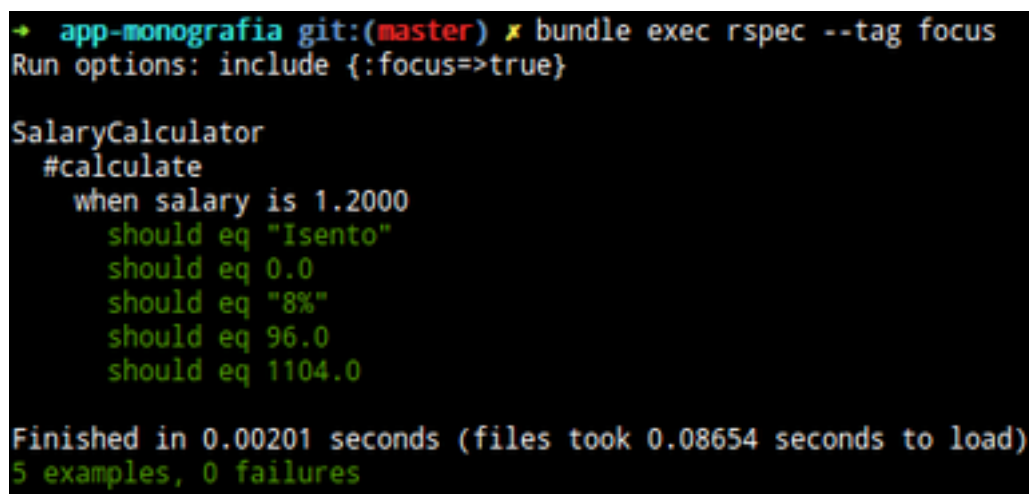
```

1  class SalaryCalculator
2    attr_reader :result
3
4    def initialize(total_salary)
5      @total_salary = total_salary
6      @result = {}
7    end
8
9    def calculate
10     if total_salary <= 1200.00
11       @result[:irrf_tax] = 'Isento'
12       @result[:irrf_value] = 0.0
13       @result[:inss_tax] = '8%'
14       @result[:inss_value] = total_salary * 0.08
15       @result[:real_salary] = total_salary - @result[:inss_value]
16     end
17   end
18
19   private
20
21   attr_reader :total_salary
22 end

```

Figura 3.14: Criando código mínimo para que os testes passem
Autoria Própria

No próximo passo é preciso executar novamente os testes unitários referentes na Figura 3.12 e como a codificação mínima foi feita todos os testes passam, como visto na Figura 3.15.

A terminal window with a black background and colorful text. The first line shows a command: `→ app-monografia git:(master) ✖ bundle exec rspec --tag focus`. The second line shows the run options: `Run options: include {:focus=>true}`. The next line is `SalaryCalculator`. Then, `#calculate` is shown. Below it, a `when salary is 1.2000` block contains five `should eq` assertions: `"Isento"`, `0.0`, `"8%"`, `96.0`, and `1104.0`. The final two lines show the execution results: `Finished in 0.00201 seconds (files took 0.08654 seconds to load)` and `5 examples, 0 failures`.

```
→ app-monografia git:(master) ✖ bundle exec rspec --tag focus
Run options: include {:focus=>true}

SalaryCalculator
  #calculate
    when salary is 1.2000
      should eq "Isento"
      should eq 0.0
      should eq "8%"
      should eq 96.0
      should eq 1104.0

Finished in 0.00201 seconds (files took 0.08654 seconds to load)
5 examples, 0 failures
```

Figura 3.15: Executando novamente os testes unitários
Autoria Própria

Agora seguindo para a próxima etapa é preciso realizar refatoração do código, já que o código inicial era apenas para sair do estado de vermelho e atingir o estado verde. Nessa quarta etapa do processo de BDD é preciso entrar no processo de refatoração dentro do TDD e também validar se os testes unitários ainda passam com a nova implementação, a nova implementação é vista na Figura 3.16.

```

9   def calculate
10     calculate_inss
11     calculate_irrf
12   end
13
14   private
15
16   def calculate_inss
17     if total_salary <= 1399.12
18       result[:inss_tax] = '8%'
19       result[:inss_value] = total_salary * 0.08
20     elsif total_salary >= 1399.13 || total_salary <= 2331.88
21       result[:inss_tax] = '9%'
22       result[:inss_value] = total_salary * 0.09
23     elsif total_salary >= 2331.89 || total_salary <= 4663.75
24       result[:inss_tax] = '11%'
25       result[:inss_value] = total_salary * 0.11
26     end
27
28     result[:real_salary] = total_salary - result[:inss_value]
29   end
30
31   def calculate_irrf
32     if result[:real_salary] <= 1903.98
33       result[:irrf_tax] = 'Isento'
34       result[:irrf_value] = 0.0
35     elsif result[:real_salary] >= 1903.99 || result[:real_salary] <= 2826.65
36       result[:irrf_tax] = '7.5%'
37       result[:irrf_value] = (result[:real_salary] * 0.075) - 142.8
38     elsif result[:real_salary] >= 2826.66 || result[:real_salary] <= 3751.05
39       result[:irrf_tax] = '15%'
40       result[:irrf_value] = (result[:real_salary] * 0.15) - 354.8
41     elsif result[:real_salary] >= 3751.06 || result[:real_salary] <= 4664.68
42       result[:irrf_tax] = '22.5%'
43       result[:irrf_value] = (result[:real_salary] * 0.225) - 636.13
44     elsif result[:real_salary] > 4664.68
45       result[:irrf_tax] = '27.5%'
46       result[:irrf_value] = (result[:real_salary] * 0.275) - 869.36
47     end
48
49     result[:real_salary] -= result[:irrf_value]
50   end
51 end

```

Figura 3.16: Reescrevendo o código da classe *SalaryCalculator*
 Autoria Própria

Para validar o final do ciclo do TDD é preciso executar novamente os testes unitários da Figura 3.12 e verificar que todos ainda passam, ou seja, a nova implementação além de melhor continua garantindo o funcionamento do código, conforme visto na Figura 3.17.


```

→ app-monografia git:(master) ✕ bundle exec rspec --tag focus
Run options: include {:focus=>true}

SalaryCalculator
  #calculate
    when salary is 1.2000
      should eq "Isento"
      should eq 0.0
      should eq "8%"
      should eq 96.0
      should eq 1104.0

Finished in 0.00201 seconds (files took 0.08654 seconds to load)
5 examples, 0 failures

```

Figura 3.17: Executando novamente os testes unitários, código continua funcionando após a refatoração

Autoria Própria

Com esses passos se encerra o ciclo do TDD e para finalizar o ciclo do BDD é preciso executar novamente a especificação escrita na Figura 3.3. Na Figura 3.18 a execução da especificação mostra que o ciclo do BDD foi completado pois a especificação que antes era falha agora está executando corretamente, garantindo a funcionalidade da aplicação.

```

→ app-monografia git:(master) ✕ bundle exec cucumber
# language: pt
Funcionalidade: Calcular salário líquido
  Como um usuário
  Quero poder calcular meu salário líquido
  Para saber qual será meu ganho real

  Cenário: Salário R$ 1.200,00 reais é isento de IRRF e 8% de INSS.
    Dado que informo meu salário bruto de '1200.00'
    E cliço para calcular o meu salário líquido
    Então vejo a mensagem IRRF - Aliquota (%): 'Isento'
    Então vejo a mensagem IRRF - Valor: R$ '0.00'
    Então vejo a mensagem INSS - Aliquota (%): '8%'
    Então vejo a mensagem INSS - Valor: R$ '96.00'
    Então vejo a mensagem Salário Líquido: R$ '1104.00'

1 cenário (1 passed)
7 steps (7 passed)
0m0.019s

```

Figura 3.18: Executando a especificação, código funcionando corretamente

Autoria Própria

A execução de todos os testes e principalmente a da especificação sem erros, mostra que o ciclo do BDD foi realizado com sucesso.

Capítulo 4

Resultados e Discussões

Capítulo 5

Considerações Finais

Referências Bibliográficas

BARAÚNA, H. *Cucumber e Rspec - Construindo aplicações Ruby e Rails com testes e especificações*. [S.l.]: Casa do Código, 2013.

BECK, K. *Manifesto para Desenvolvimento Ágil de Software*. 2001. Disponível em: <<http://www.manifestoagil.com.br/>>. Acesso em: 17 jun. 2015.

CAMPOS, F. M. *Qualidade de Software e Garantia de Qualidade de Software são as mesmas coisas ?* 2012. Disponível em: <<http://www.linhadecodigo.com.br/artigo/1712/qualidade-qualidade-de-software-e-garantia-da-qualidade-de-software-sao-as-mesmas-coisas.aspx>>. Acesso em: 19 jun. 2015.

CORBUCCHI, H.; ANICHE, M. *Test Driven Development: Teste e design no mundo real com Ruby*. [S.l.]: Casa do código, 2013.

COSTA, P. *10 falhas de software que marcaram a história*. 2012. Disponível em: <<http://crowdtest.me/10-falhas-software-marcaram-historia/>>. Acesso em: 29 jun. 2015.

FOWLER, M. *The new Methodology*. 2003. Disponível em: <<http://martinfowler.com/articles/newMethodology.html>>. Acesso em: 17 jun. 2015.

JAMIL, V. *O que é a metodologia Scrum em projetos?* 2013. Disponível em: <<http://www.scriptcase.com.br/blog/metodologia-scrum-projetos/>>. Acesso em: 18 jun. 2015.

KUHN, G. R. *Apresentando XP: Encante seus clientes com Extreme Programming*. 2008. Disponível em: <<http://javafree.uol.com.br/artigo/871447/Apresentando-XP-Encante-seus-clientes-com-Extreme-Programming.html>>. Acesso em: 25 jun. 2015.

NORTH, D. *Introducing BDD*. 2006. Disponível em: <<http://dannorth.net/introducing-bdd/>>. Acesso em: 29 jun. 2015.

PASSUELO, L. *A Nova Metodologia*. 2005. Disponível em: <<http://www.p3software.com.br/home/artigos/6-a-nova-metodologia>>. Acesso em: 17 jun. 2015.

PRESSMAN, R. S. *Engenharia de Software*. 7. ed. [S.l.]: MCGRAW HILL - ARTMED, 2011.

ROCHA, F. G. *TDD: Fundamentos do Desenvolvimento de Software orientado a testes*. 2013. Disponível em: <<http://www.devmedia.com.br/tdd-fundamentos-do-desenvolvimento-orientado-a-testes/28151>>. Acesso em: 28 jun. 2015.

RUBY, S.; THOMAS, D.; HANSSON, D. H. *Agile Web Development with Rails 4*. [S.l.]: The Pragmatic Bookshelf, 2013.

SABBAGH, R. *Scrum - Gestão Ágil para projetos de sucesso*. [S.l.]: Casa do Código, 2013.

SANCHEZ, I. *Apresentando Behavior Driven Development*. 2006. Disponível em: <<https://dojofloripa.wordpress.com/2006/10/28/apresentando-behaviour-driven-development-bdd/>>. Acesso em: 29 jun. 2015.

SCHWABER, J. S. K. *Guia do Scrum*. [S.l.]: Scrum.org, 2013.

SOARES, I. *Desenvolvimento Orientado a Comportamento*. 2011. Disponível em: <<http://www.devmedia.com.br/desenvolvimento-orientado-por-comportamento-bdd-artigo-java-magazine-91/21127>>. Acesso em: 29 jun. 2015.

SOFTWARE, R. *Test your code and reap app performance*. 2013. Disponível em: <<http://www.ruby-software.com/test-your-code-and-reap-app-performance/>>. Acesso em: 01 jul. 2015.

TI, P. *10 falhas de software que marcaram a história*. 2012. Disponível em: <<http://www.profissionaisti.com.br/2012/01/alguns-dos-mais-famosos-erros-de-sofware-da-historia/>>. Acesso em: 29 jun. 2015.

ÁGIL, D. *O que é Extreme Programming ?* 2013. Disponível em: <<http://www.desenvolvimentoagil.com.br/xp/>>. Acesso em: 25 jun. 2015.

ÁGIL, D. *O que é Scrum ?* 2013. Disponível em:
<<http://www.desenvolvimentoagil.com.br/scrum/>>. Acesso em: 15 jun. 2015.

ÁGIL, M. *Manifesto Ágil*. 2001. Disponível em:
<<http://www.agilemanifesto.org/iso/ptbr/>>. Acesso em: 15 jun. 2015.