Luiz Cezer Marrone Filho

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

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
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 Bacharaledo em Sistemas de Informação. Curso de Pós-Graduação em Engenharia de Software.

Sumário

Lista de Figuras

Resumo

Abstract

Capítulo 1

Introdução

??

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 auxília no desenvolvimento de *software*;

1.3 Justificativa do 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 desenvolvedore

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 metologias 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 levantado 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 reune 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:** Da 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 metologia 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 tera 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 qualque 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 propoem a crição 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 comple- xos. *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 metologias, 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:

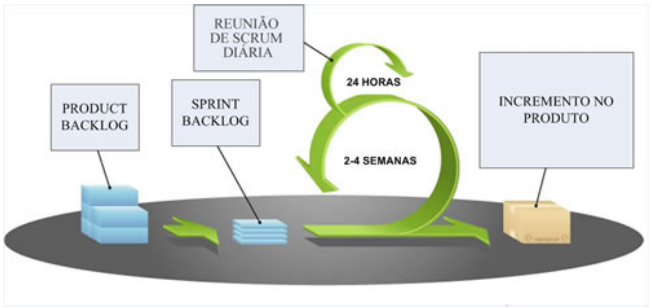

Figura 2.1: Visão geral de uma *Sprint* dentro do *Scrum*
(JAMIL, 2013)

2.4.1 O time do Scrum

Dentro do *Scrum* existem três papéis definidos: o Time de desenvolvimento, *Scrum Master* e o *Product Owner*. 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.

2.4.1.1 Time de desenvolvimento

São os responsáveis pela entrega do produto, geralmente um time é composto de três a dez pessoas. Formam um time auto-organizado, multidisciplinar, motivado e com foco em qualidade. Dentro do time não há divisões entre programadores, *designers*, engenheiros ou analistas. Todos trabalham juntos de forma igualmente responsável, com objetivo de entregar o que foi prometido para a *Sprint*. (SABBAGH, 2013) (ÁGIL, 2013b)

O time possui várias atividades e tarefas dentro da *Sprint*:

- Planejar seu trabalho junto ao *Product Owner* e posteriomente dividir as tarefas entre a equipe;
- Realizar as tarefas para que o objetivo da *Sprint* seja atingido;

- Interagir com *Product Owner* sempre que for necessário esclarecimento de alguma regra de negócio;
- Identificar e informar ao *Scrum Master* todo e qualquer impedimento que possa atrapalhar o trabalho da equipe;
- Entregar valor para o cliente de forma frequente, organizada e com qualidade.

2.4.1.2 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.

Também colabora com o time de desenvolvimento a longo da *Sprint* no esclarecimento de dúvidas sobre as regras de negócio e fica responsável por aceitar ou não o trabalho entregue pela equipe no decorrer da *Sprint*, validando se o que foi entregue é realmente o que foi combinado e irá satisfazer a necessidade do cliente. (SABBAGH, 2013) (ÁGIL, 2013b)

2.4.1.3 Scrum Master

O *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. (SABBAGH, 2013) (ÁGIL, 2013b)

2.4.2 Artefatos do Scrum

Os artefatos do *Scrum* servem como ferramentas de opoio 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.

2.4.2.1 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*. Necessariamente não contêm somente novas funcionalidades, podem também conter melhorias, correções de falhas, questões técnicas, ou seja, tudo que for necessário e possa vir a ser desenvolvido para entregar o produto.

Incialmente não precisa conter todas as tarefas, pode conter somente aquelas necessárias para iniciar o desenvolvimento do produto e a medida que o time e o cliente aprendem mais sobre o produto, novas tarefas serem adicionadas. (SABBAGH, 2013) (ÁGIL, 2013b)

2.4.2.2 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 liberadade de escolher os itens que são inclusos na *Sprint* de acordo com a sua percepção do tempo que cada item levará para ficar pronto.

O desempenho do time pode ser medido, baseando-se na quantidade de itens que foram entregues entre um dia outro da *Sprint*. (SABBAGH, 2013) (ÁGIL, 2013b)

2.4.2.3 Quadro de Tarefas

Para tornar mais fácil e visual o acompanhamento das tarefas dentro da *Sprint*, é um comum equipes usarem um quadro de tarefas, que geralmente é quebrado de acordo com a situação de cada tarefa dentro da *Sprint*. (ÁGIL, 2013b)

Geralmente esse quadro possui as seguintes situações: Pendente, Andamento, Finalizado. Um exemplo desse quadro pode ser visto na Figura 2.2 abaixo:

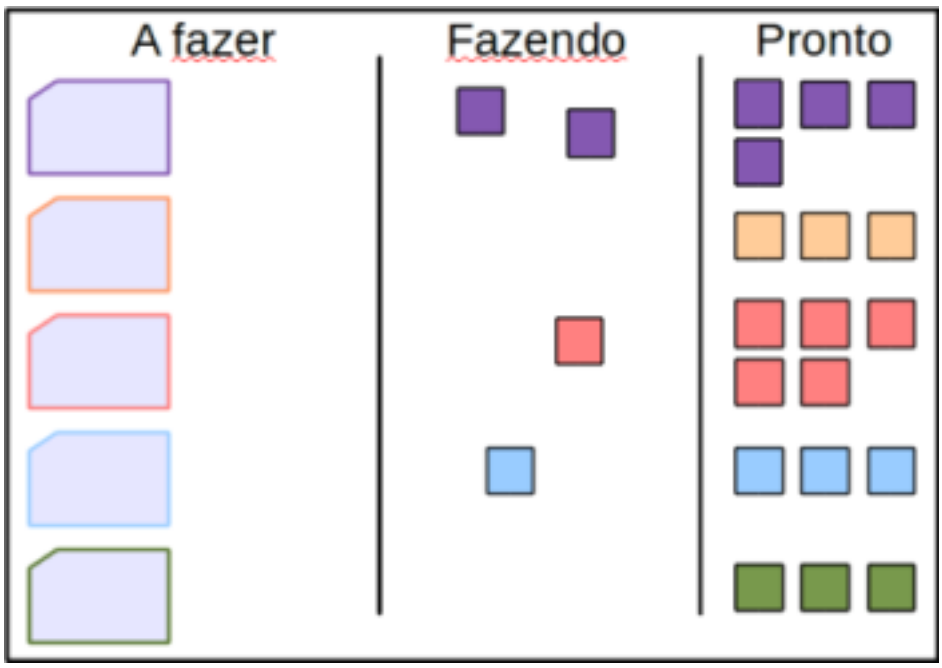

Figura 2.2: Quadro de tarefas de uma equipe ágil
(FERNANDES, 2011)

2.4.2.4 Gráfico de Burn Down

Aliado ao quadro de tarefas, é comum equipes ágeis utilizarem gráficos para melhorar o acompanhamento de tudo aquilo que foi feito e o que ainda está pendente dentro da *Sprint*. Com esse gráfico é possível medir o ritmo da equipe e detectar possíveis impedimentos. (ÁGIL, 2013b)

Um exemplo de gráfico, pode ser visto na Figura 2.3 abaixo:

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*.

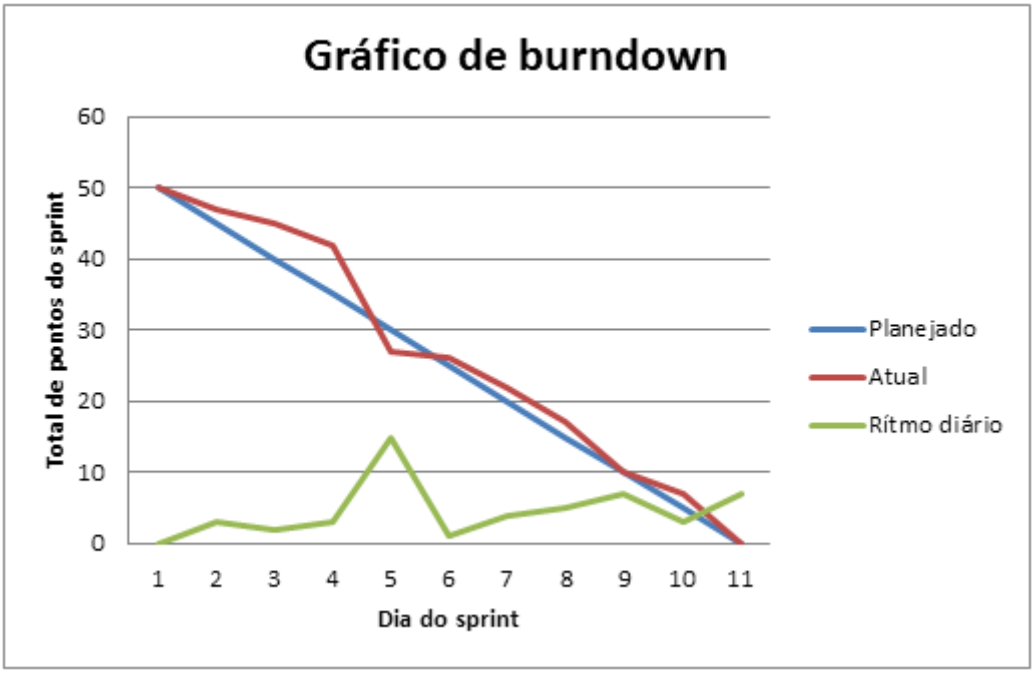

Figura 2.3: Gráfico de *Burn Down* do *Scrum*
(CUNHA, 2012)

2.4.3.1 Sprint

A *Sprint* é um mini-projeto, que tem duração entre uma e quatro 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.

As *Sprints* sempre se iniciam com uma *Sprint Planning* que é uma reunião, com participação de todos os envolvidos no projeto e nessa reunião são definidas quais tarefas serão desenvolvidas pelo time de desenvolvimento para adicionar um novo incremento ao produto. Ocorrem em sequência, ou seja, não há intervalos entre as *Sprints*, cada vez que uma termina, uma nova *Sprint* se inicia. (SCHWABER, 2013) (SABBAGH, 2013)

Existem algumas regras que devem ser seguidas dentro de uma *Sprint*:

- Não são feitas mudanças que podem por em risco o objetivo da *Sprint*;
- As metas de qualidade nunca diminuem;
- O escopo pode ser negociado com o *Product Owner*;
- Somente o *Product Owner* tem autoridade para cancelar uma *Sprint*.

2.4.3.2 Sprint Planning

É a primeira cerimônia de uma nova *Sprint*. É uma reunião com participação de todos os envolvidos onde o *Product Owner* mostra quais as atividides de maior prioridade naquele momento e a equipe irá selecionar quais dessas atividades entram na *Sprint* para gerar um novo incremento no *software*.

Nessa reunião o time de desenvolvimento tem a liberdade de questionar o *Product Owner* sobre as atividades e também obter esclarecimento sobre dúvidas. Essas tarefas selecionadas darão origem ao *Sprint Backlog*. (SCHWABER, 2013) (ÁGIL, 2013b)

2.4.3.3 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 quinze minutos. Essa reunião visa melhorar a organização da equipe e tem como foco a transparência para que todos saibam o que todos estão fazendo e se alguém está passando por alguma dificuldade.

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 ?

Se algum impedimento é encontrado, o *Scrum Master* irá agir com a finalidade de remover esse bloqueio para que o fluxo de trabalho da equipe continue. (ÁGIL, 2013b)

2.4.3.4 Sprint Review

A *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*. Também serve para detectar se houveram problemas críticos que

pudessem atrapalhar o andamento do trabalho e como eles foram solucionados. (ÁGIL, 2013b)

2.4.3.5 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. (ÁGIL, 2013b)

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* satifaz 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 2.4.

Figura 2.4: A Pirâmide de Qualidade de Software
(CAMPOS, 2012)

2.6 Testes de Software

A evolução tecnologica 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 utlizavam um aparelho chamado *Therac-25* para o tratamento com radicação 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 uma 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 previnilás 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) de 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 garatem que cada pequeno componente consiga funcionar de forma isoloada;
- **Teste de Integração:** Visa testar o fluxo do sistema como um tudo 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 2.5, partindo da menor unidade do *software* até sua integração total e seu uso.

Figura 2.5: 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 garatir 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. 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 2.6 e pode ser definido da seguinte forma:

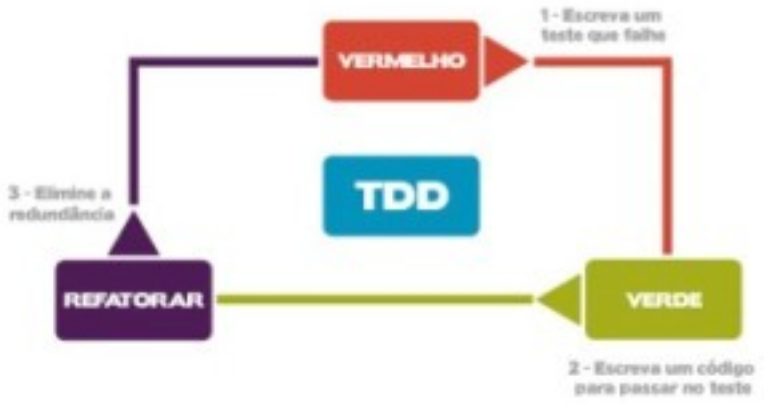

Figura 2.6: O ciclo do TDD
(ROCHA, 2013)

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 anteriomente.

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 previnir e arrumar falhas de codificação de forma imediata diminuindo os custos e previnindo futuras manutenções no código.

A Figura 2.7 mostra como é a forma de recebimento do *feedback* do código pelo desenvolvedor que prática o TDD e um desenvolvedor que não prática TDD.

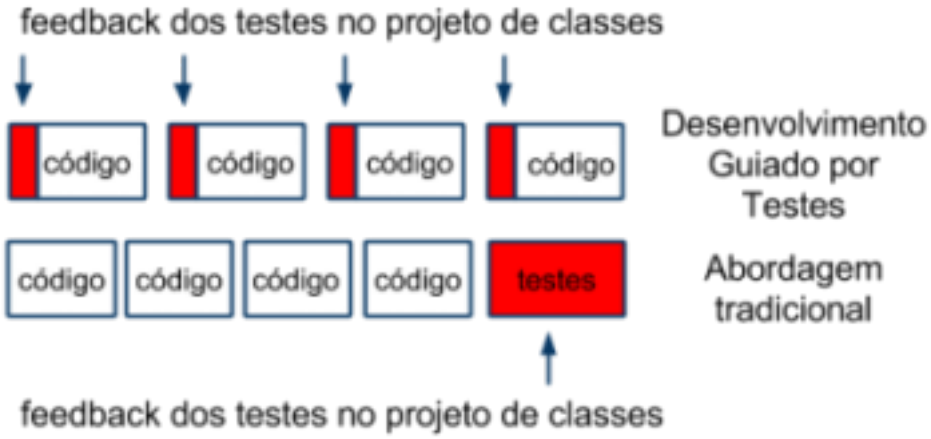

Figura 2.7: *Feedback* do código utilizando TDD
(CORBUCCI; ANICHE, 2013)

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. Nessa prática as equipes irão se focar em descrever um comportamento esperado pelo *software* para após isso desenvolvedor 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 acertivo. (BARAúNA, 2013)

O BDD pode ser dito como um evolução sobre o TDD, visto que o TDD foca na construção do código, BDD se preocupa em saber se o código criado irá se comporta de forma esperada e 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, visto 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 a seus alunos TDD, por conta de sua nomeclatura 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 tentantiva 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.

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 2.8 visa ilustrar a forma de desenvolvimento proposta pelo BDD:

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 ba-

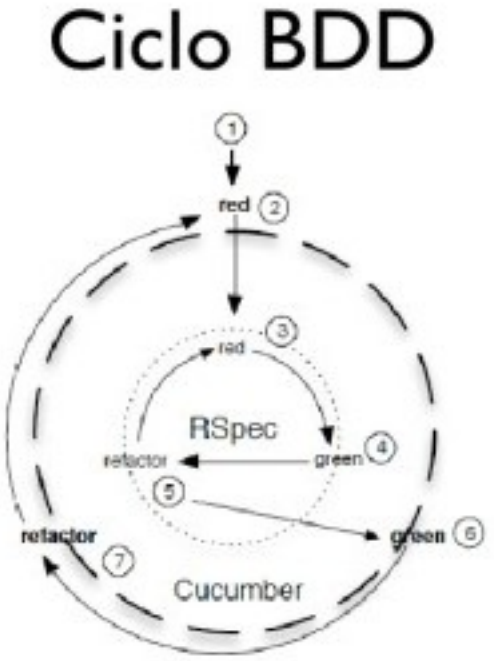

Figura 2.8: O ciclo do BDD
(CORBUCCI; ANICHE, 2013)

seada no que havia de melhor me outras linguagens como: *Smalltalk*, *Lisp* e *Ada*. Tudo em Ruby é um 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 melhoria constante na correção de erros e facilitando a melhoria e inclusão de novas funcionalidades a linguagem. (RUBY; THOMAS; HANSSON, 2013)

Na Figura 2.9 é possível ver um exemplo de código Ruby:

Na primeira linha da Figura 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, a interpretador irá entender que a variável number será do tipo inteiro.

Como em Ruby tudo é um objeto é possível fazer a checagem de qual tipo de objeto a variável *number* é, nesse caso executando o comando *number.class* e obtendo o resultado *Fixnum*, indicando que a variável é um objeto do tipo inteiro.

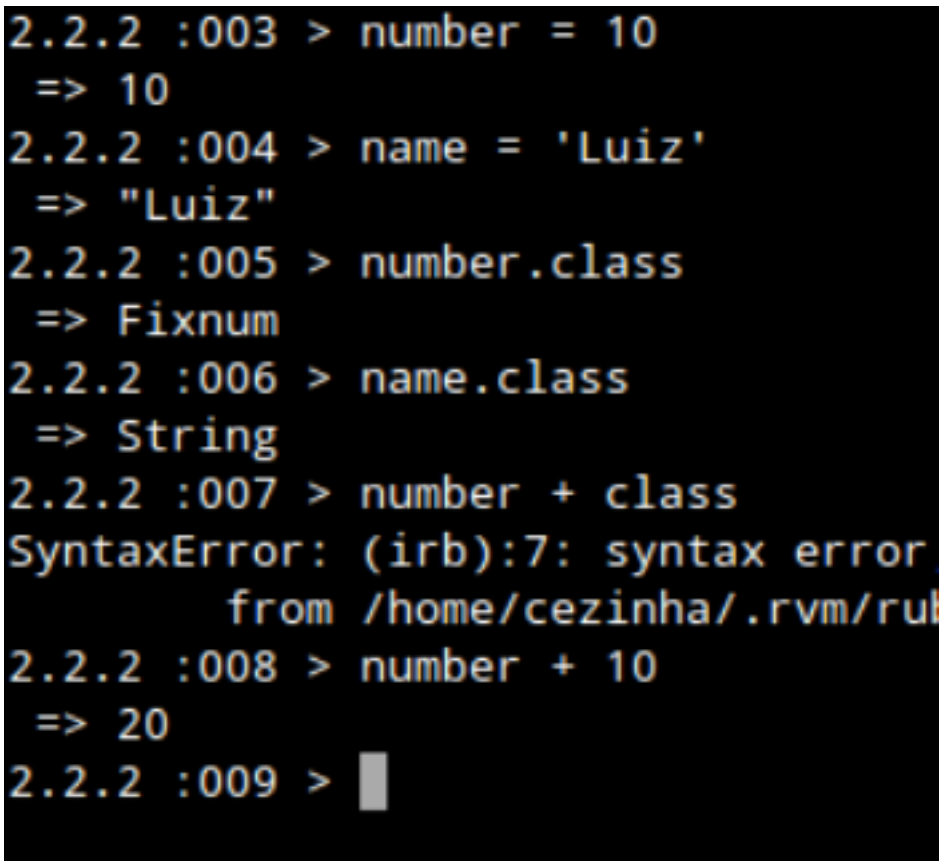

Figura 2.9: Exemplo de código Ruby
Autoria Própria

Referências Bibliográficas

BARAúNA, H. *Cucumber e Rspec - Contrua 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.

CORBUCCI, 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.

CUNHA, D. *Metodologia de Gerenciamento baseado em Scrum*. 2012. Disponível em: <http://www.olharcritico.com/engenhariadesoftware/2012/01/metodologia-de-gerenciamento-baseada-em-scrumparte-8/>. Acesso em: 18 jun. 2015.

FERNANDES, C. *Pensando em métricas para times ágeis*. 2011. Disponível em: <http://blog.caelum.com.br/pensando-em-metricas-para-times-ageis/>. Acesso em: 18 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 Agil 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.

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-softwares-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.