UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
DIRETORIA DE PESQUISA E PÓS-GRADUAÇÃO
ESPECIALIZAÇÃO EM ENGENHARIA DE *SOFTWARE*

MARCIO ANGELO MATTÉ

# TESTES DE *SOFTWARE*: UMA ABORDAGEM DA ATIVIDADE DE TESTE *SOFTWARE* EM METODOLOGIAS ÁGEIS APLICANDO A TÉCNICA *BEHAVIOR DRIVEN DEVELOPMENT* EM UM ESTUDO EXPERIMENTAL

MONOGRAFIA DE ESPECIALIZAÇÃO

MEDIANEIRA
2011

MARCIO ANGELO MATTÉ

# TESTES DE *SOFTWARE*: UMA ABORDAGEM DA ATIVIDADE DE TESTE *SOFTWARE* EM METODOLOGIAS ÁGEIS APLICANDO A TÉCNICA *BEHAVIOR DRIVEN DEVELOPMENT* EM UM ESTUDO EXPERIMENTAL

Monografia apresentada como requisito parcial à obtenção do título de Especialista na Pós Graduação em Engenharia de *Software*, da Universidade Tecnológica Federal do Paraná – UTFPR – Campus Medianeira.

Orientador: Prof MSc. Alan Gavioli.

MEDIANEIRA

2011

Ministério da Educação
**Universidade Tecnológica Federal do Paraná**
Diretoria de Graduação e Educação Profissional
Especialização em Engenharia de *Software*

## TERMO DE APROVAÇÃO

**Testes de *Software*: Uma abordagem da atividade de teste de *software* em metodologias ágeis aplicando a técnica *Behavior Driven Development* em um estudo experimental.**

Por

**Marcio Angelo Matté**

Esta monografia foi apresentada às 11h00min do dia 15 de dezembro de 2011 como requisito parcial para a obtenção do título de Especialista no curso de Especialização em Engenharia de *Software*, da Universidade Tecnológica Federal do Paraná, *Campus* Medianeira. Os acadêmicos foram arguidos pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. M.Sc Alan Gavioli
UTFPR – Campus Medianeira
(orientador)

_____
Prof M.Sc. Everton Coimbra de Araújo
UTFPR – Campus Medianeira

_____
Prof M.Sc. Fernando Schutz
UTFPR – Campus Medianeira

## AGRADECIMENTOS

À Deus acima de tudo, pela saúde que me deu e a alegria de estar vivo ao redor de uma família maravilhosa.

Aos meus pais, Adilor e Domingas, que sempre deram o suporte necessário desde os primeiros dias da minha vida e que dão até hoje, muito obrigado.

Aos meus irmãos, Angela, Junior, Marcelo e Mônica que por mais que pareça um clichê, são os melhores irmãos do mundo de verdade.

Aos meus “cunhados”, André, Franciele, Ana e Juliano que fazem parte da família e são pessoas que tenho apreço e admiração. Muito obrigado.

Ao pequeno Martin, o “Tinzinho”, que chegou em nossas vidas há quase um ano e desde então a luz emanada por ele faz nossos olhos brilharem todos os dias.

À minha namorada, noiva e futura esposa Tayrine, que desde que nos conhecemos só trouxe alegria a minha vida. O companheirismo dela simplesmente é combustível em minha vida e só tenho a agradecer todos esses dias vividos ao seu lado e que Deus me permita viver muitos mais daqui por diante.

Ao meu orientador Alan Gavioli que pela confiança e ter estado presente sempre que necessitei de seu apoio, nas orientações, nas conversas informais e pela amizade, obrigado.

Aos meus colegas docentes na UTFPR, com quem tenho aprendido muito nessa carreira de docência que pretendo levar como profissão até que a saúde me permitir.

Finalmente a todos aqueles que participam de minha vida de algum modo, amigos, colegas, alunos e companheiros. A todos muito obrigado!

“Sempre que eu quero, revejo meus dias
e as coisas que eu posso, eu mudo ou arrumo!
Mas deixo bem quietas as boas lembranças,
vidinha que é minha, só pra o meu consumo”.

(Luiz Marenco)

## RESUMO

MATTÉ, Marcio Angelo. Teste de *Software*: Uma abordagem do teste de *software* em metodologias ágeis aplicando a técnica *Behavior Driven Develpment* em um estudo experimental. 2011. 53f. Monografia (Especialização em Engenharia de *Software*). Universidade Tecnológica Federal do Paraná, Medianeira, ano.

Este trabalho apresenta conceitos sobre o teste de *software* em metodologias ágeis através da abordagem da técnica de BDD (*Behavior Driven Develpment*) Desenvolvimento Dirigido pelo Comportamento. Também é parte deste documento a contemplação de assuntos que permeiam o tema qualidade de *software*.



**Palavras-chave:** Desenvolvimento de *software*, processo de desenvolvimento e métodos de desenvolvimento.

## ABSTRACT

MATTÉ, Marcio Angelo. *Software* Test: An approach to *software* testing in agile methodologies applying the technique Behavior Driven Development in an experimental study. 2011. 53f. Monografia (Especialização em Engenharia de *Software*). Universidade Tecnológica Federal do Paraná, Medianeira, 2011.

This paper aims bring together the concepts of *software* testing in agile methodologies approach through the technique of BDD – Behavior Driven Development. Also part of this document is the contemplation of issues that permeate the theme of software quality.



**Keywords:** Software development, process of development and development methods.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE SIGLAS

| | |
|---|---|
| BDD | *Behavior Driven Development* |
| CMMI | *Capability Maturity Model Integration* |
| MPS.br | Melhoria de Processo de Software Brasileiro |
| TDD | *Teste-Driven Development* |
| SQA | *Software Quality Assurance* |
| XP | *Extreme Programming* |

## LISTA DE QUADROS

**SUMÁRIO**

# 1 INTRODUÇÃO

O conceito de qualidade é, para muitos, algo intangível ou que só é atingido se o resultado de um processo, método, atividade ou, qualquer outra coisa que culmine com um “produto” final, esteja sem defeitos ou em pleno funcionamento.

Segundo Koscianski e Soares (2007) “a ideia de qualidade é aparentemente intuitiva; contudo, quando examinando mais longamente, o conceito se revela complexo”.

Com o *software,* a qualidade se define da mesma maneira, ou seja, de forma que possa ser utilizado um processo de validação e verificação.

Antigamente, talvez não mais do que vinte cinco anos atrás, construir *software* era um processo quase que artesanal, exceto para grandes corporações precursoras no desenvolvimento de *software*. Não havia processos tão bem definidos ou tão bem testados, ou tecnologia abundante a ponto de garantir que o *software* produzido fosse executado com a certeza de contemplar todos os seus requisitos na completude e/ou sem falhas.

Com o passar dos anos, o aumento significativo da tecnologia como um todo e do oferecimento de subsídios técnicos, de conhecimento ou de produtos resultantes da tecnologia (computadores, dispositivos eletrônicos, Internet, etc...), fizeram com que as pessoas começassem a se preocupar mais com a qualidade em toda sua forma, adaptado de (KOCIANSKI e SOARES, 2007).

Visando a qualidade, testar o produto é atividade inerente ao processo de desenvolvimento e se faz altamente necessário para garantir que não seja introduzida uma falha ou defeitos durante o processo.

Nas metodologias ágeis o teste é atividade que acontece junto com o desenvolvimento, ou seja, ao contrário das metodologias clássicas, que realizam os testes com o produto já construído, permitindo assim, que no caso de falha ou defeito o produto, volte ao seu ambiente de desenvolvimento para correções até ser entregue ao cliente final.

## 1.1 OBJETIVO GERAL

Aplicar em um estudo experimental a técnica de teste de *software* em processos de desenvolvimento ágil guiado pelo comportamento – BDD.

## 1.2 OBJETIVOS ESPECÍFICOS

- Realizar estudo sobre teste de *software* sob o ponto de vista do desenvolvimento ágil em relação ao desenvolvimento tradicional;
- Aplicar a técnica de BDD em um estudo experimental;
- Exemplificar por meio de modelos a aplicação da técnica de BDD com a ferramenta *JBehave*.

## 1.3 JUSTIFICATIVA

Este tema foi escolhido devido à popularização e aplicação das metodologias ágeis no desenvolvimento de *software* frente ao desenvolvimento tradicional.

O uso de metodologias ágeis não substitui de forma alguma todo o conhecimento em métodos tradicionais de desenvolvimento, pelo contrário, sua aplicação tem como objetivo ser uma alternativa aos processos mais burocráticos, que, às vezes, torna o gerenciamento de projetos de pequeno e médio porte, uma tarefa quase que impossível de ser administrada.

Segundo Koscianski e Soares (2007, p.190) apud Sommerville (2003), uma metodologia de desenvolvimento é constituída de várias atividades que ajudam na construção do *software*, onde, o resultado destas atividades é o produto de *software*. Todavia, existem várias metodologias aplicadas ao desenvolvimento, mas que quase sempre são constituídas das mesmas atividades.

Nas metodologias ágeis as atividades também são executadas com o propósito de construção de um produto de *software*, porém estas nem sempre são tão burocráticas e se baseiam nas competências dos indivíduos formando um elo entre todos os participantes da equipe de desenvolvimento.

Pressman (2011, p.82) referencia uma discussão proposta por Ivar Jacobson, onde é mencionado que a agilidade é palavra da moda e, todos querem ou desejam ser ágeis, onde equipes ágeis são aquelas que dão respostas adequadas às mudanças durante o ciclo ou processo de desenvolvimento, e que encarem a mudança de forma natural como é proposto no manifesto ágil.

Deste modo, tanto nas metodologias tradicionais quanto nas metodologias ágeis, as atividades compõem o processo de desenvolvimento visando à entrega de

produto com valor e qualidade, e a qualidade só é atingida se o produto for construído de acordo com um processo bem definido.

## 1.4 ESTRUTURA DO TRABALHO

Este trabalho está divido em cinco capítulos. No capítulo um é feita uma introdução sobre o panorama do tema a ser estudado, bem como, os objetivos específicos, objetivo geral e justificativa.

No capítulo dois é feito o embasamento teórico sobre os assuntos que circundam o tema principal. Neste capítulo o foco é mais geral e não tem como objetivo aprofundar em cada item nele contido.

O capítulo três contém o estudo aplicado sobre o tema central do trabalho e apresenta um estudo de caso simples para ilustrar o desenvolvimento deste tema.

Já os capítulos quatro e cinco apresentam os resultados e discussões e ainda o desfecho com a conclusão do trabalho, respectivamente.

## 1.5 DELIMITAÇÃO DO TRABALHO

Este trabalho em nenhum momento propõe qualquer tipo de comparação entre processos de desenvolvimento ou metodologias aplicadas aos processos. Restringe-se especificamente a aplicação de uma técnica de teste a um estudo experimental.

# 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta o embasamento do tema bem como a fundamentação com base na engenharia de *software*.

## 2.1 ENGENHARIA DE *SOFTWARE*

Para Hélio Engholm Júnior (2010) a engenharia de *software* possui concentração nos aspectos práticos das atividades de elaboração, construção e manutenção do produto de *software* em todo seu ciclo de desenvolvimento.

Surgiu com o objetivo de mitigar problemas na concepção do produto de *software*, oriundas da chamada “crise do *software*”, nos anos setenta, onde o aumento na demanda por sistemas e a complexidade dos mesmos culminavam em produtos de baixa qualidade, prazos e custos muito maiores do que os inicialmente previstos e produtos que não atendiam as reais necessidades de seus *sponsors*[1], além dos custos elevados para correção ou evolução do produto.

Segundo Sommerville (2011) a engenharia de *software* tem por objetivo apoiar o desenvolvimento profissional de *software* mais do que as atividades de programação individual. Trata-se de uma disciplina de engenharia onde o foco principal está em todos os aspectos da produção de *software* até a conclusão do projeto, mantendo o gerenciamento na evolução do sistema, mesmo depois de entrar em operação por parte do usuário final.

Sommerville, através desta definição, aponta duas expressões importantes: Disciplina de engenharia e todos os aspectos da produção de *software*.

1. **Disciplina de engenharia**: é a atividade desenvolvida por engenheiros, onde o objetivo é ter o produto funcionando. O uso é seletivo e, portanto, problemas ocorridos durante o desenvolvimento devem ser resolvidos com base no conhecimento do engenheiro, através das metodologias, processos, experiências entre outras. Também faz parte da disciplina desenvolvida pelo engenheiro de *software* trabalhar com as restrições diversas (financeira, temporal, humana, etc) e assim organizar e conduzir

[1] **Tradução livre**: *sponsor* pode ser considerado, no âmbito da Engenharia de *Software*, um dos interessados, mais precisamente o cliente patrocinador do projeto/produto de *software*.

para que o resultado final seja satisfatório, ou dentro do critério de aceitação do cliente;

2. **Todos os aspectos da produção de *software***: os processos técnicos não são as únicas preocupações da engenharia de *software*, mas também as atividades de gerenciamento de projeto, desenvolvimento de ferramentas, métodos e teorias para apoiar a produção de *software* (SOMMERVILLE, 2011, p. 5).

Deste modo a engenharia de *software* se torna importante por dois motivos: cada vez mais as pessoas, empresas e usuários em geral, precisam de sistemas mais avançados que custem o justo e sejam entregues em tempo razoável dentro do previsto na fase de projeto (SOMMERVILLE, 2011).

## 2.2 PROCESSO DE *SOFTWARE*

O produto gerado através da engenharia de *software* é a resultante de diversas atividades executadas umas após as outras, com o objetivo de "materializar" este produto. Estas atividades executadas de forma coordenada sob uma metodologia é conhecida como **processo de *software***.

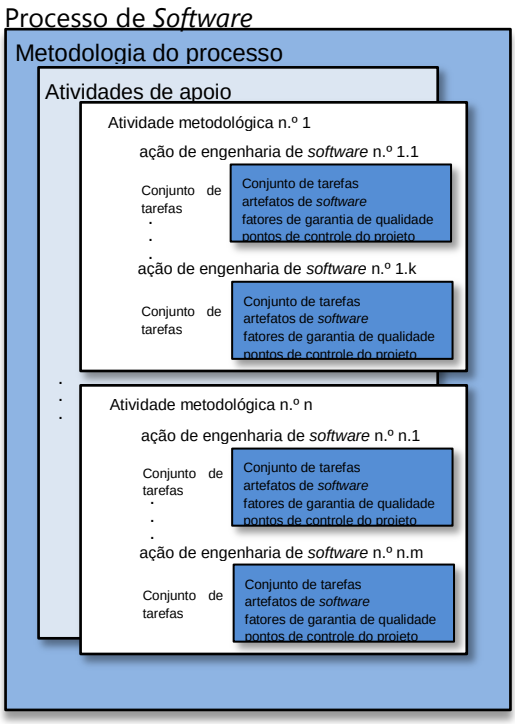


**FIGURA 1 - Representação esquemática do processo de *software*.**

**Fonte: Pressman (2011, p.53).**

Segundo Pressman (2011, p.53) o processo de *software* é composto por várias atividades metodológicas, e, em cada atividade metodológica, existem ações específicas compostas por um conjunto de tarefas, artefatos, fatores de garantia de qualidade e pontos de controle do projeto, mostradas na FIGURA 1. Estas atividades metodológicas compõem as tarefas de apoio à metodologia do processo de *software*, e, desta forma, iniciam um ciclo que se repete até a conclusão do projeto e obtenção do produto de *software*.

Existem várias metodologias de processo ou desenvolvimento de *software* atualmente, algumas mais burocráticas ou pesadas, outras ditas leves ou flexíveis. Independentemente de qual seja a metodologia ou processo adotado, é possível realizar as atividades de desenvolvimento através de uma metodologia genérica, proposta por Pressman (2011, p. 53), onde ele estabelece cinco atividades metodológicas: **comunicação**, **planejamento**, **modelagem**, **construção** e **entrega**.

Ainda de acordo com Pressman (2011), estas atividades metodológicas genéricas servem como orientação para que o processo de *software* atinja seus objetivos de forma satisfatória. As cinco atividades sempre serão executadas, independente do processo, como já dito anteriormente, uma após a outra até que seja completado o fluxo do processo. Existem quatro tipos de fluxo de atividades, segundo Pressman (2011, p.54): **fluxo linear**, **fluxo de processo iterativo**, **fluxo de processo evolucionário** e **fluxo de processo paralelo**, como ilustra a FIGURA 2.

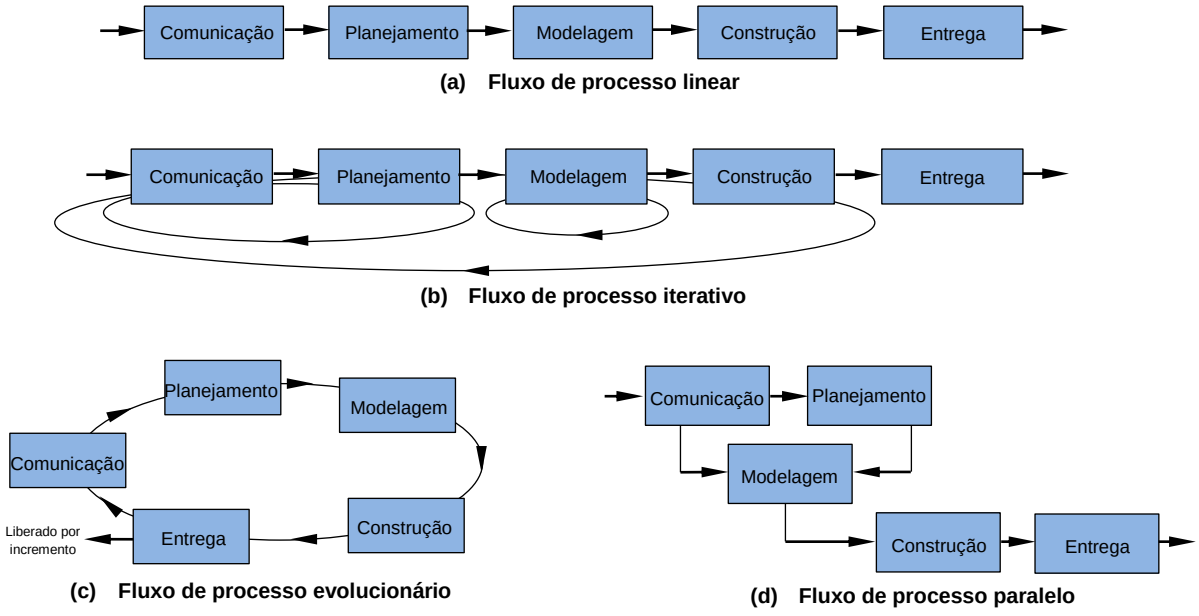


**FIGURA 2 - Fluxo de processo**
**Fonte:Pressman (2011, p.54)**

Cada fluxo visto na FIGURA 2 determina como o processo é executado, ou seja, entre o início do projeto e o seu fim. Em cada etapa, são realizadas as tarefas referentes àquela etapa. Na comunicação são executadas todas as tarefas que antecedem o planejamento, no planejamento as tarefas que antecedem a modelagem e assim sucessivamente até completar o fluxo.

No **(a) fluxo de processo linear**, as atividades acontecem de maneira que ao atingir a última atividade o produto é entregue. Já o **(b) fluxo de processo iterativo** as entregas são feitas em etapas até ser conseguir um produto, o que acontece em ciclos (iterações) como se fossem várias atividades lineares, porém, neste caso as atividades nem sempre são todas executadas, podendo haver, entre uma entrega e outra, a execução ou não de uma determinada atividade. No **(c) fluxo de processo evolucionário** as atividades são executadas de forma circular, onde a cada volta completa neste círculo incrementa-se uma nova versão do produto, contendo novas funcionalidades incorporadas ao produto entregue na volta anterior. Por fim, o **(d) fluxo de processo paralelo** pode ter atividades acontecendo em paralelo, ou seja, a modelagem pode estar acontecendo ao passo que o planejamento é executado (PRESSMAN, 2011).

## 2.3 DESENVOLVIMENTO ÁGIL

O termo “Desenvolvimento Ágil” traz a ideia de agilidade como algo que todos os profissionais de TI (Tecnologia da Informação), mais precisamente os desenvolvedores de sistemas, querem ser, ou gostariam de ser. Isto se deve ao **Manifesto Ágil** proposto por Kent Beck e mais dezesseis renomados consultores, desenvolvedores e autores da área de *software* em 2001, Pressman (2011, p. 81).

Beck e os dezesseis profissionais de TI se reuniram para juntos encontrar e (com base na grande experiência de todos) disseminar as melhores práticas da engenharia de *software* e assim contribuir com a melhoria nos processos de desenvolvimentos já existentes e tão conhecidos.

Com base neste manifesto o que se preconiza a partir de então é o valor do indivíduo para com o seu trabalho de desenvolvimento, conforme segue abaixo:

- **Indivíduos e interações** do que ferramentas ou processos;
- ***Software* que funcione** ao invés de documentação excessiva;
- **Colaboração dos clientes** do que apenas negociação contratual;

- **Respostas a mudanças** do que seguir planos apenas (BECK, BEEDLE, *et al.*, 2001).

Deste modo o manifesto ágil proposto valoriza mais os itens em negrito do que seus respectivos à direita.

A base então do desenvolvimento ágil é fazer entrega constante de valor agregado e não somente um *release*[2] executável compilado em alguma linguagem computacional.

Ainda com base no manifesto ágil foram elaborados doze princípios que norteiam aqueles que querem desenvolver *software* priorizando a entrega de valor agregado. Como todas as metodologias ágeis seguem estes princípios SOMMERVILLE (2011) define alguns dos princípios que se encaixam de forma igual independente da metodologia, conforme TABELA 1.

**TABELA 1 - Os princípios dos métodos ágeis**

| Princípios | Descrição |
|---|---|
| **Envolvimento do cliente** | Deve estar ligado ao projeto intimamente e conhecer o processo. Fornece os requisitos e ajuda a priorizar estes avaliando estes requisitos ao final de cada iteração. |
| **Entrega incremental** | O *software* é desenvolvido em incrementos com o cliente, especificando os requisitos para serem incluídos em cada um. |
| **Pessoas, não processos** | As habilidades da equipe de desenvolvimento devem ser reconhecidas e exploradas. Membros da equipe devem desenvolver suas próprias maneiras de trabalhar, sem processos prescritivos. |
| **Aceitar as mudanças** | Deve-se ter em mente que os requisitos do sistema vão mudar. Por isso, projete o sistema de maneira a acomodar essas mudanças. |
| **Manter a simplicidade** | Focalize a simplicidade, tanto do *software* a ser desenvolvido quanto do processo de desenvolvimento. Sempre que possível, trabalhe ativamente para eliminar a complexidade do sistema. |

**Fonte: Sommerville (2011, p.40).**

De acordo com Pressman (2011) apud Fowller (2002) um processo de desenvolvimento ágil caracteriza-se pelo relacionamento com diversos preceitos-chave acerca da maioria dos projetos de *software*, conforme segue:

- É difícil definir e priorizar quais os requisitos do cliente sofrerão mudanças durante o processo de desenvolvimento, bem como, quais requisitos persistirão. Também é difícil antecipar as necessidades do cliente conforme o projeto avança;
- Nos termos do projeto é difícil conduzir os processos, bem como, as atividades pertencentes a ele, pois, muitas vezes as tarefas executadas nas atividades do processo são sucessivas, e, portanto, a

[2] É o termo utilizado para referenciar um produto de *software* executável (binário) em um sistema operacional.

garantia de que ao final o produto entregue é o projetado é inexistente. Não é possível dar a certeza de que o produto está sendo construído de forma correta e que o produto construído é também o correto, pois primeiro se constrói para depois testar, verificar e validar.

- Analisar, projetar, construir (desenvolver) e testar não é tão previsível quanto gostaríamos que fosse segundo o ponto de vista do plano de desenvolvimento.

De todo modo, a agilidade fica evidenciada quando se faz a relação direta com as capacidades e habilidades das equipes, e suas relações com o universo do projeto, enaltecendo os indivíduos e suas capacidades. As equipes passam a ser mais colaborativas, e assim, tornam o produto mais confiável aumentando a satisfação por parte do cliente patrocinador.

As metodologias ou processos de desenvolvimento ágeis não nasceram com a finalidade de se contrapor aos processos tradicionais e mais burocráticos. Ao contrário, nasceram com a finalidade de completar e diminuir aspectos complexos como documentação extensa, por exemplo.

Assim os fatores humanos são extremamente importantes para que a metodologia funcione. Os fatores são relacionados a seguir, como sugere Pressman (2011, p.86):

- **Competência** é um aspecto importante, pois cada indivíduo se posiciona frente ao projeto de acordo com esta e demais habilidades. Podem ser de conhecimento técnico ou principalmente conhecimento do negócio do cliente. Cada indivíduo deve ter a capacidade de disseminar seu conhecimento dentro do grupo ou equipe.
- **Foco comum** deve ser a premissa básica de uma equipe disciplinada, pois deve haver a garantia de que a união e a colaboração atinjam o objetivo que é a entrega do incremento na versão atual do desenvolvimento, mantendo sempre a agregação de valor ao produto final.
- **Colaboração** é totalmente pertinente aos itens anteriores, pois, criar mecanismos e formas de manter a equipe unida e com a comunicação em pleno funcionamento, buscando manter o foco no negócio do cliente.

- **Habilidade na tomada de decisão** faz com que a equipe tenha o poder de decidir o seu próprio destino dentro do processo que desenvolve. Deste modo, a equipe é responsável por aquilo que deliberou em conjunto, ou seja, as responsabilidades estão sempre com a equipe respaldada pelo cliente.
- **Confiança mútua e respeito** faz com que a equipe ágil seja literalmente consistente, ou seja, uma equipe unida e "blindada" ao nível que seja desarticulada por fatores externos Pressman (2011) apud DeMarco e Lister (1998).
- **Auto-organização** no contexto ágil implica em três fatores: (1) a equipe se organiza para realizar o trabalho, (2) a equipe organiza o processo para melhor se adequar ao ambiente local, (3) a equipe organiza seu cronograma para entregar o produto no próximo incremento. Existem diversos outros benefícios da auto-organização, porém, o mais importante talvez seja melhorar a colaboração entre os membros da equipe mantendo-a sempre unida e com o moral elevado.

### *2.3.1 Scrum*

O termo SCRUM é o nome de uma jogada que acontece em uma partida de *rugby*[3] onde a equipe se concentra ao centro do campo de jogo para criar uma estratégia de atingir o objetivo que é conduzir a bola até o outro lado do campo. Com base nesta ideia Jeff Sutherland, no início dos anos 90, juntamente com sua equipe de desenvolvimento criou uma metodologia para o desenvolvimento ágil (incremental) batizando-a com o nome de *Scrum*, Pressman (2011, p. 95).

Como na metodologia de desenvolvimento genérica proposta por Pressman (2011), o *Scrum* também concentra suas atividades de desenvolvimento no alicerce estrutural, ou seja, atividades para levantamento de requisitos, análise, projeto, evolução e entrega do produto, de modo que estas atividades acontecem em um padrão de processo chamado *SPRINT*. A *SPRINT* é um ciclo que dura de 2 a 4 semanas e contempla uma quantidade de requisitos a serem implementados, aqui conhecidos como estórias. As estórias (*stories*) são definidas pelo cliente durante a fase de levantamento de requisitos e assim que são elaboradas passam a compor

[3] É o nome de um esporte parecido com o Futebol Americano (FOOTBALL) onde a equipe tem o objetivo de levar a bola ao fundo do campo adversário marcando assim certa pontuação.

uma lista denominada *PRODUCT BACKLOG* (registro pendente de trabalhos). No *PRODUCT BACKLOG* estão contidas todas as características que o cliente almeja para o seu sistema e assim a cada *SPRINT* uma quantidade de estórias será priorizada e fará parte da lista de itens de trabalho da próxima *SPRINT*, o que leva o nome de *SPRINT BACKLOG*, conforme ilustra a FIGURA 3.

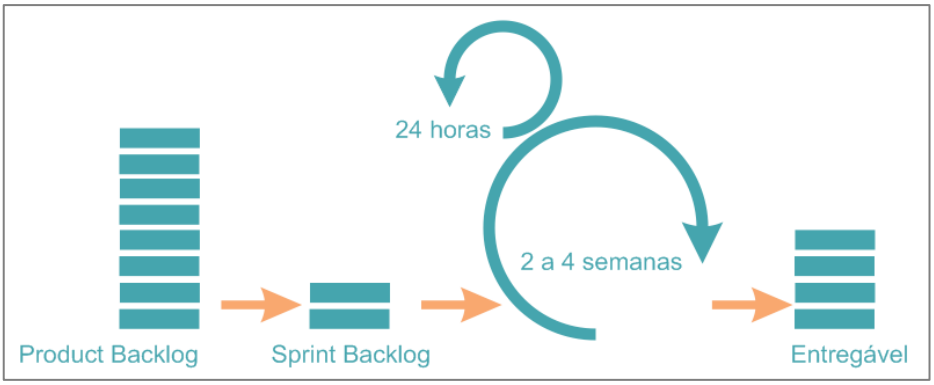


**FIGURA 3 - Fluxo de processo do SCRUM**

**Fonte: Adaptado de ALLIANCE (2011).**

O Scrum é uma metodologia que simplifica os papéis (*roles*) dos envolvidos limitando este número a apenas três.

- ***Product Owner*** é o “dono do projeto”, ou seja, na verdade é o cliente que fornece os requisitos (estórias) para preencher o *PRODUCT BACKLOG*. Normalmente é quem prioriza aquilo que fará parte dos itens de trabalho da *SPRINT* e juntamente com a equipe tem o poder de decidir sobre o ciclo de vida de um requisito no projeto, podendo vir a solicitar mudanças neste requisito, o que no *Scrum* é bem visto e bem vindo.
- ***Scrum Master*** é o indivíduo que tem o papel de negociar com o *Product Owner* os prazos e também a desobstrução dos possíveis impedimentos que por ventura apareçam e que possam de algum modo interferir nas atividades realizadas pela equipe.
- ***Team*** (equipe) são todas as pessoas que fazem parte do time de desenvolvimento, cada qual com seu papel definido (programadores, testadores, analistas, arquitetos, engenheiros, etc). Normalmente as equipes são compostas de cinco a sete pessoas e estas pessoas juntas coordenam as atividades de desenvolvimento do projeto segundo suas

habilidades e capacidades, conforme sugere o manifesto ágil. Adaptado de (ALLIANCE, 2011).

O *Scrum* segue os princípios ágeis e com isso é extremamente importante que sejam aprendidas lições a cada *SPRINT* ou projeto. É por este motivo que a cada dia é feita a reunião diária (*Daily Scrum*), uma cerimônia do *Scrum*, que dura aproximadamente quinze minutos e tem a finalidade de responder a três questionamentos: O que eu fiz ontem? O que eu farei hoje? Quais as barreiras ou impedimentos tiveram que serem eliminadas? Esta reunião acontece diariamente e é promovida e coordenada pelo ***Scrum Master***, e onde é feita a atualização do *BURNDOWN CHART*, gráfico que mostra o progresso da equipe rumo ao alvo definido para a *SPRINT*.

Ainda existem duas outras cerimônias no *Scrum*, que acontecem no início e no final da *SPRINT*, respectivamente. Antes de iniciar a *SPRINT* é realizada a reunião de planejamento, que consiste em definir quais itens de trabalho farão parte da *SPRINT BACKLOG*. Ao final de cada *SPRINT* acontece a *SPRINT REVIEW*, que dura aproximadamente quatro horas e tem por objetivo levar ensinamentos obtidos com os erros ou fortalecer a equipe através dos acertos. Nesta hora é que a equipe se mostra auto organizada e madura.

### *2.3.2 Extreme Programming (XP)*

O *Extreme Programming* ou simplesmente XP, como é mais conhecido, nome dado por Kent Beck, é uma metodologia ágil de desenvolvimento de *software* que segue os princípios definidos no manifesto ágil. O primeiro projeto a utilizar os princípios do XP é datado de 1996, adaptado de (WELLS, 2009).

Sua filosofia é levar o processo de desenvolvimento ao nível extremo principalmente pelo fato de ser um modelo iterativo (SOMMERVILLE, 2011).

Assim como no *Scrum*, os requisitos do cliente são sempre definidos com a participação do próprio cliente e são representados em forma de cenários ou histórias. Cada história é decomposta em partes menores chamadas de tarefas. É com base nas tarefas que os programadores começam a construir os *releases* (entregáveis como produto ao final da iteração) e as atividades de codificação são realizadas muitas vezes em pares, ou seja, o código é gerado sob o ponto de vista

de dois programadores o que na visão do XP diminui consideravelmente riscos de introdução de falhas simples, muitas vezes não detectadas nos testes unitários (por às vezes não serem executados). Desta forma a propriedade é definida como **Propriedade Coletiva**, mais uma característica do XP, e assim pode haver entregas em pequenos espaços de tempo, incrementando o produto final, adaptado de (SOMMERVILLE, 2011).

Algumas práticas do XP são definidas por Sommerville (2011) conforme segue abaixo:

- **Desenvolvimento incremental** que é sustentado pela entrega frequente de *releases*, muitas vezes pequenos *releases*. As funcionalidades (cenários ou histórias) são selecionadas para a iteração com base na priorização realizada em conjunto com o cliente.
- O **cliente sempre faz parte do processo**, pois é encorajado e engajado a participar pelos membros da equipe de projeto, o que de fato é muito importante e previsto no manifesto ágil.
- **Valor** para **pessoas** e não para **processos**, é sustentado pelo fato da atribuição aos programadores realizarem a codificação em pares, o que de alguma forma diminui riscos de falhas simples introduzidas na fase de codificação.
- **Mudanças** ou **melhorias** são vistas como **evolução**, e no XP, assim como no *Scrum*, não são motivo de grande preocupação para as equipes ágeis.
- A **manutenção** é sempre feita ao passo que o desenvolvimento acontece, por meio da **refatoração**, o que vai impactar consideravelmente na qualidade do produto entregue ao cliente.

Com base nas características do XP, mostradas anteriormente, o ciclo das atividades no processo são ilustradas conforme a FIGURA 4.

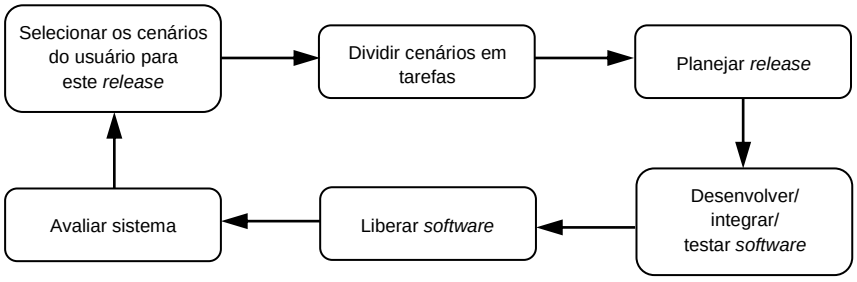


**FIGURA 4 - O ciclo de um release em Extreme Programming.**
**Fonte: (SOMMERVILLE, 2011, p. 44)**

As equipes que adotam o XP como método ou processo de desenvolvimento seguem a risca todos os princípios já mencionados anteriormente e o cliente faz parte de todo este processo desde o início.

A principal característica do cliente no processo, é que ele fornece os requisitos para que sejam implementadas as do sistema. Deste modo no XP, como dito anteriormente, os requisitos são cenários ou histórias, e que representam aquilo que o sistema tem como característica (funcionalidade – requisito funcional). É preciso então entender como o cenário funciona e como é decomposto em tarefas para então fazer parte de um ciclo de atividade de construção.

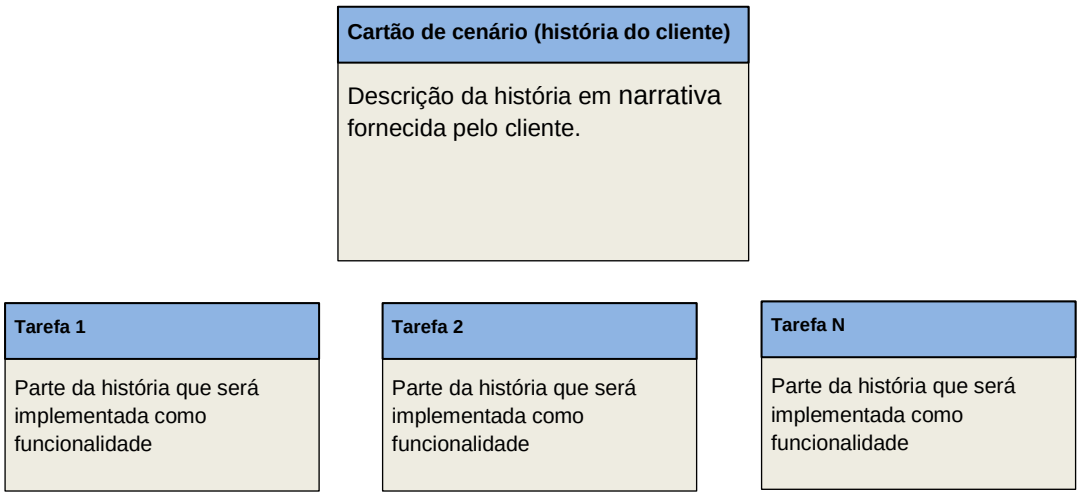


**FIGURA 5 - Cenários do cliente e a decomposição em tarefas.**

Os cenários ou histórias do cliente são entregues a equipe em forma de cartão. Este cartão contém uma narrativa de como o processo ocorre na organização ou negócio do cliente, como ilustra a FIGURA 5. A equipe com posse deste cartão faz a análise e identifica esta funcionalidade em forma de pequenas subpartes do todo já começando o processo de desenvolvimento propriamente. Ao se fazer a análise, se é encontrada alguma inconsistência, o cliente, por fazer parte do processo, já é comunicado, e a reorganização da funcionalidade é feita conforme o entendimento das partes. A FIGURA 5 mostra um exemplo de cartão de cenário (história do cliente) e as suas decomposições em tarefas que serão implementadas e farão parte da entrega do release.

O processo no XP é cíclico e iterativo, portanto a cada volta na iteração é entregue valor ao cliente, produto de *software* funcionando.

## 2.4 QUALIDADE DE *SOFTWARE*

Este tópico deste trabalho não tem como objetivo explanar sobre todos os pontos que permeiam a qualidade do *software*, mas sim definir quais aspectos da qualidade se aplica ao *software*, de um modo geral.

Para Hélio Engholm Jr (2010, p.275) a qualidade de *software* está diretamente ligada ao produto entregue para o cliente, onde suas expectativas são satisfeitas e em conformidade com o que foi acordado no início do projeto.

Ainda para Hélio Engholm Jr (2010) mencionando a norma ISO 9000, a qualidade “é o grau em que um conjunto de características inerentes a um produto, processo ou sistema cumpre os requisitos inicialmente estipulados para estes, podendo ser vista como conformidade aos requisitos do projeto” (ENGHOLM, 2010, p. 275).

A premissa da qualidade era, até pouco tempo, uma forma até de marketing para se vender um produto, hoje em dia isto é uma obrigação para quem entrega este produto, processo ou sistema.

A qualidade de *software*, sob o ponto de vista da engenharia de *software*, não se refere apenas ao produto entregue estar em conformidade com o projeto, mas também de que as atividades metodológicas aplicadas no processo de desenvolvimento seguem padrões ou boas práticas para a construção.

Um produto para ter a qualidade desejada deve ser concebido de forma a atender suas necessidades e prevenir os possíveis problemas, ao invés de ter que corrigi-los. Para garantir isto, existe na equipe de desenvolvimento uma área especializada na garantia da qualidade de *software*, chamada de SQA (*Software Quality Assurance*).

Esta área é responsável por verificar dentro do processo se as práticas executadas atendem os preceitos estipulados pelos modelos de maturidade como CMMI (*Capability Maturity Model Integration*) e MPS.Br (Melhoria do Processo de *Software* Brasileiro), por exemplo.

Pressman (2011, p.361) menciona a proposta de McCall, Richards e Walters (1977) que criaram uma categorização de alguns fatores que afetam a qualidade para o produto de *software*:

- Correção é o quanto um sistema atende as especificações e objetivos da missão do cliente.

- Confiabilidade é a capacidade que o programa tem de realizar as tarefas propostas e a que precisão.
- Eficiência refere-se à quantidade de recursos computacionais e de código exigidos para realizar a sua função.
- Integridade é a capacidade que o sistema tem de controlar a quantidade de dados produzidos que pode ser acessado por usuários não autorizados.
- Usabilidade é o esforço necessário para aprender, operar, preparar a entrada de dados e interpretar as saídas geradas pelo sistema.
- Facilidade de manutenção é o quão de esforço é feito para localizar e corrigir os problemas.
- Flexibilidade é a capacidade empregada na modificação de um sistema em operação.
- Testabilidade esforço necessário para testar um programa de modo a garantir que ele desempenhe a função destinada.
- Portabilidade esforço necessário para portar a estrutura de um *hardware* ou *software* para outro.
- Reusabilidade o quanto de um programa ou parte dele pode ser reutilizado em outro ou em parte do outro.
- Interoperabilidade capacidade ou esforço de um programa interagir ou se integrar a outro (PRESSMAN, 2011, p. 362).

Estes são alguns dos fatores que influenciam na qualidade de um sistema de *software* proposto por Pressman (2011) apud. McCall et. *al.* (1977), e determinam um modelo conforme mostra FIGURA 6.

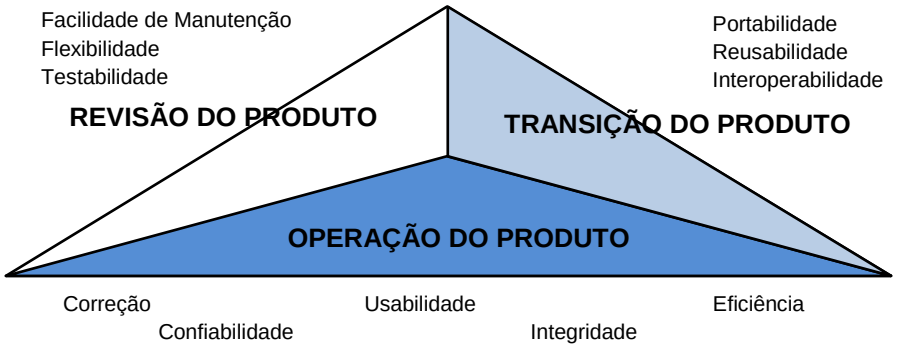


**FIGURA 6 - Fatores de qualidade de *Software* de McCall**

**Fonte: (PRESSMAN, 2011, p. 362)**

## 2.5 TESTE DE *SOFTWARE*

Uma das atividades da engenharia de *software* que está diretamente ligada à qualidade é a atividade de teste. Testar é um mecanismo de verificar se um sistema está realmente executando aquilo que lhe é determinado a executar e se sua execução não causa nenhum tipo de anomalia que diga respeito à saída produzida pelo requisito implementado (*software*).

O processo de teste visa atingir dois objetivos distintos, segundo Ian Sommerville (2011, p.144): demonstrar ao desenvolvedor e ao cliente que o sistema atende aos seus requisitos e demonstrar que o *software* não se comporta de maneira não correta, não desejável ou de forma diferente do que foi especificado (SOMMERVILLE, 2011).

A proposta do teste de *software* é analisar um conjunto de entradas que serão processadas pelo sistema em forma de uma “caixa-preta”, ou seja, não é importante saber como essas entradas são processadas, sob o ponto de vista do usuário, apenas que sejam processadas, e gerar um conjunto de informações de saída, ou seja, o resultado do processo de transformação. Ian Sommerville (2011) propõe um diagrama que ilustra esta ideia de teste, conforme mostra a FIGURA 7.

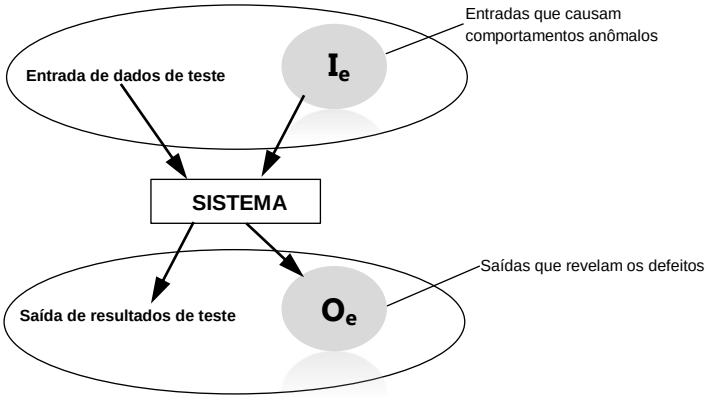


**FIGURA 7 - Um modelo de entrada-saída de teste de programa.**
**Fonte: Ian Sommerville (2011, p. 145).**

Para entender melhor o processo de teste é preciso compreender a ideia do diagrama mostrado na FIGURA 7. Um conjunto de entradas é preparado para ser introduzidas no sistema a ser testado, $\mathbf{I_e}$ (*input*). Este conjunto é formado por dados específicos que podem causar erro, ou seja, de acordo com o requisito especificado é elaborado um caso de teste que possa dar resposta aos estímulos informados pelo

conjunto de entrada. Na extremidade oposta um volume de dados é obtido como resultado do processamento executado **$O_e$** (*output*). Entre as duas extremidades do processo de teste está o **SISTEMA**, a caixa-preta, que contém a implementação do requisito testado.

Para Pressman (2011) uma boa estratégia de teste deve acomodar em nível mais baixo de teste, se um pequeno fragmento de código, algoritmo, esta implementado de forma correta e testes em níveis mais altos para validar as funcionalidades do sistema de acordo com seus requisitos.

O teste de *software* também tem por objetivo realizar a verificação e a validação (V&V) Pressman (2011) apud Boehm (1981), que é uma forma de testar a construção do produto de forma a verificar se o produto está sendo construído de forma correta e se está construindo o produto correto. Em outras palavras o teste de verificação é a garantia de que o *software* implementa as características (funcionalidades) especificadas, enquanto a validação é um conjunto de tarefas que garantem se o *software* implementado pode ser rastreado segundo os requisitos fornecidos pelo cliente (PRESSMAN, 2011, p.402).

Segundo Pressman (2011) a estratégia de testes pode ser vista ou entendida como um espiral, partindo do centro para a extremidade ilustra as fases do teste dentro do processo. A FIGURA 8 mostra a ideia proposta por Pressman.

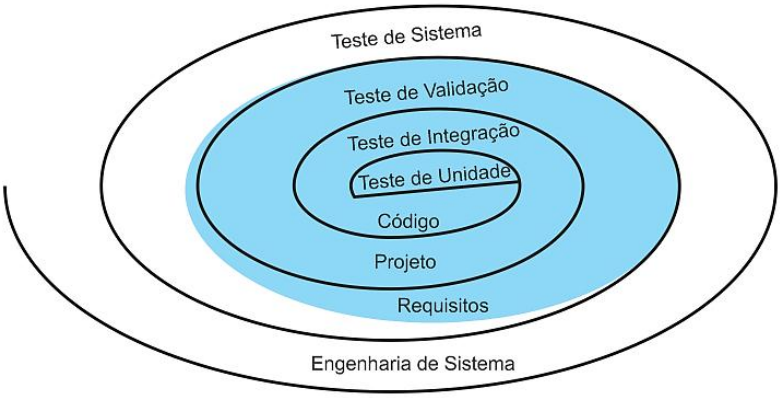


**FIGURA 8 - Estratégia de teste**
**Fonte: (PRESSMAN, 2011, p. 404)**

Os testes unitários estão ao centro do espiral e representam os pequenos fragmentos de requisitos implementados, ou seja, código fonte. Conforme se avança no espiral pode se ver que a fase de projeto relaciona-se com os testes de

integração, ou seja, o produto criado deve ser testado a cada novo *release* integrado, garantindo que as novas funcionalidades não causarão erros nos fragmentos ou módulos de sistema já implementados. A validação, já mencionada anteriormente, está ligada aos requisitos do sistema e ao passo que o desenvolvimento evolui os pontos de verificação validam a construção. Por fim o teste de sistema verifica de um modo global se todas as funções e elementos de sistema se combinam a ponto de materializar o produto de *software* propriamente dito.

Conforme a estratégia é possível identificar etapas que são executadas com sob um ponto de vista procedimental onde quatro etapas executadas sequencialmente, conforme Pressman (2011). A FIGURA 9 ilustra as etapas do processo de teste partindo do ponto mais baixo (fragmentos unitários) até chegar ao teste de requisitos (ordem superior).

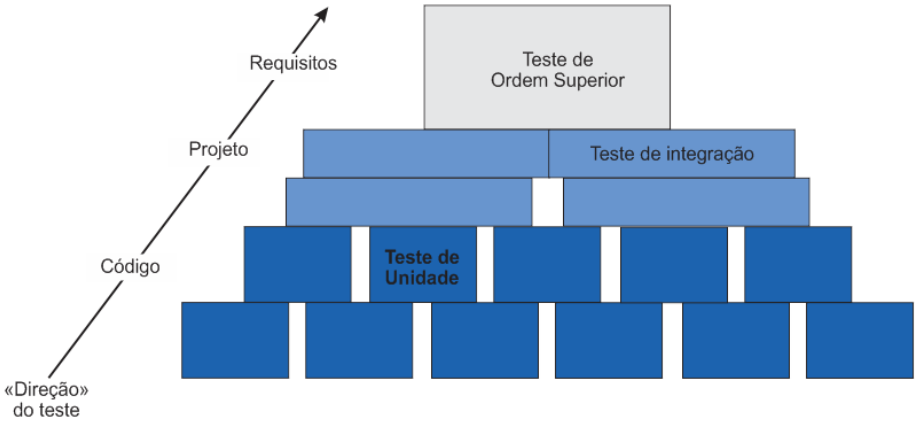


**FIGURA 9 - Etapas de teste de *software***
**Fonte: (PRESSMAN, 2011, p. 405)**

De um modo geral o teste de *software* deve ainda prever testes não apenas no produto, mas também no processo. Para que um *software* atinja os níveis de qualidade é extremamente importante testar outros elementos como, arquitetura base do sistema, requisitos funcionais, não funcionais e teste de estrutura dos métodos e ferramentas de desenvolvimento.

### *2.5.1 Teste de Software em métodos ágeis*

Os testes em metodologias ágeis normalmente seguem padrões bastante diferentes dos testes em métodos tradicionais, devido ao fato de que nos métodos

ágeis ou iterativos não existe a extensa documentação que possa servir de entrada nos planos de teste formais.

O teste, mais precisamente no XP, aborda a atividade de uma maneira um pouco mais informal, ou seja, as atividades de teste fazem parte do processo de desenvolvimento e ocorrem em paralelo, ou até mesmo, antes de ser codificado um requisito (história ou cenário do usuário).

Segundo Caetano (2007), em XP usa-se o desenvolvimento guiado ao teste TDD (*Test Driven Development*) para escrever um teste unitário aplicado ao requisito antes mesmo deste ser codificado. Isso propicia a criação de um ambiente específico para uma boa prática de refatoração (*refactoring*).

Para Sommerville (2011) como os modelos de processo de desenvolvimento incremental seguem padrões um tanto informais para documentação, não há equipes de teste externas promovendo a execução dos planos de teste, pois, não há planos de teste. Desta maneira XP enfatiza de maneira importante os testes do programa reduzindo as chances de erros não previstos ou desconhecidos na versão atual do programa.

Sommerville (2011) ainda destaca quatro características dos testes em XP:

1. Desenvolvimento *test-first* (teste primeiro);
2. Desenvolvimento de teste incremental a partir de cenários;
3. Envolvimento dos usuários no desenvolvimento de testes de validação;
4. Uso de *frameworks* para testes automatizados (SOMMERVILLE, 2011, p. 47).

A caraterística de *test-first* é uma das inovações no desenvolvimento baseado em XP. Realizar os testes antes mesmo de se ter a implementação ajuda muito, pois, ao passo que se codifica, também se testa o código escrito. Há um ganho razoável em relação aos métodos tradicionais, uma vez que os erros podem ser detectados nas fases que antecedem a implementação, tornando o custo para correções muito baixo ou quase inexistente.

No *Extreme Programming* os requisitos são escritos na forma de histórias do usuário, através dos cartões de cenários, o que facilita também a abordagem do *test-first*, pois, a decomposição em tarefas permite que sejam escritos testes para cada tarefa (SOMMERVILLE, 2011, p. 47).

O processo de teste em métodos ágeis, como o XP, por exemplo, leva muito em consideração o teste de aceitação como um parâmetro de qualidade. O cliente

participando do processo ajuda também na elaboração dos critérios para aceitação e o nível de qualidade desejado. Segundo Sommerville (2011) isso nem sempre é possível, já que o cliente está envolvido com o negócio e tem pouco tempo para apoiar esta atividade. A falta de comprometimento muitas vezes pode acabar fazendo com que o cliente fique relutante ao processo de teste.

Caetano (2007) em seu artigo sobre o papel dos testadores em processos ágeis sugere algumas características para o teste ágil:

- Nos testes de aceitação as histórias do cliente vão ser decompostas em tarefas. Os programadores e o cliente, às vezes não têm a capacidade ou maturidade para conhecer um bom parâmetro para estabelecer os critérios de aceitação e qualidade. É indicado que seja feito por programadores ou membros da equipe com mais experiência e que conheçam outras partes do teste, como testes de carga, performance, usabilidade, etc.
- Redundância é a palavra de ordem para o testador ágil, pois ele não fica satisfeito com apenas uma abordagem de teste, é preciso esgotar as possibilidades para obter a garantia sobre o teste.
- Integração contínua com as partes já implementadas e a verificação se as novas funcionalidade não introduziram defeitos ou falhas naquilo que já estava implementado. Sempre que é submetido um código para um servidor de integração, acoplado a um sistema de versionamento, são executados testes de unidade antes de gerar um *build* (produto).
- Testes automatizados, se possível, caso não seja, executar testes manuais que garantam a integridade do produto submetido ao sistema de integração contínua.

### *2.5.2 Desenvolvimento guiado pelo teste (TDD)*

O teste guiado pelo desenvolvimento TDD (*Test-Driven Development*) é uma abordagem que se divide, horas em atividades de teste, horas em atividades de desenvolvimento de código Sommerville (2011) apud Beck (2002); Jeffries e Melnik (2007).

Basicamente o TDD é uma forma de desenvolvimento onde o código é escrito através de uma especificação de teste, em outras palavras, através do teste do requisito é feita a implementação deste. Desta maneira o incremento só é feito quando aquele requisito for aprovado pelos testes executados (SOMMERVILLE, 2011).

Sommerville (2011) sugere etapas do processo do TDD, mostrados a seguir:

- Inicialmente é identificado um incremento, uma nova funcionalidade é deve ser codificada em poucas linhas.
- É escrito então um teste para esta funcionalidade e este teste é implementado através de uma ferramenta automatizada (e.g. JUnit). Assim a ferramenta é capaz de revelar se o requisito pode passar nos teste e relatará o *status*, passou ou não passou.
- É executado os testes juntamente com todos os outros testes criados e a resultante é uma falha, uma vez que a funcionalidade não foi implementada, isto é proposital, pois mostra que o teste acrescentou mais alguma coisa ao conjunto de testes.
- É implementada a funcionalidade e submetida ao teste novamente, podendo ocasionar a refatoração ou melhoria do código existente, para então ser incluído nas funcionalidades já implementadas e testadas.
- Assim que os testes forem executados, com o status que passou pelos testes, é possível então desenvolver mais uma parte da funcionalidade (SOMMERVILLE, 2011).

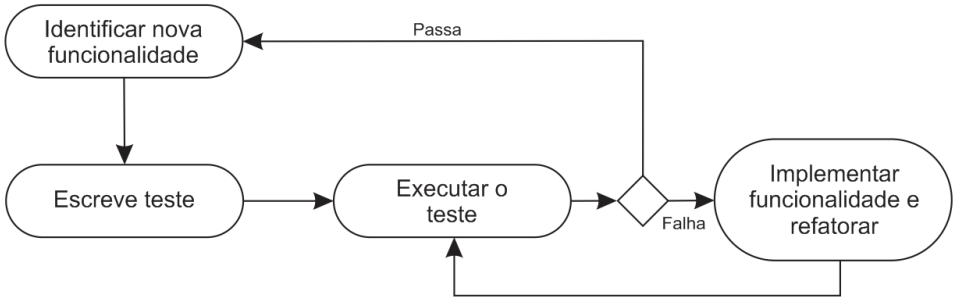


**FIGURA 10 - Desenvolvimento dirigido a testes.**
**Fonte: (SOMMERVILLE, 2011, p. 155)**

Uma das premissas do TDD é que os testes devem ser automatizados, sempre que possível. As ferramentas de automação de testes ajudam a construir

meios para apoiar esta atividade realizando a verificação da codificação em sua menor parte, a unidade (teste unitário).

Os testes dirigidos ao desenvolvimento ajudam os desenvolvedores no aspecto qualidade do código, mas não apenas nisso. Segundo Sommerville (2011), quando um programador conhece o objetivo do código que está sendo escrito, ele tem mais segurança para codificar. Em outras palavras, a base do TDD é que o desenvolvedor deve saber escrever um teste para a funcionalidade que vai codificar, pois, sem saber escrever um bom teste, dificilmente este código será escrito de forma a atender a sua especificação.

Para Scott W. Ambler (2011) uma vantagem significativa em usar o TDD é que o desenvolvedor habilita a capacidade de ter pequenos trechos de código sendo verificados ao passo que são desenvolvidos (princípio do *test-first*). Com isso a automação dos testes via ferramentas de análise de unidade (xUnit), permitem a evolução da funcionalidade com a garantia de terem sidos executados os testes nas menores partes da implementação. A FIGURA 10 ilustra um esquema de utilização de um *framework* para testes unitários, conforme (AMBLER, 2011).

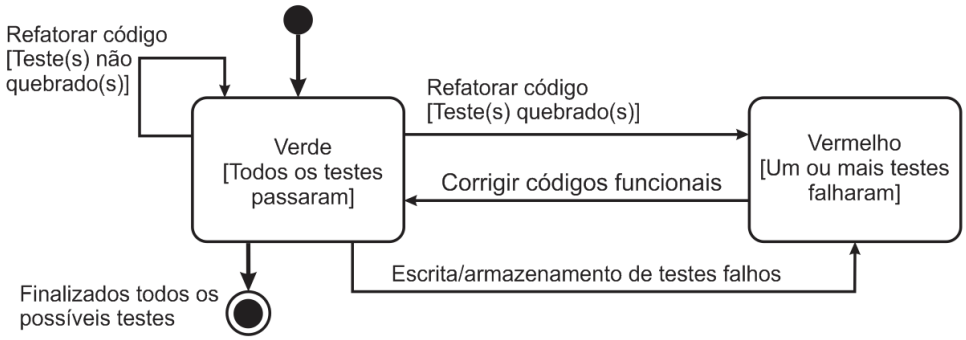


**FIGURA 11 - Teste em TDD via ferramenta xUnit (Diagrama de estados).**

**Fonte: Adaptado de AMBLER (2011).**

Sommerville (2011) ainda aponta alguns benefícios do desenvolvimento dirigido ao teste:

- **Cobertura de código**, é a garantia de que todo código foi testado, mesmo que tenha sido um pequeno trecho, uma vez que o código é proveniente de um teste e foi codificado após ser testado.
- **Depuração simplificada** acontece naturalmente, pois quando um teste falha após tiver sido codificado um requisito, a probabilidade do código

que acabou de ser escrito conter erros é muito alta, o que também ajuda no tempo de localização e correção deste erro. Isto quase que elimina a necessidade de utilização de ferramentas de depuração para fazer a busca dos erros no código. Alguns relatos de equipes que utilizam a prática do TDD informam que é quase desnecessário o uso das ferramentas de depuração Pressman (2011) apud Martin (2007).

- **Teste de regressão** é extremamente necessário realizar nas funcionalidades que já estão implementadas, para que seja verificada se não houve introdução de falhas nas funcionalidades já codificadas.
- **Documentação de sistema**. Os testes servem também como uma forma de documentar, uma vez que para serem codificados os requisitos, é necessária a compreensão do que os documentos de teste apontam.

### *2.5.3 Teste guiado pelo comportamento (BDD)*

Para Soares (2011) o BDD (*Behavior Driven Development*), que em português significa desenvolvimento dirigido pelo comportamento, é uma evolução do TDD. É uma técnica que visa integrar as regras de negócio com o desenvolvimento (linguagem de programação), dando ênfase no comportamento e ainda escrevendo os testes antes do código funcional.

O desenvolvedor, através do BDD, faz uso da linguagem narrativa unida à linguagem ubíqua[4] (*Ubiquitous Language*) para escrever os casos de testes da mesma maneira que no TDD, isto mantém o foco no "porque" está sendo escrito o código ao invés de apenas aspectos técnicos, aproximando e mantendo a comunicação entre a equipe de testes e a equipe de desenvolvimento, adaptado de SOARES (2011).

Ainda sobre o conceito, Soares (2009) apud Dan North (2003), sugere que a linguagem aplicada ao BDD seja extraída dos cenários propostos na fase de análise ou levantamento de requisitos. Isto proporciona uma maneira clara de comunicação e entendimento para todos os envolvidos, conforme FIGURA 12.

[4] É um conceito de linguagem introduzida pro Eric Evans em seu livro *Domain Driven Development* onde o domínio do negócio é fundido à linguagem natural formando uma abordagem livre compreensível para todos os interessados (*stakeholders*). Adaptado de (SATO, 2009).

```
Título (uma linha descrevendo a história)

Narrativa:
Como [o papel]
Eu quero [recurso]
Assim que [benefício]

Critérios de Aceitação: (apresentado como Cenários)

Cenário 1: Título
Dado contexto []
  E [um pouco mais de contexto] ...
Quando [eventos]
Então [resultado]
  E [outro resultado ...]

Cenário 2: ...
```

**FIGURA 12 - Modelo de narrativa em linguagem de ubíqua.**

**Fonte: (SOARES, 2011).**

No modelo ilustrado pela FIGURA 12 é possível identificar um texto simples usando uma linguagem comum, compreensiva por qualquer participante do projeto, e, que não faz uso de termos ou jargões técnicos de difícil entendimento.

Soares (2009) menciona que usar o BDD como uma técnica de desenvolvimento dirigido ao comportamento tem vantagens, e, em nenhum momento se contrapõe ao TDD. Ao contrário, Soares (2011) apud North (2003), afirma que BDD complementa as melhores práticas propostas no TDD e assim reafirma o objetivo de se construir produtos mais bem avaliados antes de serem colocados em produção (ambiente de execução do sistema).

Soares (2009) vai além, ele julga que usando o BDD é possível aproximar mais as equipes, desenvolvedores e testadores. É difícil manter essas duas equipes trabalhando juntas com um objetivo comum e em foco o tempo todo. Na maioria das empresas de desenvolvimento que adotam as metodologias ágeis, seguir o manifesto é dever de todos, porém as equipes de teste sempre são vistas como vilãs, ou seja, carregam o estigma de ser opoentes dos desenvolvedores, quando na verdade, são aqueles que estão na equipe para garantir o melhor do produto visando à qualidade, adaptado de SOARES (2009).

Da mesma maneira que Soares (2009) aponta a existência de vantagens em se utilizar o BDD, também coloca em evidência as dificuldades, principalmente para aqueles que não têm prática ou experiência no desenvolvimento dirigido ao teste ou ao comportamento (*behavior*). Talvez a pior das dificuldades seja exatamente o ponto de partida, o começo de um projeto.

Por conta disso, escrever código pouco coeso e de acoplamento alto, e principalmente antes dos testes, às vezes, faz com que o trabalho acabe tendo que ser refeito, diminuindo os benefícios propostos pelo TDD e BDD.

Escrever os testes antes de desenvolver o código funcional é sempre o ponto de partida na aplicação do BDD, por ser também sua característica essencial.

Soares (2011) apud Chelimski (2010) evidencia que as empresas de desenvolvimento encontram muitos problemas de ordem de comunicação e que isso dificulta o processo. Para melhor compreender o exemplo citado por Soares (2011) é que um analista da área de negócios quando vai repassar para os analistas de sistemas, engenheiros ou mesmo os desenvolvedores, uma informação para ser transformada em uma funcionalidade, não vai usar termos técnicos como – “ao realizar a gravação da ordem de compra as informações devem ser persistidas em banco de dados relacional ANSI-Compliant”, por exemplo. Este nível de linguagem pouco tem a ver com o negócio, é mais relacionado com aspectos de ordem estrutural e arquitetural. Ao invés disso, usaria uma informação mais relevante para o negócio, como por exemplo – “ao realizar a ordem compra o sistema deve manter a informação no sistema possibilitando uma releitura dessas em um tempo futuro”.

Isto é o que leva a principal característica do BDD, o comportamento e uso de linguagem natural (ubíqua).

Assim para realizar o desenvolvimento de cenário com base no seu plano de testes usando o BDD é necessário realizar a separação deste cenário em três pontos importantes (comportamento), usando palavras-chave para isso, conforme sugere Soares (2011):

- ***Given*** – dado um contexto;
- ***When*** – quando acontecer um evento;
- ***Then*** – é esperado que alguma coisa aconteça. Adaptado de SOARES (2011).

A FIGURA 13 mostra um cenário do cliente para uma funcionalidade de sistema, neste caso hipotético, para um sistema de reserva de ingressos de uma casa de espetáculos. Neste cenário é possível visualizar o título e a funcionalidade, logo na continuação a decomposição do cenário em casos específicos que serão

usados para aplicação da técnica do BDD, o que acontece neste trabalho na sessão seguinte como estudo de caso.

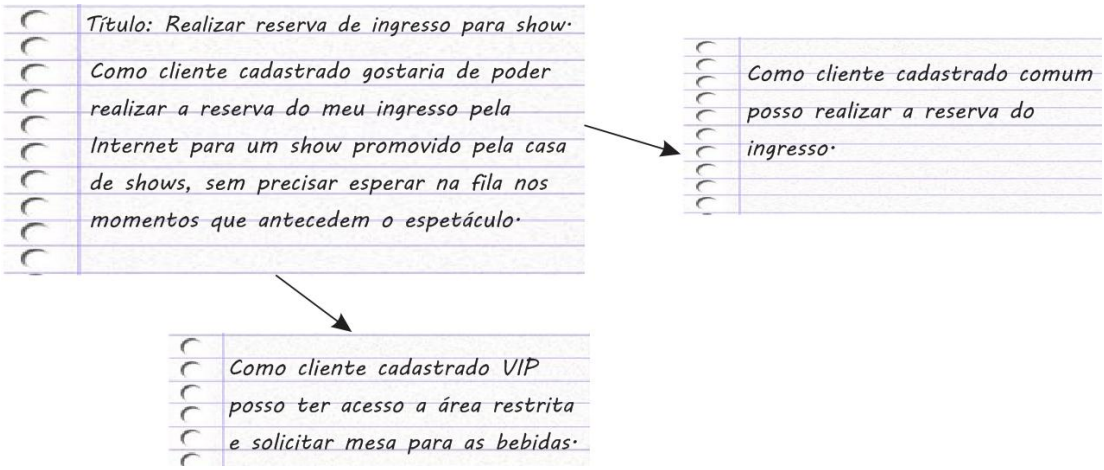


**FIGURA 13 - Exemplo de decomposição do cenário em tarefas para o BDD.**

# 3 PROCEDIMENTOS METODOLÓGICOS DA PESQUISA

Este capítulo visa aplicar em um modelo de estudo de caso a técnica do BDD através do uso do *framework JBehave* e a suíte de testes *JUnit*.

## 3.1 APLICAÇÃO DO BDD (BEHAVIOR DRIVEN DEVELOPMENT)

Como visto no capítulo dois, o BDD é destinado à prática de desenvolvimento guiado pelo comportamento. É preciso definir o comportamento através das palavras-chave que servem como definição de comportamento, de acordo com a TABELA 2.

**TABELA 2 - Termos usados no BDD**

| Palavra-chave (termo) em inglês. | Termo em português | Descrição do comportamento |
|---|---|---|
| ***Given*** | Como [papel] | É o contexto para que seja elaborado o teste. Através deste item é determinado tudo aquilo que circunda o cenário a ser resolvido (implementação). |
| ***When*** | Quando acontecer algo | É o evento a ser considerado para aplicação do teste, ou seja, para o contexto criado o evento a ser analisado é o que estiver contido neste item dentro do plano de teste. |
| ***Then*** | Preceda da seguinte forma | É a ação aplicada na resolutividade do caso/requisito dado pelo contexto. |

Os termos na TABELA 2 serão usados como separação das partes do caso de teste criado para o cenário.

Deste modo o BDD necessita que haja a definição de uma linguagem comum entre àqueles que fazem parte da equipe, desde o cliente até o desenvolvedor. Esta linguagem foi abordada no capítulo dois, a linguagem ubíqua.

O QUADRO 1 ilustra a aplicação da linguagem ubíqua juntamente com os termos aplicados à técnica.

## 3.2 FERRAMENTA JBEHAVE E SUÍTE DE TESTES JUNIT

JBehave é um *framework* para automação do processo de testes em BDD. É uma ferramenta de código aberto e de livre uso. Foi criada justamente para realizar o que propõe o BDD, testes através do comportamento, facilitando a comunicação entre as equipes de desenvolvimento e teste, com participação de membros de outras áreas fora do contexto técnico, adaptado de SOARES (2011).

É um framework escrito na linguagem Java e serve para automação de testes unitários na filosofia BDD através de uma suíte de testes, o JUnit.

A ferramenta é composta por diversas classes e interfaces desenvolvidas com a finalidade de atender aos modelos de teste em BDD de forma que os casos de teste do usuário final sempre farão uso por extensão e implementação destas classes e interfaces.

Ela é acoplada à suíte de testes JUnit (ferramenta para automação de testes unitário em Java que segue o modelo TDD), que acaba simplificando a tarefa de realização de testes unitários.

As características técnicas e detalhes no desenvolvimento com JBehave é abordado na sessão seguinte.

## 3.3 ESTUDO EXPERIMENTAL

O estudo experimental proposto para a demonstração da aplicação do BDD ilustra de maneira rápida e transparente como evoluir de um modelo de requisito escrito (história) para um modelo de implementação de teste por meio do comportamento.

O QUADRO 1 descreve a narrativa em linguagem natural deste estudo de caso, conforme as características do BDD.

```
[Narrativa/História]
No sistema bancário o cliente solicita a realização de depósito em conta poupança. Como se
trata de um banco cooperativista existe a opção de que se o cliente for associado, terá o
acréscimo de 0,05% ao depósito, como forma de incentivo à poupança.
Scenario: Um cliente não associado
Given um cliente não associado com saldo inicial de 300 reais
When é realizado um depósito de 100 reais
Then o saldo é atualizado para 400 reais

Scenario: Um cliente associado
Given um cliente associado com saldo inicial de 300 reais
When o cliente realiza um depósito de 100 reais
Then o saldo da poupança é atualizado para 400.10 reais
```

**QUADRO 1 - Exemplo de narrativa ou cenário de usuário**

Com base na narrativa o estudo de caso começa a ser montado em um projeto escrito na linguagem Java.

Para iniciar a elaboração do projeto é necessário ter as ferramentas instaladas conforme a TABELA 3 - Ferramentas necessárias para o estudo de casoTABELA 3.

**TABELA 3 - Ferramentas necessárias para o estudo de caso**

| Ferramenta | Descrição |
|---|---|
| Java Development Kit 1.6.x – Kit de desenvolvimento Java | É necessário para ambientes de desenvolvimento através da plataforma Java. É o arcabouço com compilador e máquina virtual para execução das aplicações escritas em Java. |
| Eclipse IDE 3.7 | É uma ferramenta de apoio ao desenvolvimento de aplicações que agiliza o processo de codificação e execução em tempo de desenvolvimento. |
| Maven 2.2.1 | É um gerenciador de projetos Java que contém arquétipos (modelos pré-configurados de projeto) com a base pronta dentro de um formato disponível para aplicações de vários tipos em Java (*Web*, *Desktop*, *Mobile*, etc). |
| JUnit 4.8.1 | Framework para desenvolvimento guiado ao teste (TDD) em Java. |
| JBehave 2.3.1 | Framework para desenvolvimento guiado ao comportamento (BDD) em Java. |

A seguir será iniciado um projeto exemplo que servirá como estudo experimental para a aplicação da técnica descrita até aqui.

## 3.4 PROJETO

A primeira etapa do projeto é criar na IDE (*Integrated Developrment Environment*), em português, ambiente de desenvolvimento integrado, um projeto Java padrão (*console*), através do *Maven*. A FIGURA 14 mostra o projeto já criado na estrutura do *Maven*.

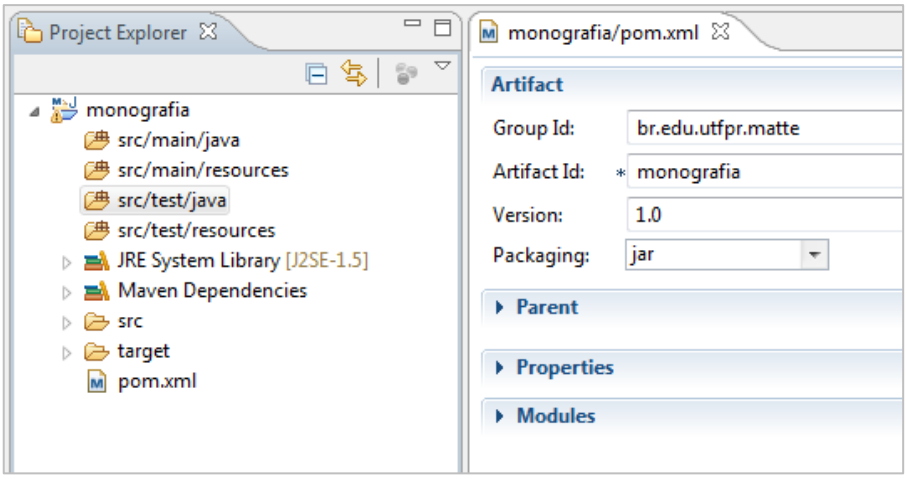


**FIGURA 14 – *Screenshot* da estrutura do projeto de estudo experimental**

O objetivo da FIGURA 14 é apresentar a estrutura de pacotes que é importante, pois o início de um projeto que utiliza o BDD é sempre pela descrição do caso de teste.

O *JBehave* utiliza um padrão de convenções para que seja possível ler o arquivo texto contendo a narrativa, o cenário e os comportamentos (*Given, When e Then*). O arquivo texto deve ficar no mesmo nível de pacote da classe que será lida pela suíte JUnit, para executar os testes.

A FIGURA 15 ilustra o arquivo já com as características de comportamento e cenário implementado.

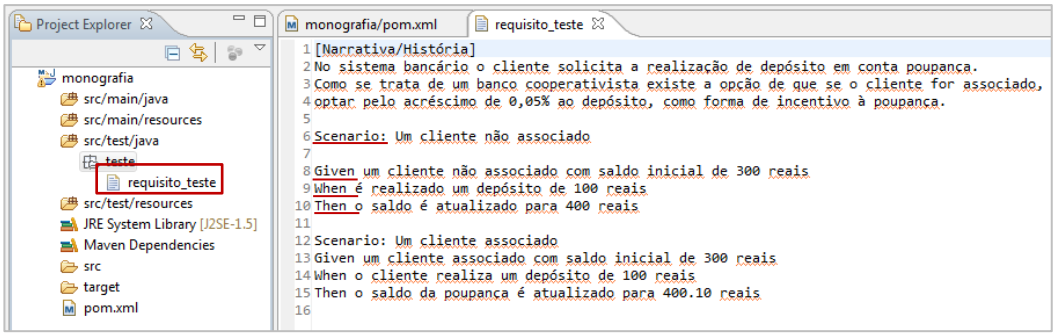


**FIGURA 15 - Arquivo texto com o cenário e os comportamentos**

Como pode ser visto na FIGURA 15 o início do projeto, em termos de implementação, é justamente o caso que descreve o cenário (história/requisito) e os respectivos comportamentos, aqueles listados na TABELA 2, ou seja, *Given*, *When* e *Then*.

Os comportamentos são justamente as entradas para que a ferramenta de automação, no caso, *JUnit*, proceda intermediado pelo *JBehave* a execução dos testes.

Neste caso específico o arquivo de cenário deve estar listado no diretório (pacote) onde a classe responsável pelo teste será criada. Como padrão do *JBehave*, a classe de teste deve ter o mesmo nome do arquivo de cenário, trocando o caractere _ (*underscore*) pelo modelo de código *CamelCase*[5], como ilustra a FIGURA 16.

[5] *CamelCase* é uma forma criada para unir palavras compostas capitalizando a primeira letra de cada palavra exceto a primeira da frase. Adaptado de (SOARES, 2011, p. 78).

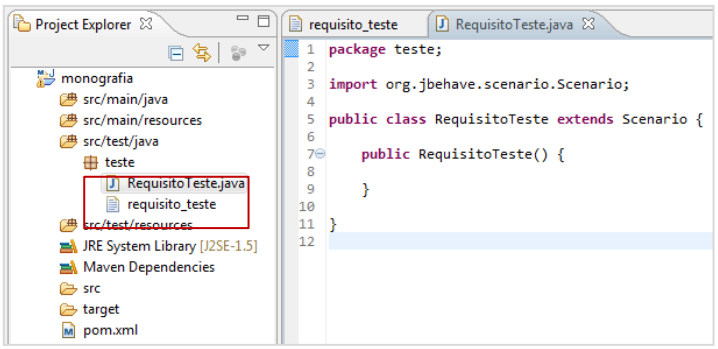


**FIGURA 16 – *Screenshot* do código da classe que representa o cenário**

A classe responsável pelo cenário deve estender da classe **org.jbehave.scenario.Scenario**, que é responsável por fazer a injeção dos comportamentos contidos no arquivo texto. Em seguida é necessário criar uma classe que será usada como repositório dos comportamentos, o que o no *JBehave* é chamado de *Step*, e deve estender da classe *Steps* do pacote **org.jbehave.scenario.steps**. A FIGURA 17 ilustra o diagrama de classes UML com a estrutura das classes do projeto para o teste.

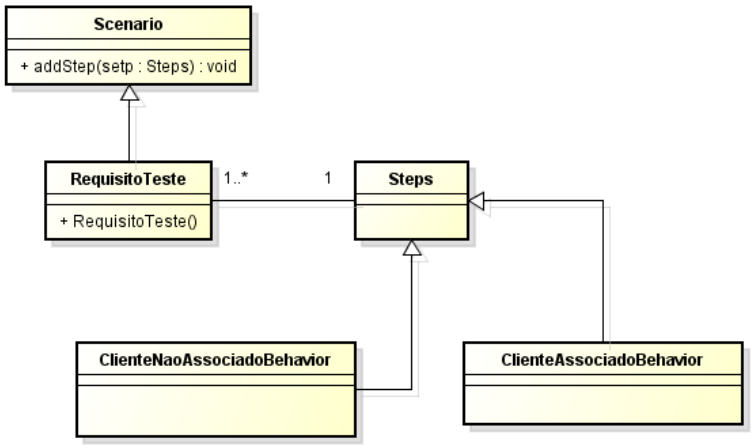


**FIGURA 17 - Diagrama de classes do esquema de teste do projeto**

A estrutura a partir do modelo ilustrado na FIGURA 17 está pronta para iniciar a aplicação do BDD na prática. Com base no modelo de comportamento criado, através do arquivo texto, o requisito pode começar a ser modelado de forma reversa (do comportamento para a modelagem do requisito).

A narrativa que é o cenário principal do projeto descreve o ambiente e quais as partes envolvidas no negócio: o cliente de um banco cooperativo (QUADRO 1). Este cliente possui uma conta que é o objeto manipulado, segundo o requisito solicitado. A FIGURA 18 mostra o diagrama de classes do modelo (entidades).


**FIGURA 18 - Diagrama de classes do modelo (entidades)**

As classes Cliente e Conta, respectivamente, possuem uma associação bidirecional, onde, um cliente pode ter uma conta e esta conta é exclusiva de um único cliente.

Deste modo o requisito hipoteticamente fornecido pelo cliente patrocinador do projeto implicaria em uma regra de negócio simples que faz referencia aos depósitos realizados pelo cliente poupador. Na sequência o QUADRO 2 ilustra o conteúdo da classe Cliente.

```
package pojo;

public class Cliente {

	private String nome;
	private boolean associado;
	private Conta conta;

	public Cliente() {}

	public String getNome() {return nome;}
	public void setNome(String nome) {this.nome = nome;}
	public boolean isAssociado() {return associado;}
	public void setAssociado(boolean associado) {this.associado = associado;}
	public Conta getConta() {return conta;}
	public void setConta(Conta conta) {this.conta = conta;}

}
```

**QUADRO 2 - Classe de modelo (entidade) Cliente**

O QUADRO 3 ilustra o conteúdo da classe Conta que será usada para a aplicação dos testes no *framework JBehave*.

```
package pojo;

public class Conta {
	private Cliente cliente;
	private double saldoInicial;
	private double saldo;

	public Conta() {
	}

	public Cliente getCliente() {return cliente;}
	public void setCliente(Cliente cliente) {this.cliente = cliente;}
	public Double getSaldoInicial() {return saldoInicial;}
	public void setSaldoInicial(Double saldoInicial) {this.saldoInicial = saldoInicial;}
	public Double getSaldo() {return saldo;}
	public void setSaldo(Double saldo) {this.saldo = saldo;}

	public void deposito(double deposito) {
		if (cliente.isAssociado()) {
			saldo = saldoInicial + ((deposito * 0.05) + deposito);
		} else {
			saldo = saldoInicial + deposito;
		}
	}
```

```
}
```

**QUADRO 3 - Classe modelo (entidade) Conta**

A classe conta agora foi implementada seguindo o comportamento definido de acordo com a solicitação hipotética do cliente patrocinador, que de acordo com o tipo de cliente realiza a operação de depósito de forma distinta.

A partir da definição das regras serão implementados os cenários através da ferramenta, ou seja, nas classes **ClienteNaoAssociadoBehavior** e **CleinteAssociadoBehavior**, conforme FIGURA 17.

Estas duas classes serão responsáveis por realizarem a configuração do ambiente de teste do BDD seguindo o arquivo texto criado, conforme ilustrou a FIGURA 15.

Como já mencionado anteriormente a classe **RequisitoTeste** fará a injeção das classes *Steps*, citadas anteriormente.

Para finalizar a etapa de projeto a seguir é mostrada a configuração das classes *Step*, **ClienteNaoAssociadoBehavior** e **CleinteAssociadoBehavior**, respectivamente.

```
package teste;
import org.jbehave.scenario.Scenario;

public class RequisitoTeste extends Scenario {
        public RequisitoTeste() {
                addSteps(new ClienteNaoAssociadoBahavior());
                addSteps(new ClienteAssociadoBehavior());
        }
}
```

**QUADRO 4 - Classe RequisitoTeste**

A classe **RequisitoTeste** que foi mostrada no início da configuração do projeto na FIGURA 16, agora recebe no construtor as instâncias das classes já mencionadas anteriormente, como poder ser visto no QUADRO 4, através do método ***addSteps(...),*** executado pelo *framework JBehave*. Cada classe é um cenário representado no arquivo texto.

```
package teste;

import static org.junit.Assert.assertTrue; ...
//importações ocultadas
public class ClienteAssociadoBehavior extends Steps {
        private Cliente cliente; private Conta conta;

        @Given("um cliente associado com saldo inicial de $saldoInicial reais")
        public void setup(double saldoInicial){
                this.cliente = new Cliente();
                this.conta = new Conta();

                this.cliente.setNome("Marcio Angelo Matté");
                this.cliente.setAssociado(true);
                this.conta.setCliente(cliente);
                this.conta.setSaldoInicial(saldoInicial);
        }
```

```
        @When("o cliente realiza um depósito de $dep reais")
        public void depositarPoupanca(double dep){
                this.conta.deposito(dep);
        }
        @Then("o saldo da poupança é atualizado para $saldo reais")
        public void verifica(double saldo){
                assertTrue(this.conta.getSaldo()==saldo);
        }
}
```

**QUADRO 5 - Classe Step ClienteAssociadoBehavior**

O QUADRO 5 mostra a existência das anotações ***@Given***, ***@When*** e ***@Then*** que serão analisadas pelo framework fazendo com que os textos existentes no corpo da anotação sejam comparados com aqueles que existem no arquivo texto e ainda, os valores contidos no texto original sejam substituídos pelos coringas através precedidos de $, como **@Given("... $saldoInicial ...")**, por exemplo, e assim injetados nos métodos sob o mesmo nome como **setup(double saldoInicial),** por exemplo.

O mostra a classe contendo o cenário para um cliente não associado.

```
package teste;

import static org.junit.Assert.*;
//importações ocultadas
public class ClienteNaoAssociadoBahavior extends Steps {

        private Cliente cliente;  private Conta conta;

        @Given("um cliente não associado com saldo inicial de $saldoInicial reais")
        public void setup(double saldoInicial){
                this.cliente = new Cliente();
                this.conta = new Conta();

                this.cliente.setNome("Marcio Angelo Matté");
                this.cliente.setAssociado(false);
                this.cliente.setConta(conta);
                this.conta.setCliente(cliente);
                this.conta.setSaldoInicial(saldoInicial);
        }
        @When("é realizado um depósito de $dep reais")
        public void depositarPoupanca(double dep){
                this.conta.deposito(dep);
        }
        @Then("o saldo é atualizado para $saldo reais")
        public void verifica(double saldo){
                assertTrue(this.conta.getSaldo()==saldo);
        }
}
```

**QUADRO 6 - Classe Step ClienteNaoAssociadoBehavior**

O QUADRO 6, assim como no QUADRO 5, apresenta um cenário para um cliente não associado e possui as mesmas anotações, porém, um detalhe muito importante é que, os textos contidos nas anotações devem ser diferentes, da mesma maneira como foram escritos no arquivo texto e que assim não haja uma confusão na hora do *framework* realizar a leitura e carga dos valores para aplicar aos métodos dos testes.

A FIGURA 19 mostra como ficou a estrutura do projeto pronto para serem executados os testes e realizar a verificação.

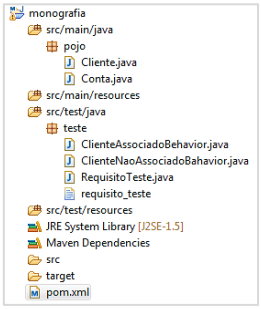


**FIGURA 19 - Estrutura do projeto pronto para ser executado**

# 4 RESULTADOS E DISCUSSÃO

Este capítulo apresenta os resultados dos testes aplicados ao estudo experimental mostrado no capítulo três. Serão abordados aqui quatro testes para verificar se em todos não há falhas quanto à especificação com base no comportamento.

## 4.1 AMBIENTE DE EXECUÇÃO

O primeiro teste foi realizado no trecho de comportamento do cenário um, conforme QUADRO 7.

```
Scenario: Um cliente não associado

Given um cliente não associado com saldo inicial de 300 reais
When é realizado um depósito de 100 reais
Then o saldo é atualizado para 400.0 reais
```

**QUADRO 7 - Cenário um - cliente não associado**

Verificando o QUADRO 7 e fazendo uma relação com a implementação do método da classe **ClienteAssociadoBehavior**, **setup(...)**, o texto do cenário é introduzido no corpo da anotação e o valor escrito em vermelho é injetado na variável **$saldoInicial**, como mostra o QUADRO 8.

```
@Given("um cliente associado com saldo inicial de $saldoInicial reais")
	public void setup(double saldoInicial){
		this.cliente = new Cliente(); this.conta = new Conta();

		this.cliente.setNome("Marcio Angelo Matté");
		this.cliente.setAssociado(true);
		this.conta.setCliente(cliente);
		this.conta.setSaldoInicial(saldoInicial);
	}
```

**QUADRO 8 - Método de configuração - Given**

Com a configuração realizada é executado o teste por meio do *JUnit* e o resultado obtido é o mostrado na FIGURA 20, através da barra verde.

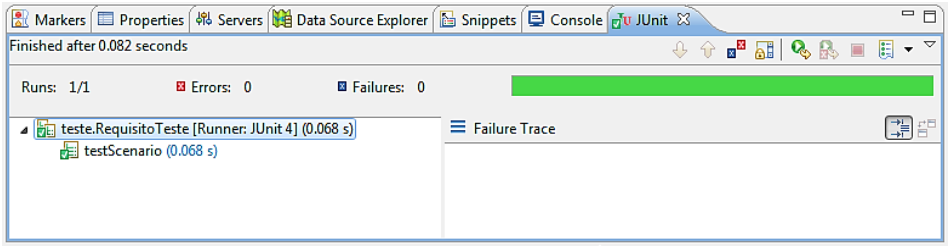


**FIGURA 20 - Teste com sucesso cenário um**

Os testes foram executados com sucesso, aplicando os comportamentos aos objetos gerados no método **setup(...)** que introduziu os valores obtidos através do arquivo texto em **@Given**. O QUADRO 9 mostra os métodos de comportamento da classe.

```
@When("o cliente realiza um depósito de $dep reais")
	public void depositarPoupanca(double dep){
		this.conta.deposito(dep);
	}
	@Then("o saldo da poupança é atualizado para $saldo reais")
	public void verifica(double saldo){
		assertTrue(this.conta.getSaldo()==saldo);
	}
```

**QUADRO 9 - Métodos que aplicam os valores conforme as anotações**

O segundo teste para o primeiro cenário é realizada apenas a alteração do valor de entrada no arquivo texto, ou seja, a estrutura da aplicação não foi alterada, mas sim o arquivo texto.

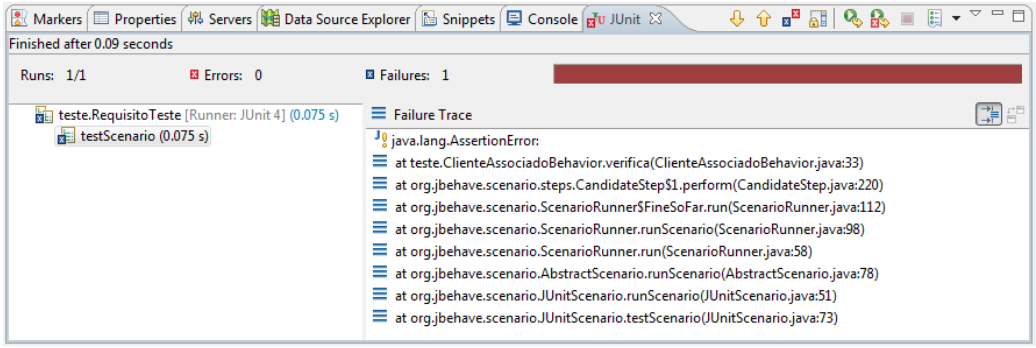


**FIGURA 21 - Teste sem sucesso do cenário um**

No arquivo texto foi alterado o valor para o comportamento ***@Given***, de trezentos (300), para duzentos (200), mantendo os valores ***@When*** e ***@Then***,

inalterados. Isto de fato gerou inconsistência na execução do teste, uma vez que a codificação do método na classe Conta, gera valores diferentes daqueles que estão no momento no arquivo texto, o que pode ser identificado através da FIGURA 21.

Deste modo quando houver a necessidade de realizar testes com valores distintos, basta que sejam introduzidos valores no arquivo texto criado no início do projeto (princípio *test-first*).

Para o segundo cenário a aplicação é a mesma, porém o que muda é a regra de negócio aplicada no método de depósito da classe Conta. Deste modo podem ser criados diversos casos de teste para funcionalidades mesmo antes destas serem codificadas.

## 4.2 RESULTADOS

Fazendo uma análise nos resultados do teste é possível verificar que as configurações podem variar para realizar rapidamente combinações diferentes para valores, e assim preceder com diversos casos diferentes, permitindo que seja feita uma refatoração, se necessário for.

# 5 CONSIDERAÇÕES FINAIS

Este capítulo apresenta as conclusões do trabalho bem como aponta linhas para futuras pesquisas acerca do tema.

## 5.1 CONCLUSÃO

Este trabalho permitiu a exploração uma área que é tão importante no desenvolvimento de *software* e que ainda hoje não tem o respeito devido.

Os aspectos da qualidade sempre fazem parte do cotidiano dos engenheiros, arquitetos, analistas e desenvolvedores de sistemas, mas problemas de ordens diversas, e principalmente de ordem pessoal, afastam os desenvolvedores dos testadores quando deveria ser o contrario. Em toda revisão bibliográfica os autores relatam este problema com base na experiência.

Então aprofundar no mundo do teste principalmente focando no teste em ambientes que seguem as metodologias ágeis, fez com que aumentasse mais o gosto pelo tema o que poderei explorar ainda mais no futuro.

Os principais pontos positivos foram:

- Conhecer o ambiente do “universo” ágil, mesmo que através da literatura ou de pequenos cases acadêmicos ou experiências de outros desenvolvedores;
- Fazer o uso de um processo ágil no quesito entendimento de método de testes;
- Aplicação de uma ferramenta automatizada de testes que segue a fundo o princípio do BDD;
- Poder analisar os resultados, mesmo que em um nível superficial, para elaborar diferentes casos com base nestes resultados.

Principais pontos negativos:

- Um tema relativamente novo, o que dificulta encontrar casos evoluídos para pesquisa;
- Falta de profissionais ou empresas próximas que pudessem servir de campo exploratório e fonte de pesquisa;
- Literatura escassa.

## 5.2 TRABALHOS FUTUROS

Este trabalho foi um começo na pesquisa da aplicação do BDD em teste ágil, servindo como um ponto de partida.

O tema é vasto do ponto de vista da aplicação do teste de *software*, mesmo que técnicas novas venham a surgir, e pode ser explorado mais profundamente sendo posto em prática em empresas de desenvolvimento ou até mesmo em bancos das academias.

Foco nas ferramentas automatizadas é um passo futuro que pode ser estudado focando na melhoria da aplicação das técnicas e obtendo resultados mais confiáveis como resultante dos estudos.

## REFERÊNCIAS

ALLIANCE, S. Scrum Is an Innovative Approach to Getting Work Done. **What is SCRUM**, 2011. Disponivel em: <http://www.scrumalliance.org/pages/what_is_scrum>. Acesso em: 02 Novembro 2011.

AMBLER, S. W. Introduction to Test Driven Development (TDD). **Introduction to Test Driven Development (TDD)**, 2011. Disponivel em: <http://www.agiledata.org/essays/tdd.html>. Acesso em: 16 Novembro 2011.

BECK, K. et al. Manifesto for Agile *Software* Development. **Manifesto for Agile *Software* Development**, 2001. Disponivel em: <http://agilemanifesto.org/>. Acesso em: 08 Novembro 2011.

CAETANO, C. Testes Extremos - Entenda o papel do testador em projetos ágeis. **Linha de Código**, 2007. Disponivel em: <http://www.linhadecodigo.com.br/artigo/1229/Testes-ExtremosEntenda-o-papel-do-testador-em-projetos-ágeis.aspx>. Acesso em: 02 Novembro 2011.

ENGHOLM, H. J. **Engenharia de *Software* na prática**. 1a. ed. São Paulo: Novatec, v. I, 2010.

KOCIANSKI, A.; SOARES, M. D. S. **Qualidade de *Software***. 2a. ed. São Paulo: Novatec, v. I, 2007.

PRESSMAN, R. S. **Engenharia de *Software* - uma abordagem profissional**. 7a. ed. Porto Alegre: AMGH Bookman, v. I, 2011.

SATO, D. Introduzindo Desenvolvimento Orientado por Comportamento (BDD). **Introduzindo Desenvolvimento Orientado por Comportamento (BDD)**, 2009. Disponivel em: <http://www.dtsato.com/blog/work/introduzindo_desenvolvimento_orientado_comportamento_bdd/>. Acesso em: 02 Novembro 2011.

SOARES, I. Desenvolvimento orientado por comportamento (BDD) - Um novo olhar sobre o TDD. **Java Magazinne**, Rio de Janeiro, v. I, n. 91, 2011. ISSN 1676836-1.

SOMMERVILLE, I. **Engenharia de *Software***. 9. ed. São Paulo: PEARSON, v. I, 2011.

WELLS, D. Extreme Programming. **Extreme Programming:** A gentle introduction, 2009. Disponivel em: <http://www.extremeprogramming.org>. Acesso em: 09 Novembro 2011.