KELLY BEZERRA DA SILVA

ANÁLISE DE BOAS PRÁTICAS PARA *O TEST-DRIVEN DEVELOPMENT* VISANDO A QUALIDADE DE *SOFTWARE*

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para conclusão do curso de MBA em Tecnologia de *Software*.

São Paulo
2012

KELLY BEZERRA DA SILVA

ANÁLISE DE BOAS PRÁTICAS PARA *O TEST-DRIVEN DEVELOPMENT* VISANDO A QUALIDADE DE *SOFTWARE*

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Tecnologia de *Software*.

Área de Concentração: Tecnologia de *Software*

Orientador: Prof. Dr. Kechi Hirama

São Paulo
2012

DEDICATÓRIA

Dedico este trabalho em primeiro lugar a Deus que sempre me dá forças para alcançar meus objetivos e aos meus pais que me apoiam a cada decisão tomada.

AGRADECIMENTOS

À Universidade de São Paulo – USP por abrir espaço para este curso.

À Escola Politécnica da Universidade de São Paulo – EPUSP que forneceu toda estrutura necessária ao longo desses 24 mêses.

Ao PECE – Programa de Educação Continuada em Engenharia que elaborou um curso que atendia às minhas espectativas.

Ao meu orientador Prof. Dr. Kechi Hirama pela paciência, suporte e incentivo durante o período de produção deste trabalho.

A minha amiga Patrícia Regina que me aconselhou a procurar o curso.

Aos meus amigos que insistem em minha companhia apesar dos vários convites recusados para me dedicar a este curso.

A todos que me ajudaram a concretizar este trabalho, avaliando, opinando, corrigrindo ou qualquer que fosse o auxílio.

RESUMO

O *Test-Driven Development* é uma técnica de desenvolvimento de *software* popularizada com o surgimento dos métodos ágeis que tem como principal objetivo entregar código limpo que funcione.

O seu uso se dá por uma técnica aparentemente simples, mas que exige muita disciplina e aderência aos passos por ela propostos.

Diversos trabalhos, tendo a análise do TDD como foco, foram produzidos e, em alguns deles, foi verificada a carência de uma atenção maior à maneira que é empregado. Foram estudadas possibilidades de haver falhas e até mesmo a necessidade de práticas adicionais a serem observadas durante sua aplicação que poderiam melhorar a qualidade do *software*.

Este trabalho apresenta inicialmente uma análise dessas práticas que em seguida foram agrupadas e avaliadas na forma de questionário submentido a profissionais que aplicam o TDD em seus projetos. A pesquisa teve como objetivo verificar o grau de relevância dessas práticas.

ABSTRACT

The Test-Driven Development is a *software* development technique popularized with the advent of agile methods whose aim deliver clean code that works.

Using TDD seems to be simple, however it requires great discipline and adherence to the steps proposed by it.

Several studies, with analysis focused on TDD, were produced and, in some of them, there was a lack of greater attention to the way it is applied. Possibilities of failures and even the need for additional practices to be observed during the implementation in order to improve the quality of *software* had been presented by these studies.

This paper presents an initial analysis of these practices posteriorly gathered and submitted in the form of a questionnaire to professionals who use TDD in their project. The research aimed to determine the relative importance of these practices.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

O presente trabalho faz uma análise da aplicação da técnica de *Test-Driven Development* (TDD) apresentando uma sugestão de boas práticas coletadas de diferentes fontes, que visam melhorar a qualidade de *software*.

Através deste capítulo, são apresentadas as motivações, o objetivo, as justificativas ao tema e a estrutura do trabalho.

1.1 MOTIVAÇÕES

A Engenharia de *Software* é uma disciplina que, de acordo com Pressman (2009), visa regulamentar as atividades que possibilitam o controle do processo de desenvolvimento e oferece uma base para a construção de *software* de alta qualidade.

Para atender às necessidades impostas pelo mercado atual, *softwares* cada vez mais complexos precisam ser produzidos dentro de prazos reduzidos e custos controlados, se tornando fundamental a melhoria dos processos da engenharia de *software* pela exigência de mais organização e dinamismo para suprir tal demanda.

Por este motivo, processos tradicionais de desenvolvimento de *software* passam a ser considerados dispendiosos e mesmo inadequados diante da necessidade de uma reação com agilidade. Nesse cenário, surgiram os métodos ágeis, que enfatizam princípios associados à comunicação, objetividade, maior foco no desenvolvimento e interação com o cliente, de forma a proporcionar agilidade e eficiência no desenvolvimento (SANTOS, 2011).

Vale reforçar que a exigência por produtos com alta complexidade, amplo escopo e concluídos com rapidez, tem influência sobre a qualidade por tornar difícil a prevenção ou eliminação de defeitos e seus impactos negativos relacionados, portanto, além da garantia da entrega do produto no prazo, vê-se como fundamental que haja qualidade.

Uma constatação recorrente em relação a defeitos encontrados em um *software* segundo Black et al. (2006), é que o custo de encontrar e corrigir defeitos,

aumenta consideravelmente através do ciclo de vida de desenvolvimento, em outras palavras, se encontrados em fases ainda primárias, menor será seu custo financeiro e menos esforço será despedido para corrigi-los, além disso, possíveis impactos causados por defeitos identificados tardiamente podem elevar ainda mais esses custos.

Essa atual realidade do mercado faz com que uma das principais discussões na área de Engenharia de *Software* seja a busca de atividades de garantia da qualidade a serem realizadas para prevenir e/ou eliminar certas classes de defeitos, ou ainda, para reduzir a probabilidade ou severidade de tais impactos quando não se pode evitar (RIBEIRO, 2010).

Segundo Beck (2003), uma das técnicas que tem como objetivo obter um código limpo que funcione com qualidade é o *Test-Driven Development* (TDD) que foi popularizada e incorporada ao *eXtreme Programming* (XP), uma prática de desenvolvimento ágil.

Através do TDD, o desenvolvimento é orientado a testes automatizados, ou seja, um novo código é escrito apenas se um teste automatizado destinado a ele já existir. Assim, é possível reduzir a densidade de defeitos (BECK, 2003).

Muitos trabalhos questionam a eficiência do TDD e o quanto é mais efetivo em relação à adoção apenas de técnicas tradicionais de testes, as chamadas *test-last*, onde o esforço de teste é concentrado após a criação do código (VU, et al., 2009; JANZEN; SAIEDIAN, 2008).

Esses trabalhos, muitas vezes, têm como base para suas conclusões, experimentos controlados, executados em ambiente acadêmico e/ou corporativo. Observando-se a maneira como esses experimentos foram conduzidos, novos questionamentos surgiram direcionados à maneira como TDD está sendo aplicado nestes experimentos.

A partir dessa observação, é possível constatar que alguns erros são recorrentes e deveriam ser considerados durante um projeto que faz uso do TDD.

1.2 Objetivo

Beck (2003) ressalta que apesar de as regras aplicadas no TDD serem simples, é necessário observar possíveis implicações técnicas, enumerando algumas delas:

- Quem desenvolve o código, deve desenvolver o teste, pois não se pode esperar de outra pessoa a execução desta tarefa diversas vezes durante a codificação;
- O ambiente de desenvolvimento deve prover resposta rápida a pequenas mudanças;
- Os códigos devem consistir de componentes altamente coesivos e fracamente acoplados para que se possa desenvolver testes simples.

Estas regras podem ser ampliadas através de observações adicionais presentes em recomendações da literatura pesquisada que abordaram a aplicação do TDD.

Assim, o objetivo deste trabalho é apresentar um conjunto de boas práticas a serem consideradas durante a adoção do TDD para minimizar erros durante sua aplicação.

1.3 Justificativa

Apesar de o conceito da técnica de *Test-Driven Development* (TDD) estar bem definido por Beck (2003), trata-se de uma prática relativamente recente que gera uma série de questionamentos em relação a sua real eficiência, em consequência, há muitas pesquisas sobre os resultados do TDD, como o desenvolvido por Kollanus (2010).

Estudos mostram que de fato o TDD é uma técnica que, quando usada durante o desenvolvimento de um *software*, resulta em código de melhor qualidade com menor número de defeitos, porém, a maneira como é aplicada pode comprometer seus objetivos propostos.

É possível encontrar relatos e discussões sobre possíveis causas para os resultados abaixo das expectativas em relação ao que teoricamente é oferecido pelo TDD, (ANICHE; GEROSA, 2010), ou até que mesmo fatores que inibam a adoção da técnica (CAUSEVIC; SUNDMARK, PUNNEKKAT, 2011). Sugestões para explorar de maneira correta e obter o maior retorno desta técnica são citadas em diferentes textos.

Porém, não foi encontrado até o momento, um trabalho que reunisse essas sugestões e críticas à maneira que o TDD vem sendo utilizado.

Faz-se necessário uma análise da aplicação do TDD para detectar essas falhas e levantar possíveis práticas a serem utilizadas junto aos seus princípios básicos para que este seja o mais efetivo possível.

1.4 Estrutura do Trabalho

Capítulo 1 – INTRODUÇÃO – Este capítulo apresenta motivações, objetivo, justificativa e estrutura do trabalho.

Capítulo 2 – MÉTODOS ÁGEIS – Este capítulo apresenta de forma sucinta, os conceitos de métodos ágeis de desenvolvimento de *software*.

Capítulo 3 – QUALIDADE DE *SOFTWARE* EM MÉTODOS ÁGEIS – Este capítulo apresenta definições de qualidade de *software* mostrando como é tratada em métodos ágeis, abordando também conceitos básicos sobre testes de *software.*

Capítulo 4 – *TEST-DRIVEN DEVELOPMENT* – Este capítulo apresenta a definição desta técnica de desenvolvimento.

Capítulo 5 – ANÁLISE DO TDD – Este capítulo traz uma síntese baseada em referências bibliográficas que analisaram a aplicação do TDD, sua eficiência e real contribuição para a qualidade de *software* resultando em uma lista de boas práticas extraídas de sugestões listadas nestes trabalhos validadas através de pesquisa em forma de questionário.

Capítulo 6 – CONSIDERAÇÕES FINAIS – Este capítulo apresenta uma análise final das boas práticas apresentadas no capítulo anterior.

REFERÊNCIAS – Apresenta a relação de livros, artigos acadêmicos e outras produções literárias que serviram de suporte para o desenvolvimento deste trabalho.

APÊNDICE 1 – Apresenta o questionário divulgado através de fóruns voltados para TDD com o objetivo de coletar opiniões dos profissionais da área.

APÊNDICE 2 – Planilha que apresenta as respostas obtidas pela pesquisa.

2. MÉTODOS ÁGEIS

No final da década de 90, diversas métodos começaram a chamar cada vez mais atenção do público (na comunidade técnica). Cada qual com sua combinação diferente de velhas ideias, novas ideias, e velhas ideias adaptadas, todas enfatizaram uma estreita colaboração entre a equipe de programadores e especialistas em negócios; a comunicação face a face (como mais eficiente do que a documentação escrita); entregas frequentes agregando valor de negócio; equipes unidas e auto-organizadas, e meios de desenvolver código e equipes de tal forma que as mudanças inevitáveis requisitos, não sejam uma crise (Agile Alliance, 2011).

Neste capítulo, são apresentados de forma ampla, a origem dos métodos ágeis, sua definição, valores e princípios que guiam o desenvolvimento que o segue como modelo, assim como os principais métodos ágeis presentes no contexto de desenvolvimento de *software*.

2.1 A Origem dos Métodos Ágeis

Na economia de hoje, as necessidades de negócio mudam o valor dos requisitos de *software* muito rapidamente. O que está definido como bom conjunto de requisitos num dado momento, pode não ter o mesmo valor em um curto espaço de tempo. Mesmo sendo os requisitos acordados com os clientes, o dinamismo do mundo dos negócios não vai desacelerar para que a realidade na qual foram definidos continue constante (FOWLER, 2005).

Métodos de desenvolvimento tradicionais se mostram deficientes para esse cenário, pois partem do princípio de que os requisitos do produto a ser desenvolvido serão definidos e acordados no início do projeto, não havendo mudanças significativas durante a evolução da construção do *software*.

Pode-se ainda listar como pontos negativos desses métodos, a burocracia imposta pela alta carga de documentação exigida pela formalidade que os caracterizam, uma vez que produzir e manter esses artefatos demanda tempo e

esforço dos envolvidos e também a definição de papéis específicos para cada indivíduo, impedindo uma participação mais ampla no projeto, uma vez que suas atividades são restritas às responsabilidades a ele designadas.

Com o intuito de suprir as deficiências apresentadas pelos métodos convencionais, novos métodos contrários aos padrões até então utilizados na indústria, vinham sendo aplicadas por diferentes desenvolvedores, entre eles Kent Beck.

Durante um *workshop* em Snowbird, Utah, EUA, em fevereiro de 2001, esses desenvolvedores apresentaram e analisaram a similaridade entre seus métodos e manifestaram razoável interesse em alguma forma de trabalho coletivo. O resultado é o "Manifesto para Desenvolvimento Ágil de *Software*", uma declaração de valores comuns e princípios de processos ágeis, assim como a "Aliança Ágil", uma organização sem fins lucrativos que procura promover o conhecimento e discussão sobre todos os métodos ágeis (FOWLER, 2005).

Portanto, pode-se concluir que os métodos ágeis surgiram visando suprir as fraquezas percebidas e reais da engenharia de *software* convencional (PRESSMAN, 2009).

2.2 Definição de Métodos Ágeis de Desenvolvimento

"Agilidade é a capacidade tanto de criar quanto de responder a mudanças a fim de se beneficiar em um ambiente de negócios turbulento"
Jim Highsmith apud Kelly(2008).

Métodos tradicionais adotam processos definidos de desenvolvimento divididos em fases onde cada uma possui uma entrada definida que por sua vez, gera uma saída definida. Esse tipo de estratégia segue um caminho linear para definir requisitos, projeto, desenvolvimento e os testes de *software* (COHEN, 2010).

Esse tipo de desenvolvimento assume que o mundo é ordenado e uma vez definidos os requisitos, nada será alterado, portanto, dificilmente assumem a possibilidade de mudança de direção (VILET, 2008).

Desenvolvimento ágil é um modo alternativo de construir produtos oferecendo uma resposta mais rápida às necessidades do mercado através do acolhimento de alterações, que é o principal guia para a agilidade, logo, o processo ágil gerencia imprevisibilidade e é adaptável a mudanças (PRESSMAN, 2009).

Isso se explica pelo fato de que entregar os requisitos definidos do cliente não seja o único objetivo de desenvolvimentos ágeis (os métodos tradicionais já fazem isso), mas também se trata de atender às mudanças desses requisitos até que o produto seja implantando (KELLY, 2008).

Métodos ágeis assumem que o mundo é caótico e que mudanças são inevitáveis e ao mesmo tempo, focam em entregar valor ao cliente o mais rápido possível (VILET, 2008).

Para entregar um produto de qualidade, é necessário que haja um controle sobre esse ambiente onde a qualquer momento, pode haver uma mudança inesperada em relação ao que foi solicitado, um mecanismo de *feedback* pode dizer com precisão qual a situação atual em intervalos frequentes e a chave para esse *feedback* é o desenvolvimento iterativo (FOWLER, 2005).

O desenvolvimento iterativo não é uma novidade, porém possui um diferencial quando aplicado através de métodos ágeis, as iterações são menores e mais frequentes, entregando funções reais e, além disso, cada entrega deve agregar valor ao negócio. Em abordagens tradicionais, o valor de negócio só era visualizado no final do projeto quando os componentes do *software* eram integrados (COEHN, 2010).

Abaixo, seguem os valores-chave propostos no Manifesto Ágil que sumarizam o que foi exposto até o momento (AGILE ALLIANCE, 2011):

• **Indivíduos e iterações** são mais importantes do que processos e ferramentas. Métodos ágeis são orientados a pessoas e não a processo, enfatiza o elemento humano e a sinergia da equipe é considerada muito importante.

Os ciclos de desenvolvimento são pequenos e incrementais não sendo extensivamente planejados, mas revisados a cada início de novo ciclo de onde se

verifica o que deve ser melhorado ou mudado (VILET, 2008).

• ***Software* funcionando** é mais importante do que documentação completa. Não se gasta muita energia com documentação, pois esta fica desatualizada, o conhecimento das pessoas envolvidas é a fonte de informação. Não há uma extensiva arquitetura ou projeto em fases muito adiante uma vez que pode ser perda de tempo, pois pode haver mudanças significativas. Esse tipo de esforço é aplicado apenas para fases imediatamente seguintes à atual (VILET, 2008).

• **Colaboração com o cliente** é mais importante do que negociação de contratos. O cliente passa a ter um controle muito mais refinado sobre desenvolvimento do produto que solicitou, podendo em cada iteração, alterar a direção do desenvolvimento conforme novas necessidades são identificadas no início o que leva a um relacionamento muito mais próximo e a uma verdadeira parceria de negócios (FOWLER, 2005).

• **Adaptação a mudanças** é mais importante do que seguir o plano inicial.

Além dos valores, foram também definidos doze princípios pela Aliança Ágil que auxiliam a alcançar a agilidade proposta por este conceito, são estes listados na tabela 1:

TABELA 01 – Princípios definidos no Manifesto Ágil - (AGILE ALLIANCE, 2011).

1.	Nossa maior prioridade é satisfazer ao cliente desde o início por meio de entrega contínua de *software* valioso.
2.	Modificações de requisitos são bem-vindas, mesmo que tardias no desenvolvimento. Os processos ágeis aproveitam as modificações como vantagens para competitividade do cliente.
3.	Entrega de softwares funcionando frequentemente, a cada duas semanas até dois meses, de preferência no menor espaço de tempo.
4.	O pessoal de negócio e os desenvolvedores devem trabalhar juntos diariamente durante todo o projeto.
5.	Construção de projetos em torno de indivíduos motivados. Forneça-lhes o ambiente e apoio que precisam e confie que eles farão o trabalho.

6.	O método mais eficiente e efetivo de levar informação para dentro de uma equipe de desenvolvimento é a comunicação face a face.
7.	*Software* funcionando é a principal medida de progresso.
8.	Processos ágeis promovem desenvolvimento sustentável. Os *stakeholders*, desenvolvedores e usuários devem ser capazes de manter um ritmo constante, indefinidamente.
9.	Atenção contínua a excelência técnica e ao bom projeto facilitam a agilidade.
10.	Simplicidade – a arte de maximizar a quantidade de trabalho não efetuado – é essencial.
11.	As melhores arquiteturas, requisitos e projetos surgem de equipes auto-organizadas.
12.	Em intervalos regulares, a equipe reflete sobre como se tornar mais efetiva, então sintoniza e ajusta adequadamente seu comportamento.

Seguindo um desenvolvimento com métodos ágeis, cada final de fase apresenta um produto executável, isso faz com que em cada uma dessas fases, os usuários adquiram mais conhecimento sobre o produto e quem está trabalhando no desenvolvimento aprenda novas lições e, por este motivo, mudanças serão necessárias e com elas, o ajuste das práticas para acomodá-las. Seguir o plano não é o objetivo, mas sim desenvolver um *software* que resolva os problemas do cliente (COEHN, 2010).

2.3 Métodos Ágeis

São apresentados a seguir, de forma sucinta, alguns dos modelos de processo ágeis mais conhecidos na comunidade técnica, dando-se maior atenção ao *eXtreming Programming* por ter difundido a técnica de *Test-Driven Development.*

2.3.1 Família *Crystal*

Trata-se de um conjunto de métodos criado por Alistair Cockburn, onde é assumido que todo projeto tem necessidades diferentes por apresentarem

características e criticidades distintas. Estes métodos compartilham características comuns, mas com diferentes graus de sofisticação e formalismo para serem usadas conforme a complexidade de cada projeto (FILHO, 2008).

A inspiração de Cockbun para nomear esses métodos, veio das propriedades geológicas dos cristais que sob a influência da luz, muda de cor conforme a intensidade da mesma foi, portanto, adotada uma cor para cada uma delas, baseado no número de envolvidos que o projeto requer. Assim criaram-se *Crystal Clear, Crystal Yellow, Crystal Orange, Crystal Orange Web, Crystal Red, Crystal Magenta* e *Crystal Blue*. Vale lembrar que cada uma possui graus de gerenciamento e de comunicação adequados conforme número de envolvidos (SANTOS, 2011).

A figura 01 mostra três dos fatores que influenciam a escolha do método: a carga de comunicações (tamanho de equipe), a criticidade do sistema e prioridades do projeto. No eixo horizontal, observa-se à comunicação face a face entre os membros da equipe que é reduzida à medida que se afasta do eixo, sendo base para a definição do tamanho da equipe. No eixo vertical, observam-se os riscos, ou potencial do sistema em causar danos como perda de conforto, perdas financeiras ou perda no ciclo de vida do projeto que através de sua análise, é possível definir a prioridade e criticidade do projeto (SANTOS, 2011).

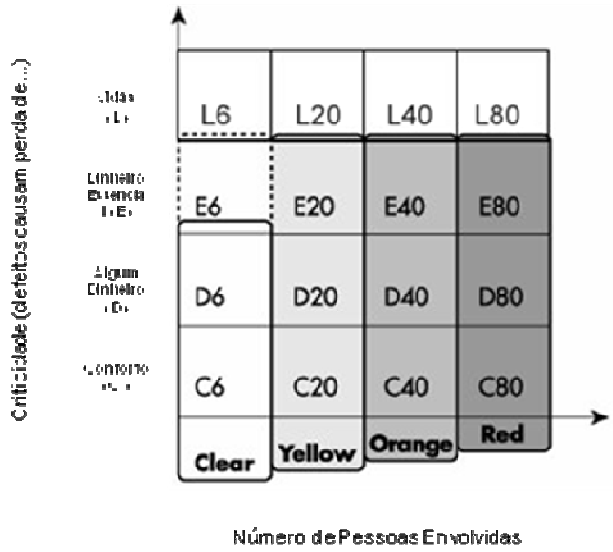

FIGURA 01 - Ciclo de vida do método da família *Crystal* adaptado de SANTOS (2011).

Os métodos *Crystal* apoiam-se em princípios que estão alinhados com o Manifesto Ágil valorizando o desenvolvimento incremental, entregas frequentes, envolvimento de usuários e clientes, testes automatizados, adaptação a mudanças e aprendizado contínuo. O que é possível devido às suas propriedades a serem observadas durante sua adoção que são entregas frequentes, comunicação intensa, melhora por reflexão da equipe, segurança pessoal, foco, acesso fácil a usuários experientes, integração contínua com testes (FILHO, 2008).

2.3.2 FDD (*Feature Driven Development*).

Peter Coad concebeu o FDD como modelo prático de processo para engenharia de *software* orientada a objetos e posteriormente foi aprimorada para que atendesse às necessidades de processos ágeis por Stephen Palmer e John Felsing. Em um ambiente de desenvolvimento com FDD, uma característica é uma função valorizada pelo cliente a ser implementada em até duas semanas (PRESSMAN, 2009).

Sendo assim, as fases do FDD apresentadas na figura 2, se organizam na implementação de pequenas partes funcionais correspondentes às características descritas pelos clientes, que podem ser entregues e inspecionadas com maior facilidade (SANTOS, 2011).

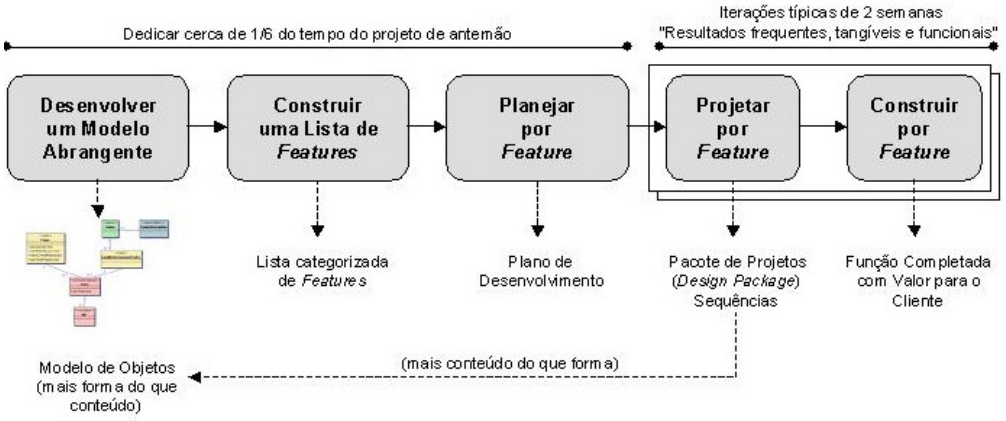

FIGURA 02 - Ciclo de vida do FDD – Adaptado de Pressman (2009).

Trata-se de um modelo que coloca mais ênfase em diretrizes e técnicas de gerenciamento de projetos comparado a outros métodos ágeis, é importante que os

envolvidos entendam o estado do projeto, quais avanços foram feitos e que problemas foram encontrados. Para conseguir definir a prioridade conforme a situação do projeto, o FDD possui seis marcos de referência a serem observados durante o projeto e implementação de uma característica: travessia do projeto, inspeção do projeto, projeto, código, inspeção do código, promoção para construção (PRESSMAN, 2009).

2.3.3 SCRUM

A origem do SCRUM se deu em 1986 através de um artigo publicado por Hirotaka Takeuchi e Ikujiro Nonaka na revista Harvard Business Review, intitulado "The New Product Development Game", nele descreveram uma abordagem holística onde as equipes multidisciplinares atuam no projeto, trabalhando em função de um objetivo comum, que é entregar um produto com qualidade. O termo SCRUM, é uma alusão a uma estratégia em jogos de rúgbi usada para o reinício do jogo (PHAN, PHAN, 2011).

Trata-se de um método de gerenciamento de projetos, os requisitos são mantidos em uma lista chamada *Product Backlog* que abrange todas as solicitações do cliente como nova funcionalidade, melhorias de uma funcionalidade existente, correção de defeitos, e assim por diante. A ordenação dessa lista baseia-se na priorização dos itens de acordo com o valor agregado ao negócio. Sendo assim, o *Product Backlog* representa todo o trabalho a ser executado (COHEN, 2010).

De acordo com Santos (2011), o funcionamento do SCRUM é estruturado em ciclos chamados *sprints*, que são iterações de trabalho que duram de duas a quatro semanas. O projeto como um todo pode ser composto de vários *sprints*.

Para cada *sprint*, itens dos *Product Backlog* são selecionados em ordem de prioridade para os *sprint backlog*. Essa ordem pode ser sempre reorganizada de acordo com novas necessidades de priorização que surgirem, porém, uma vez que um item está em desenvolvimento, ele não pertence mais ao *backlog*, portanto, não pode ser alterado. Caso ainda haja a necessidade de mudança, um novo item é adicionado ao *backlog* a ser tratado em um próximo *sprint* (COHEN, 2010).

Segundo Phan, Phan (2011), uma equipe de SCRUM é composta por um *ScrumMaster*, um *product owner* e a equipe de desenvolvimento, com todos os conhecimentos e habilidades necessários para construir o produto. Esta equipe deve ser alto-organizada, motivada a ter senso de propriedade e orgulho de seus resultados, tudo gira em torno da colaboração entre os envolvidos.

Na Tabela 02 a seguir, Phan, Phan (2011) listam as principais atividades de cada papel que atua em uma equipe de SCRUM.

Tabela 02 – Responsabilidades colaborativas dos membros da equipe de SCRUM – Adaptado de (PHAN, PHAN. 2011)

Product owner	Trabalha com os *Stakeholders* para definir o *Product Backlog* e é o responsável pelo resultado de negócio.
	Reúne os requisitos para o *Product Backlog*
	Colabora com o ScrumMaster para proteger a equipe de distúrbios externos
	Orienta o time para atingir a entrega e o objetivo planejado do *sprint*
	Mantêm-se a par do progresso do projeto
	Se faz disponível para fornecer *feedback* para a equipe
ScrumMaster	Atua como guardião do processo Scrum
	Orienta o time para atingir a entrega e o objetivo planejado do *sprint*
	Colabora com o *product owner* para proteger a equipe de distúrbios externos
	Auxilia a manter o controle do progresso do projeto conforme necessário para ajudar a equipe
	Organiza a retrospectiva do *sprint* para auxiliar o aprendizado adquirido a melhorar o processo e desempenho do projeto.

Equipe de desenvolvimento	Alto gerenciamento e alto organização
	Responsável por estimar os itens do *backlog* ou estórias de usuários por si próprio
	Autorizado a transformar um item do *backlog* ou estória de usuário em tarefa a ser trabalhada
	Manter o controle do progresso do projeto
	Responsável por exibir os resultados do *sprint* para o *Product Owner* e *Stakeholders* ao final de cada *sprint*.

De acordo com Santos (2011), durante cada *sprint* são realizadas reuniões diárias onde a equipe compartilha uns com os outros o andamento do projeto. Ao final do *sprint*, seu respectivo incremento funcional é entregue ao cliente que junto ao *Product Owner*, o analisa e caso necessário redefine os requisitos, a priori, de acordo com o interesse do cliente. Essa descrição é ilustrada através da Figura 3.

FIGURA 03 - Visão geral do método SCRUM – (SANTOS, 2011).

2.3.4 *EXtreme Programming* (XP).

Método criado por Kent Beck baseado em anos de experiência. Formalizado em um livro em 1999 *"EXtreme Programming Explained: Embrace Change"* após o mesmo ter atuado em um projeto para a empresa Chrysler dos EUA, onde selecionou as práticas que haviam se mostrado mais eficientes e as encarou como

botões de volumes, os aumentando ao seu valor máximo (FILHO, 2008; SATO, 2007).

De acordo com Sato (2007), o XP tem como objetivo a excelência no desenvolvimento de *software*, visando baixo custo, poucos defeitos, alta produtividade e alto retorno de investimento.

Após a primeira edição do livro de Beck, o método se tornou internacionalmente conhecido e respeitado, porém até então indicado para pequenas equipes de no máximo 12 pessoas. Uma segunda edição de seu livro lançada em 2004 com o auxílio de sua esposa psicóloga Cynthia Andres, apresentou o XP mais inclusivo, abrangente e flexível através do enfoque nos valores do método (FILHO, 2008).

Segundo Beck e Andres (2004), o XP pode ser definido da seguinte forma:

- XP é leve: faz-se apenas o que é preciso fazer para criar valor para o cliente
- XP é um método baseado em restrições no desenvolvimento de *software*: não aborda gestão de projetos, justificativa financeira de projetos, operações, *marketing* ou vendas. Apesar de ter implicações em todas essas áreas, não aborda diretamente essas práticas
- XP pode trabalhar com equipes de qualquer tamanho: Os valores e princípios que compõem o XP são aplicáveis em qualquer escala. As práticas precisam ser adaptadas quando muitas pessoas estão envolvidas.
- XP se adapta a requisitos mal definidos ou que mudam frequentemente.

2.3.4.1 Princípios

Os princípios do XP funcionam como ferramentas de tradução dos valores em práticas (SATO, 2007).

Existem 14 princípios do XP:

Humanidade: Um dos grandes desafios de uma equipe de desenvolvimento

de *software* é equilibrar as necessidades individuais com as necessidades da equipe, o XP tenta atender a ambas as necessidades (BECK; ANDRES, 2004).

Economia: A equipe que não tem conhecimento dos riscos econômicos não tem como evitar que o projeto seja apenas um sucesso no aspecto técnico (SATO, 2007).

Baseado nesse princípio, a equipe elege as prioridades para agregar o máximo de valor no menor intervalo de tempo, flexibilidade para reagir rapidamente a mudanças também é importante caso seja necessário reeleger prioridades (FILHO, 2008).

Benefício Mútuo: Toda atividade deve beneficiar todos os envolvidos. Este é o mais importante, porém é mais complicado princípio a se seguir. Sempre existirão soluções que beneficiarão uma pessoa, mas que podem prejudicar outra (BECK; ANDRES, 2004).

Auto-semelhança: Filho (2008), definiu como boas soluções que devem poder ser aplicadas novamente, inclusive em outros contextos e escalas. A idéia é identificar estruturas equivalentes a padrões de projeto para o processo de desenvolvimento.

Melhoria: Este princípio mostra que uma atividade iniciada agora pode ter seus resultados refinados ao longo do tempo.

O XP busca excelência em desenvolvimento de *software* através da melhoria constante, é importante fazer o melhor trabalho possível no momento, identificar o que pode ser feito melhor amanhã, ao invés de buscar a perfeição para se dar início a um trabalho (BECK; ANDRES, 2004).

Diversidade: Equipes precisam reunir uma variedade de habilidades, atitudes e perspectivas de visualizar problemas e ameaças, serem capazes de pensar em diferentes maneiras de resolver problemas e implementar soluções (BECK; ANDRES, 2004).

Dessa diversidade podem até surgir conflitos, mas estes devem ser vistos

como oportunidade para discussões e obtenção de melhores resultados (SATO, 2007).

Filho (2008), recorda que junto com a diversidade é importante que a equipe possua o valor do respeito, pois ele garante que as diferenças colaborem para o crescimento de todos.

Reflexão: Segundo Beck e Andres (2004), boas equipes não fazem somente seu trabalho, também pensam em como e por que estão trabalhando. Analisam porque tiveram sucesso ou falharam. A reflexão ocorre após uma ação, analisando seus resultados. O que é tirado dessa análise é o aprendizado que maximiza o *feedback* e a melhoria contínua.

Fluxo: Sugere a entrega de um fluxo contínuo de *software* que agrega valor envolvendo-se em todas as atividades de desenvolvimento ao mesmo tempo. O XP propõe práticas que tendem ao um fluxo contínuo ao invés de fases distintas (BECK; ANDRES, 2004).

O tempo gasto para se identificar uma tarefa que foi executada com sucesso ou não, é proporcional ao tamanho da atividade e, consequentemente, ao risco de erro, o XP propõem que as equipes entreguem incrementos pequenos de funcionalidade frequentemente, minimizando assim o risco e o tamanho de possíveis defeitos. (SATO, 2007).

Oportunidade: Beck e Anders (2004), sugerem a ver problemas como oportunidade para mudança.

Redundância: Sugere que os problemas complexos e críticos devem ser solucionados de diferentes maneiras. Programação por pares, integração contínua e desenvolvimento dirigido a testes são práticas redundantes que procuram reduzir a quantidade de defeitos e aumentar a qualidade do *software* produzido (SATO, 2007).

Falha: Segundo Beck e Andres (2004), quando não se sabe o que fazer, a falha pode ser o caminho mais seguro para o sucesso. Da falha se tira o aprendizado que muitas vezes é difícil de adquirir.

Qualidade: Sacrificar a qualidade não é um meio de controle. Projetos não evoluem mais rápido adotando um nível menor de qualidade. Em um projeto, normalmente tempo e escopo são definidos previamente, sendo o escopo o fator de controle de evolução, não há como realizar esse controle através da qualidade. Cada incremento em qualidade resulta em melhorias em outras propriedades do projeto como produtividade e eficiência (BECK; ANDRES, 2004).

Investir em qualidade também traz mais motivação para a equipe, mais satisfação para o cliente e maior velocidade para novas implementações (FILHO, 2008).

Pequenos Passos (*Baby Steps*).: Através deste princípio, a evolução do projeto se dá em pequenos incrementos, sendo possível identificar se a equipe está na direção correta. Não significam resultados extremamente demorados a serem medidos, mas diversas pequenas mudanças ocorrendo em direção ao objetivo, além disso, o tempo gasto em pequenos passos é menor do que o tempo perdido em uma grande mudança equivocada. Práticas como *test-first programming* e integração contínua, usam esse princípio (BECK; ANDRES, 2004).

Aceitação da Responsabilidade: Responsabilidade não deve ser imposta, ela deve ser apenas aceita (BACK, ANDRES, 2004).

As práticas do XP sugerem esse princípio fazendo com que cada membro da equipe seja responsável pelas tarefas que assume. Por exemplo, sugerir que uma pessoa seja responsável por uma determinada estória, a faz também responsável por sua estimativa, projeto, implementação e controle de evolução (SATO, 2007).

2.3.4.2 Valores

Filho (2008), menciona que inicialmente o XP foi baseado em quatro valores que, em alinhamento com o Manifesto Ágil, definiram as prioridades do método. Posteriormente um quinto valor foi adicionado.

Os cinco valores que sustentam o XP são:

Comunicação: É o que mais importa em uma equipe de desenvolvimento de

software, muitos dos problemas que surgem durante o desenvolvimento na maioria das vezes já são conhecidos, mas este conhecimento não chega até alguém com o poder de fazer uma mudança efetiva (BECK; ANDRES, 2004).

No XP, a comunicação é favorecida entre todos os membros da equipe através de atividades colaborativas. Os princípios de humanidade e de diversidade possuem a preocupação em estimular a comunicação principalmente entre os desenvolvedores, já os princípios de economia e do benefício mútuo promovem a colaboração entre participantes com perfil técnico e com o perfil de negócios, garantindo que os desenvolvedores compreendam os pontos de vista dos usuários e assim produzam soluções adequadas ás suas expectativas e necessidades (FILHO, 2008).

Simplicidade: O valor intelectual mais importante do XP segundo Beck e Andres, (2004), afirmam que encontrar a solução mais simples não é uma tarefa fácil.

Uma equipe de XP, em ambientes ágeis, onde a adaptação a mudanças é bem vista e encorajada, busca soluções mais simples, evitando desperdício com soluções genéricas para problemas futuros (SATO, 2007).

Filho (2008) diz que os princípios do benefício mútuo, auto-semelhança, melhoria e passos pequenos promovem a simplicidade durante o desenvolvimento, Sato (2007) complementa que técnicas como Desenvolvimento Dirigido por Testes sustentam esse valor.

Feedback: Nenhuma decisão prévia permanece válida por muito tempo se tratando de detalhes de desenvolvimento de *software*, requisitos ou a arquitetura do sistema. A mudança é inevitável e cria a necessidade do *feedback* (BECK; ANDRES, 2004).

Filho (2008), afirma um *feedback* constante entre todos os envolvidos permite que a equipe e o projeto identifiquem seus problemas e se adaptem a eles. E acrescenta que os princípios da humanidade, melhoria, reflexão, fluxo, oportunidade e passos pequenos (*baby steps*) garantem *feedback* e melhorias constantes.

Para Sato (2007), os valores do XP se complementam, por isso, o *feedback* é parte importante da comunicação e da simplicidade, adiciona que a redução dos ciclos de *feedback* é a maior contribuição para o sucesso dos métodos ágeis e que práticas como programação por pares e *Test-Driven Developement*, que será apresentado no capítulo 4, são práticas que evidenciam tal valor.

Coragem: É uma ação efetiva diante de qualquer ameaça que causa medo. A maneira que o medo é tratado por alguém em uma equipe determina se ele está trabalhando de maneira eficaz. Ter coragem, não é fazer algo sem levar em conta as consequências, mas associar a ela outros valores para se orientar em um momento de medo. (BECK; ANDRES, 2004).

Filho (2008) afirma que para motivar o surgimento de coragem junto com responsabilidade, os princípios da humanidade e da aceitação da responsabilidade valorizam as pessoas e aumentam a sua autoconfiança e os princípios do benefício mútuo, melhoria e oportunidade sugerem que a equipe se sinta segura e confiante para fazer as mudanças necessárias.

Respeito: Segundo Beck e Andres (2004), os quatro valores anteriores apontam para o quinto que os sustenta, o respeito. Se os membros da equipe não se preocuparem com os outros e com o que eles estão fazendo, o XP não vai funcionar.

A falta de respeito pode prejudicar a adoção do XP, comunicação sem respeito pode causar conflitos internos, coragem sem respeito pode trazer atitudes que influenciem no bem estar da equipe (SATO, 2007).

2.3.4.3 Práticas

São vinte e quatro as práticas do XP, divididas em *práticas primárias*, aquelas que podem ser aplicadas separadamente trazendo melhoria para equipe e *práticas corolárias*, que são mais complexas de serem aplicadas e se mostram mais eficazes se adotadas após domínio e experiência prévia com as práticas primárias. Kent Beck sugere que a utilização de cada um delas deve ser adaptada pela equipe da maneira que achar mais apropriada (SATO, 2007).

Práticas Primárias

Sentar Junto: É desejável que haja um espaço de trabalho comum para toda a equipe no lugar de cubículos individuais, com alguns espaços que proporcionem alguma privacidade quando necessário (BECK; ANDRES, 2004).

A proximidade entre as pessoas contribui para identificar e valorizar as habilidades individuais, intensificando a comunicação o que favorece a colaboração, a ajuda mútua e a disseminação do conhecimento (FILHO, 2008).

Equipe Completa: Essa prática propõe que a equipe no XP tenha senso de totalidade, para isso precisa ser composta de pessoas que possuam todas as habilidades e perspectivas necessárias para o projeto ser bem sucedido. Os indivíduos precisam ter o senso de equipe (BECK; ANDRES, 2004).

Ambiente de Trabalho Interativo: O espaço de trabalho deve ser utilizado para refletir o andamento do projeto, um observador deve ser capaz de ter uma idéia geral de como está o projeto em 15 segundos, ter informações sobre possíveis ou reais problemas se analisar mais atentamente (BECK; ANDRES, 2004).

Segundo Sato (2007), um exemplo de instrumento para difundir informação são os radiadores de informação de Cockburn, que funcionam como cartões com estórias em um mural, quadros brancos, notas em papel nas paredes e gráficos como o *burn-down chart* proposto pelo SCRUM para verificar a velocidade da equipe.

Trabalho Energizado: Trabalhar somente enquanto for produtivo, é fácil retirar valor de um projeto de *software* e quanto se está cansado, é difícil detectar que isto está acontecendo. Desenvolver *software* é um jogo de concentração e esta vem de uma mente preparada, relaxada e descansada (BACK, 2004).

Durante o planejamento o número de horas a serem dedicadas ao projeto deve ser definido de forma realista, hora-extra deve ser exceção e não regra (SATO, 2007).

Programação em Pares: O trabalho em pares durante as tarefas de

implementação promove o trabalho coletivo e colaborativo. Deve-se trocar as duplas frequentemente para que todos os participantes tenham a possibilidade de interagir, aumentando a comunicação e a união do grupo. Com duas pessoas concentradas na mesma tarefa, muitos erros são evitados e quando existem, eles são encontrados mais rapidamente. Enquanto um dos programadores digita, o outro revisa o código e sugere melhorias. O trabalho em duplas também promove a replicação de conhecimentos específicos de cada participante em vários membros da equipe (FILHO, 2008).

Estórias: É uma unidade de funcionalidade escrita por um cliente em um cartão que geralmente representa um requisito funcional. Um dos formatos propostos é:

"Como um <usuário / papel>

Eu gostaria de <funcionalidade>

Para que <valor do negócio>"

O planejamento do XP é baseado no conjunto dessas estórias que devem ser analisadas e receber estimativas dos desenvolvedores e priorizadas pelo cliente (SATO, 2007).

Ciclo Semanal: O desenvolvimento acontece iterativamente em ciclos curtos onde a equipe realiza atividades de planejamento detalhado apenas de curto prazo, com estimativas e priorizações focados nas estórias que serão implementadas durante a iteração em planejamento. A cada ciclo a equipe pode analisar seu desempenho na iteração anterior para fazer um planejamento adequado à sua velocidade (FILHO, 2008).

Beck e Andres, (2004), propõem uma reunião semanal para que a iteração seja planeja, nessa reunião, deve observar os pontos abaixo:

- Reveja o progresso até o momento, incluindo como o progresso atual em relação ao da semana anterior atingiu o planejado.
- Se os clientes selecionaram as estórias a serem implementadas

durante esta iteração.

- Divida as estórias em tarefas. Membros da equipe escolhem as tarefas em que vão trabalhar e estimam o tempo necessário para concluí-las.

Ciclo trimestral: As entregas devem ser planejadas a cada trimestre, trata-se de um plano de mais alto nível baseado em um tema, que é diferente de uma estória por se tratar da forma que o projeto se encaixa na organização e não aborda detalhes. Durante o planejamento do trimestre, a equipe deve identificar gargalos, iniciar reparos e escolher as estórias mais adequadas ao tema e que serão implementadas neste ciclo (SATO, 2007).

Folga: Durante o planejamento, deve ser considerada a imprecisão associada às estimativas, pelo fato de elas não passarem de previsões. Adicionar uma folga no formato de tarefas que possam ser descartadas caso haja necessidade, representa um grau de liberdade para assegurar que todas as tarefas principais sejam realizadas. Com essa folga, os desenvolvedores ficam mais confortáveis para assumir compromissos e o cliente tem segurança sobre o mínimo de *software* que será entregue no fim do ciclo (FILHO, 2008).

***Builds* em dez minutos:** Automatizar os *builds* do sistema todo, ou seja, pacote contendo o que foi desenvolvido até o memento, rodar todos os testes em dez minutos. Um *build* que necessita de mais tempo, será usada com menos frequência, possibilitando na perda de oportunidades para *feedback*. Um *build* em tempo menor não contribuirá muito (BECK; ANDRES, 2004).

Integração Contínua: As alterações feitas no código devem ser integradas ao código fonte que se encontra em um repositório compartilhado ao final de toda a tarefa. Uma vez que tudo está funcionando, um novo *build* é gerado e testado. Os desenvolvedores devem sempre buscar a versão mais atual do código neste repositório para trabalharem. Com esta prática, a dificuldade de realizar uma grande integração, se dilui em diversas integrações pequenas e frequentes (SATO, 2007).

Test-first programming ou Test-Driven Develpment: Escreva um teste antes de implementar qualquer código. Esta prática pode localizar diversos problemas:

- Foco no escopo: declarando objetivamente o que o programa deve fazer, é possível focar na codificação.
- Coesão e acoplamento: Se está difícil escrever um teste, é provável que haja um problema de projeto. Códigos com alta coesão e baixo acoplamento são fáceis de testar.
- Confiança: escrevendo códigos simples e mostrando durante as iterações que eles funcionam através dos testes automatizados, dá à equipe razões para confiar em quem está codificando.
- Ritmo: quando utilizada esta prática, é fácil saber o que fazer em seguida devido ao ritmo natural de teste em que o programador é envolvido: teste, código, refatorar, código, teste, etc. (BECK; ANDRES, 2004).

Projeto Incremental: Manter a simplicidade facilita a adaptação a mudanças, usando projetos mais simples, os desenvolvedores minimizam o custo com mudanças desnecessárias no futuro. Soluções simples, com o mínimo de complexidade e com flexibilidade, devem ser apresentadas para atender à necessidade do negócio (SATO, 2007).

Práticas Corolárias

Envolvimento Real do Cliente: Trabalhar diretamente com clientes reais proporciona resultados diferentes e melhores, pois é a eles que o sistema deve atender, portanto, pessoas que cujas vidas e negócio são afetados pelo sistema, devem fazer parte da equipe (BECK; ANDRES, 2004).

O papel do cliente dentro da equipe é escrever estórias, definir prioridades, responder dúvidas dos programadores e acompanhar a implantação, sempre que possível (FILHO, 2008).

Implantação Incremental: Em grandes implantações, além de arriscadas, têm custos humanos e econômicos muito altos. Ao substituir um sistema legado ou implantar um novo, é aconselhável não fazê-lo de uma só vez, é mais seguro a substituição / implantação, gradual (SATO, 2007).

Continuidade da Equipe: Conquistas resultam do talento e da sinergia das pessoas. Resultados positivos também são devidos a bons níveis de comunicação e colaboração, portanto mudanças nas equipes dificultam a criação de sinergia e deixam o sucesso depender apenas de habilidades individuais. Realocações frequentes para aproveitar pequenos períodos de disponibilidade das pessoas podem prejudicar os resultados de médio e longo prazo (FILHO, 2008).

Diminuição de Equipes: Beck e Andres, (2004) propõem uma prática usada pela empresa Toyota do Japão, onde conforme a capacidade de uma equipe cresce, seu tamanho deve ser reduzido para manter sua carga de trabalho constante, liberando pessoas para outras equipes. Quanto se torna muito pequena, deve-se integrá-la com outra também pequena.

Tentar fazer com que todos os membros pareçam ocupados, pode possivelmente esconder um excesso de recursos na equipe (SATO, 2007).

Análise da Causa Inicial: Para cada defeito encontrado, suas causas também devem ser corrigidas. Tão importante quanto corrigir o defeito, é identificar o que o causou evitando assim que ele ou outros com a mesma origem ocorram novamente. A estratégia sugerida para tratar defeitos é escrever um ou mais testes de aceitação automatizados que reproduzam o problema; escrever testes de unidade que capturem o defeito no menor escopo possível; implementar as devidas correções de forma que todos os testes passem; analisar porque o problema não foi percebido antes; analisar por que ele foi criado; e tomar atitudes para evitar que novos defeitos aconteçam e/ou que sejam evidenciados quando acontecerem (FILHO, 2008).

Código Compartilhado: Segundo Beck e Andres (2004), qualquer membro da equipe pode melhorar qualquer parte do sistema a qualquer momento.

O código-fonte é armazenado em um repositório compartilhado por toda a equipe, não há a necessidade de identificar responsáveis por cada parte do código, a equipe completa é responsável pelo sistema inteiro, assim os membros desenvolvem uma ampla visão do produto, facilitando qualquer ação a ser tomada em relação a ele (SATO, 2007).

Código e Teste: Mantenha somente código e teste como artefatos permanentes, outros documentos devem ser gerados a partir deles, deve-se confiar em mecanismos sociais para manter vivas estórias importantes do projeto (BECK; ANDRES, 2004).

Documentação adicional se desatualiza ou gera dependência que logo pode gerar esforço duplicado e desperdício. Mesmo cartões de estória devem ser descartados quando deixam de ser necessários, assim evita-se investir esforço sem acrescentar valor ao sistema (FILHO, 2008).

Repositório de Código Unificado: Apenas um repositório deve ser utilizado para armazenar o código, ramificações devem ser evitadas, pois quanto maior o número de versões concorrentes de um mesmo código, maior o trabalho para sincronizá-las e mais difícil o entendimento da equipe (SATO, 2007).

Implantação Diária: O novo *software* deve ser colocado em produção todas as noites, pois cada diferença entre o que está na máquina do programador e o que está em produção, é um risco (BECK; ANDRES, 2004).

Contrato de Escopo Negociável: Prioridades de negócio podem mudar, logo, os contratos para produção de sistemas devem fixar apenas a qualidade, o tempo e o custo, deixando o escopo como uma variável para ser definido à medida que o desenvolvimento evolui. Dessa forma, a funcionalidade a ser implementada pode ser escolhida à medida que as prioridades de negócios mudam e a equipe ganha domínio das regras de negócio. Este modelo permite que estimativas sejam mais precisas e a implementação cubra sempre as funções de mais alta prioridade (FILHO, 2008).

Pague-pelo-uso: Sugere a utilização do dinheiro como *feedback* final, o sistema é cobrado cada vez que é usado, o que permite obtenção de informações precisas e atualizada para direcionar melhorias do sistema, atualmente o modelo mais usado, é o pagamento a cada *release*, o que coloca em conflito interesses da equipe de desenvolvimento e o cliente, pois a equipe deseja um número maior de *releases* com pouca funcionalidade, o cliente deseja o menor número possível de *releases* contendo o máximo de funcionalidade, isso pode resultar em problemas de

comunicação e *feedback* (SATO, 2007).

2.4 Considerações do Capítulo

A evolução do mundo dos negócios afetou os métodos de desenvolvimento de *software*, pois o cenário atual da economia exige mais agilidade.

Método de desenvolvimento ágeis têm surgido em resposta a essa necessidade, uma breve descrição de alguns deles foi fornecida neste capítulo mostrando como os princípios de cada um deles permite que o desenvolvimento do *software* tenha resposta rápida e se adéque melhor às necessidades do cliente e do mundo corporativo.

Surgindo novos métodos, novas técnicas também são necessárias para suportá-los, o *Test Driven Develpment* é uma delas, muitas vezes referenciada como *test-first programming* esta técnica recebe maior foco neste trabalho através do capítulo 4.

Por ter sido popularizada e estar diretamente ligada ao *eXtreme Programming*, este método foi o mais explorado. É possível detectar no decorrer da descrição desse método, diversas características intrínsecas ao TDD, que é discutido posteriormente.

3. QUALIDADE DE *SOFTWARE* EM MÉTODOS ÁGEIS

Este capitulo descreve como a qualidade do *software* é verificada através do desenvolvimento de *software* inserido em um ambiente onde um método ágil é aplicado, expondo de forma sucinta a definição de testes de *software* e as atividades necessárias para garantia de qualidade nestes métodos.

3.1 Qualidade de S*oftware*

Diversas definições são atribuídas à qualidade, em poucas palavras, pode-se dizer que a qualidade do produto significa perfeita adequação ao uso. Pode ser considerada como características do produto que satisfazem a necessidades implícitas ou explícitas do cliente, sendo livre de deficiências (STAMELOS, 2007).

Vale ressaltar que para atender à necessidade do cliente, é preciso entendê-la antes de iniciar qualquer atividade, Black et al. (2006) afirmam ser muito importante que a equipe de projeto e o cliente definam as expectativas e necessidades a serem atendidas.

Para Pressman (2009), qualidade é a conformidade com requisitos funcionais explicitamente definidos, padrões de desenvolvimento explicitamente documentados e também características implícitas já esperadas de qualquer *software* desenvolvido profissionalmente.

Dentro de desenvolvimento de *software*, três fatores citados por Júnior (2008) também devem ser considerados para proporcionar qualidade:

- O número e a gravidade de defeitos residuais do processo de testes devem ser aceitáveis;
- O *software* deve ser entregue dentro do prazo e custo previstos, atendendo aos requisitos e expectativas do cliente e desenvolvedor;
- O *software* deve ser projetado de forma que, quando identificado um defeito após sua implantação, sua manutenção seja realizada de forma eficiente.

Júnior (2008) prossegue afirmando que adotar atividades de validação, verificação e testes é uma maneira de promover qualidade de *software*, pois elas ajudam a eliminar possíveis riscos aos quais o processo de desenvolvimento está suscetível.

Black et al. (2006) trazem a definição os termos acima como:

Verificação: Confirmação por exame e por meio do fornecimento de evidências objetivas de que os requisitos especificados foram cumpridos.

Validação: Confirmação por exame e através do fornecimento de evidências objetivas de que os requisitos para um uso pretendido ou aplicação específica foram cumpridos.

Testes: Processo aplicado em todas as fases do ciclo de vida de desenvolvimento de *software*. Podem ser estáticos (testando documentação ou código fora de produção) e dinâmicos (aplicados em códigos em execução).

Os testes têm como preocupação o planejamento, preparação e avaliação de produtos de *software* e outros produtos relacionados ao *software* para determinar que satisfaçam os requisitos especificados, para mostrar que eles são adequados à finalidade e para detectar defeitos.

3.2 Testes no Ciclo de Vida de Desenvolvimento de S*oftware*

Atividades de teste não são atividades independentes, elas têm o seu lugar dentro de um modelo de ciclo de vida desenvolvimento de *software* e, portanto, o ciclo aplicado determinará em grande parte como o teste é organizado (BLACK et al. 2006).

Isso pode ser muito bem observado em projetos tradicionais, onde controle de qualidade é feito por meio de um planejamento detalhado e controle rígido, além de uma grande quantidade de testes que são executados, normalmente, no ciclo de vida do *software* (VICENTE; DELAMARO; MALDONADO, 2010).

No modelo cascata (FIGURA 4), por exemplo, há uma sequencia de tarefas

executadas uma após a outra, começando pelo estudo de viabilidade, passando por diversas tarefas do projeto, chegando aos testes apenas quando a implementação está completa. Por este motivo, os defeitos são detectados próximos da data de entrega, este modelo torna difícil o *feedback* e dificulta o processo de qualidade devido aos testes normalmente acontecerem com um tempo escasso e a equipe estar sobre pressão.

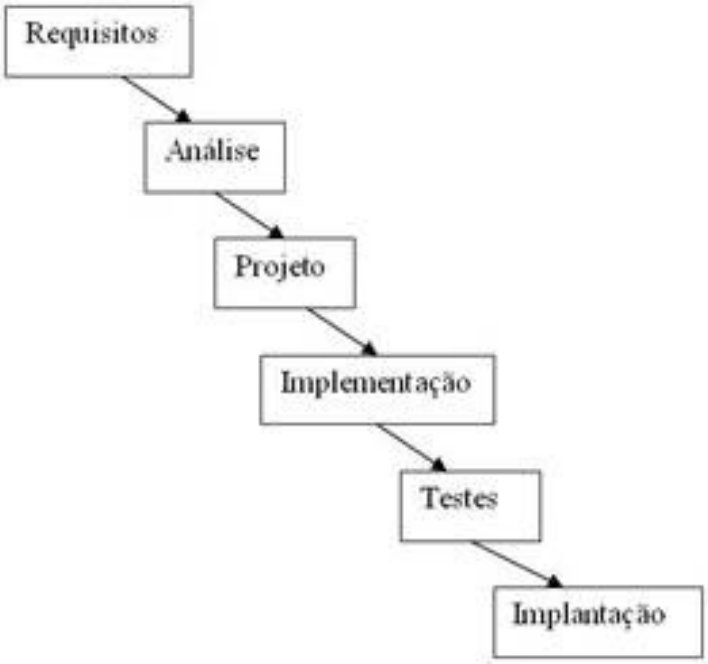

Figura 04: Modelo Cascata (Adaptado de BLACK et al., 2006).

Para suprir essas deficiências do modelo Cascata, o Modelo V foi desenvolvido, ele orienta começar os testes desde o início do ciclo de vida e que eles aconteçam em paralelo às atividades de desenvolvimento. Mostra que o teste vai além de execução, que existem atividades que precisam ser executadas antes e depois da fase de codificação, como por exemplo, testes de validação que acontecem tanto durante os primeiros estágios assim como no final do projeto (BLACK et al., 2006).

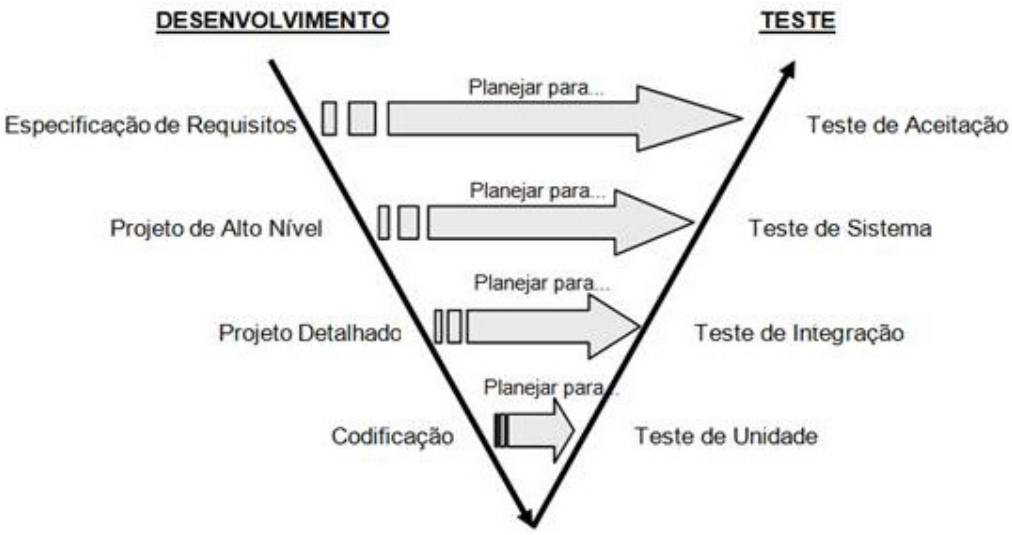

Figura 05: Modelo V – (Adaptado de Black et al. 2006).

Os testes em modelos iterativos podem seguir o modelo V dentro de cada iteração. Quando se tratam de métodos ágeis e seus valores, é necessário repensar a maneira de testar e aplicar diversas práticas para controle da qualidade conforme descrito na próxima seção.

3.3 Qualidade de S*oftware* em Métodos Ágeis

Métodos ágeis promovem mudanças evolutivas durante o processo de desenvolvimento do *software*, para aumentar o controle do projeto e a garantia de qualidade do produto, conta com um conjunto de boas práticas. Esse conjunto forma um processo disciplinado com procedimentos de garantia e controle aplicados em todo o ciclo de vida de desenvolvimento, dos requisitos ao *release*. Um produto é construído através da combinação de algumas dessas práticas que se mostrarem mais adequadas para o projeto, induzindo a uma nova perspectiva de gestão de qualidade em desenvolvimento de *software* (SFETSOS; STAMELOS, 2010).

A Tabela 03 traz uma síntese das técnicas que definem qualidade em métodos ágeis normalmente aplicados em *Extreme Programming* (XP).

Tabela 03 – Técnicas e atividades voltadas para a qualidade em métodos ágeis – Adaptado de Stamelos, (2007).

Técnica	Descrição
Refatoração	Faça pequenas mudanças no código onde seu comportamento não pode ser afetado e o resultado é um código de maior qualidade.
Test-Driven Development	Crie um teste, execute este teste, faça as mudanças necessárias até que este teste seja bem sucedido.
Teste de aceitação	Teste de qualidade feito em um sistema concluído costuma envolver usuários, clientes, *stakeholders*, etc.
Integração Contínua	Deve haver um *build* diário após certa quantidade de estórias de usuários estar implementada, e assim, requisitos implementados e integrados são testados.
Programação em Pares	Dois desenvolvedores juntos trabalhando em uma estação, defeitos são identificados assim que ocorrem proporcionando uma alta qualidade ao produto.
Comunicação face-a-face	Maneira desejada de troca de informações com o cliente. Reuniões diárias de no máximo 15 minutos são sugeridas, isso traz responsabilidade para o trabalho em progresso, o que é vital para qualidade.
Envolvimento real do cliente	Um cliente que faz parte da equipe de desenvolvimento, responsável por esclarecer requisitos.
Feedback contínuo do cliente	A cada fechamento de iteração, o cliente fornece um *feedback* sobre o que está pronto até o momento, com isto, as melhoras do sistema são mais relevantes para atender às necessidades do cliente. Qualidade é de fato, atender necessidades do cliente.
Sistema de metáforas	Estórias simples de como o sistema trabalha facilitam a discussão entre cliente, *stakeholder*, usuário e desenvolvedor de uma forma não técnica. Simplicidade é uma característica chave para qualidade.

3.4 Testes de *Software* em Métodos Ágeis

O conceito de métodos ágeis rompeu paradigmas que se aplicavam no desenvolvimento de *software*, trazendo efeitos em todo o processo, com as atividades de testes não foi diferente.

Um processo de testes apropriado continua sendo um componente crítico para a qualidade de um *software* em desenvolvimento, porém, diferente de métodos tradicionais, em muitos contextos ágeis esta atividade passa a ser executada desde o início do processo detectando defeitos o mais cedo possível. Sendo assim, um testador independente não é eficiente em um projeto ágil, ele deve trabalhar em estreita interação com os desenvolvedores e estes por sua vez, também devem assumir a responsabilidade de garantir qualidade do produto através de técnicas como testes de unidade automatizados, *Test-Driven Development*, entre outras (KETTUNEM, et al., 2010).

O sucesso dos testes pode ser comprometido se abordagens burocráticas que vão desde o planejamento de teste, passando por revisão de casos de teste e até fechamento de defeitos forem adotadas, pois os ciclos de cada iteração são curtos.

Os testes agora validam novas funções e mudanças de forma contínua podendo ser utilizados como documentação por não conterem ambiguidades, estarem sempre atualizados e poderem ser executados tanto pelo desenvolvedor como pelos clientes. Observa-se então um novo ponto de grande preocupação dos testes em um ambiente ágil, o de testar sob o ponto de vista do cliente por meio de testes de aceitação, cada estória de usuário deve ter um ou mais casos de teste de aceitação associados, que devem validar o *software* (VICENTE; DELEMARO; MALDONADO, 2010).

Testes automatizados são considerados um fator chave para testes em contextos ágeis, com eles é possível manter testes e desenvolvimento em sincronia e também garantir que o *software* ainda funcione a cada nova iteração sem que muito esforço manual seja necessário, possibilitando que novas funções sejam tratadas com mais atenção uma vez que os testes automatizados garantem o funcionamento do que já foi implementado (KETTUNEM, et al., 2010).

A Tabela 04 traz uma lista de atividades de verificação, validação e testes (VV&T) nos métodos ágeis abordados anteriormente.

Tabela 04. Técnicas e práticas de VV&T em Métodos Ágeis.Adaptado de (VICENTE; DELAMARO; MALDONADO., 2010).

Método	Atividades
XP	Integração contínua e testes *Test-Driven Development* para teste de unidade e integração Testes de aceitação associados a histórias do usuário
SCRUM	*Builds* diários e testes (não especifica nenhuma técnica de teste). Testes durante o *sprint* e teste de sistema (após *sprint*).
Família *Crystal*	Testes são executados durante a iteração Testes de regressão de funcionalidade automatizados Nível de formalidade e documentação dos testes conforme o time e complexidade do projeto
FDD	Inspeção do projeto e do código Testes de unidade (após implementação).

3.4.1 Quadrantes dos Testes Ágeis

Crispin e Gregory (2009) organizaram as técnicas de teste que podem ser usadas em testes ágeis, conforme a FIGURA 06.

Cada quadrante reflete diferentes necessidades de testes. No eixo 1, estão os testes que dão suporte às equipes e que avaliam o produto. Nos demais, são relacionados os testes voltados para o domínio e tecnologia. A maneira em que os testes foram organizados não está relacionada à ordem em que devem ser executados.

A seguir, segue a descrição de cada quadrante segundo Crispin e Gregory (2009):

- Quadrante 1: São considerados como testes a serem executados pelos programadores, analisando o código ainda em desenvolvimento permitindo medir a sua qualidade interna. Estão subdivididos em teste de unidade, que verifica uma pequena parte do sistema, como um objeto ou método, já o teste de componente verifica o comportamento abrangendo uma parte maior do sistema. Ambos costumam ser automatizados.

- Quadrante 2: Neste quadrante, são destacados os testes baseados na visão do negócio ou do cliente que define a qualidade externa e as características solicitadas pelo cliente, o desenvolvimento também é dirigido por estes testes, mas em um nível mais alto sendo também automatizado. Verificam a funcionalidade do

produto através de exemplos fornecidos pelos clientes.

Os testes contidos nestes dois quadrantes têm como um de seus principais propósitos, fornecerem informação de forma rápida possibilitando a correção do problema o quanto antes. O fato de serem automatizados possibilita que sejam executados constantemente e auxiliem em mais um dos valores dos métodos ágeis, o constante *feedback*.

- Quadrante 3: Exemplos fornecidos pelos clientes sempre ajudam no desenvolvimento do produto, mas alguns deles podem ser interpretados de forma errada, uma funcionalidade pode até ir além do que é esperado, portanto, mesmo que os testes baseados nas informações fornecidas pelo usuário passem, o produto pode não atender a real expectativa do cliente.

Verificar se o produto está realmente exercitando o que cliente sugeriu e se tem competitividade de mercado é o objetivo dos testes desse quadrante. Por terem que simular um ambiente real da maneira mais fiel possível, são em grande parte executados manualmente.

Outro valor do Manifesto Ágil é fortemente aplicado: “a colaboração do cliente é mais importante do que negociação de contratos”. Através da participação dos clientes nos testes de aceitação eles podem sugerir novas ideias que venham a surgir durante esta atividade.

Mas o grande foco desse quadrante são os testes exploratórios onde o profissional da área de teste simultaneamente desenvolve o projeto e executa testes baseado em seus conhecimentos sobre o produto, criatividade, intuição e análise de riscos, sendo possível aprender mais o produto como se fosse um usuário final, por não seguir *scripts* de testes pré definidos.

- Quadrante 4: Tão importantes em métodos ágeis como em qualquer outro tipo de desenvolvimento, são os testes que se encontram no quarto quadrante, intitulados testes de tecnologia, mas também conhecidos como testes não funcionais, como teste de desempenho, segurança, entre outros.

O foco da validação deixa de ser as funções do negócio e se voltam para

aspectos tecnológicos a serem supridos, o que exige muito conhecimento técnico dos envolvidos nestas atividades.

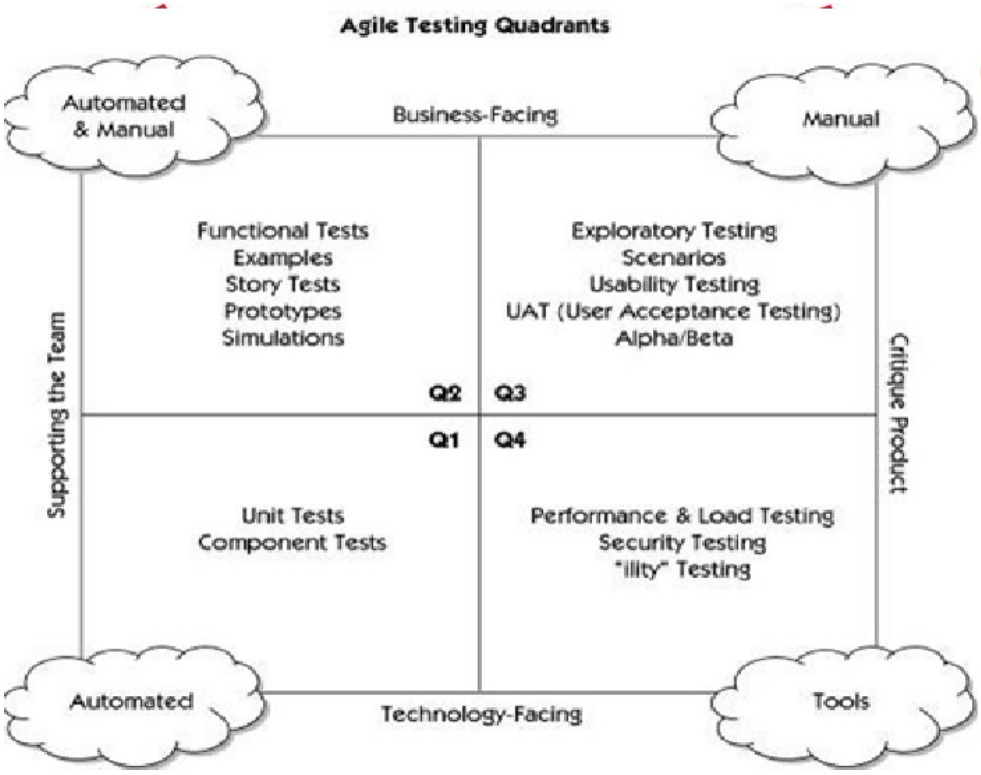

Figura 06: Quadrantes de Testes em ambientes ágeis – (CRISPIN; GREGORY, 2009).

3.5 Considerações do Capítulo

Neste capítulo foram apresentadas algumas definições sobre qualidade de *software* assim como as atividades voltadas para controle de qualidade a serem executadas em diferentes contextos de acordo com o modelo de desenvolvimento seguido. Foram citadas brevemente as tarefas de verificação, validação e testes e como elas são aplicadas em modelos tradicionais, assim como os motivos que levaram à criação do modelo “V” de testes que promovem essas tarefas desde o início do ciclo de vida de desenvolvimento.

Finalmente abordam-se métodos ágeis e a necessidades de novas tarefas para atender os valores por elas introduzidos, neste contexto o TDD é uma técnica de extrema importância,pois auxilia na construção de um produto de maior qualidade uma vez que para se escrever seus testes, é necessária uma maior reflexão por parte do codificador em relação aos requisitos fornecidos pelo cliente,

além disso, por se tratar de testes automatizados, auxilia nos testes de regressão conforme o produto evolui.

4. *Test-Driven Development* (TDD)

Este capítulo faz uma breve apresentação da técnica do TDD mostrando como vem auxiliar no desenvolvimento de um *software* em um ambiente ágil assim como seu papel em relação aos testes de *software*.

4.1 *Test-Driven Development*

Falhas de *software* demandam custos e tempo no processo de desenvolvimento de *software* e a causa dessas pode ter origem em erros de conhecimento, falha de comunicação, equívocos durante análise, defeitos de codificação, entre outros.

De acordo com Black et al. (2006) um dos princípios de teste é que testar todas as combinações de valores e pré-condições não é viável.

Para Borges (2007), existem três formas de tratar falhas de software:

Evitar: com atividades apropriadas de especificação, projeto, implementação e manutenção sempre visando evitar falhas em primeiro lugar. Pode ser exemplificada pelo reuso de blocos de *software* confiáveis.

Eliminar: através de atividades de verificação, validação e testes, detectam-se erros cometidos durante a especificação, projeto e implementação.

Tolerar: compensação em tempo real de problemas residuais como mudanças fora da especificação no ambiente operacional, erros de usuário, etc.

Grande parte dos métodos tradicionais adota a técnica de eliminação de falhas, onde testes são executados no final do ciclo de desenvolvimento. Mesmo em métodos iterativos como o *Rational Unified Process* (RUP), o desenvolvimento do sistema é dividido em pequenas partes, onde são trabalhados componentes, mas ainda assim, os testes frequentemente são tratados como uma atividade pós-construção, ainda que ocorram diversas vezes durante o ciclo de vida do projeto (STAMELOS, 2007).

Esse tipo de abordagem resulta em falhas encontradas tardiamente e, portanto, impactos negativos maiores.

Remover todos os defeitos existentes em aplicação não é possível, então pode-se reduzir consideravelmente o número dos mesmos utilizando uma infra-estrutura de testes mais elaborada, que permita identificar e remover defeitos mais cedo e de forma mais eficaz (BORGES, 2007).

Para suprir esta necessidade, o TDD é uma técnica adequada, pois apesar de seu nome estar relacionado a testes, ele vai além disso e orienta como usar testes para criar *software* de maneira simples e incremental além de melhorar a qualidade e o projeto do *software.* Ele também simplifica o processo de desenvolvimento (SANTOS, 2010).

Seu objetivo é a especificação e não a validação. Em outras palavras, é uma forma de refletir sobre a modelagem antes de escrever código funcional a partir de um teste que tenha falhado. Como efeito colateral, obtém-se um código fonte bem testado e limpo (GASPARETO, 2005).

O uso desse tipo de desenvolvimento orientado a testes pode proporcionar uma economia considerável para empresas de desenvolvimento e aumentar a qualidade do *software* produzido, uma vez que o TDD visa antecipar a identificação e correção de falhas durante o desenvolvimento do *software* (BORGES, 2007).

Além disso, em processos que seguem métodos tradicionais, há uma lacuna entre as decisões tomadas em seu início e o *feedback* fornecido através da implementação. Com o uso do TDD, essa lacuna pode ser reduzida uma vez que seus ciclos são menores que possibilitam *feedbacks* constantes ao desenvolvedor. Logo as falhas são descobertas rapidamente e sua origem é identificada mais facilmente (SANTOS R., 2010).

Nos métodos ágeis, a qualidade interna de um *software* não é negociável, ela deve ser fixada como ótima durante todo o ciclo de vida do *software*. Qualidade externa é a qualidade medida pelo cliente e qualidade interna é a qualidade medida pelas pessoas que possuem acesso ao código, como é o caso dos programadores e

arquitetos (RIBEIRO, 2010).

Segundo Stamelos (2007), os métodos ágeis têm a premissa de que qualquer componente construído terá alta qualidade.

4.2 O Ciclo de Funcionamento do TDD

Sua prática consiste em primeiramente escrever o teste e depois o código. Em seguida, fazer este teste passar e então passar para a fase de *design*. Esta fase de *design* é diferente dos processos tradicionais. No TDD, ela é chamada de refatoração, onde o *design* atual do código é transformado em um *design* melhor. O ciclo Teste-Codificação-Refatoração também pode ser chamado de ciclo Vermelho-Verde-Refatoração onde o vermelho simboliza a fase inicial do ciclo, o teste escrito falha porque o sistema não possui a funcionalidade a ser testada. Em alguns ambientes de desenvolvimento, essa falha é evidenciada através da exibição de uma barra vermelha, como pode ser observado na Figura 07.

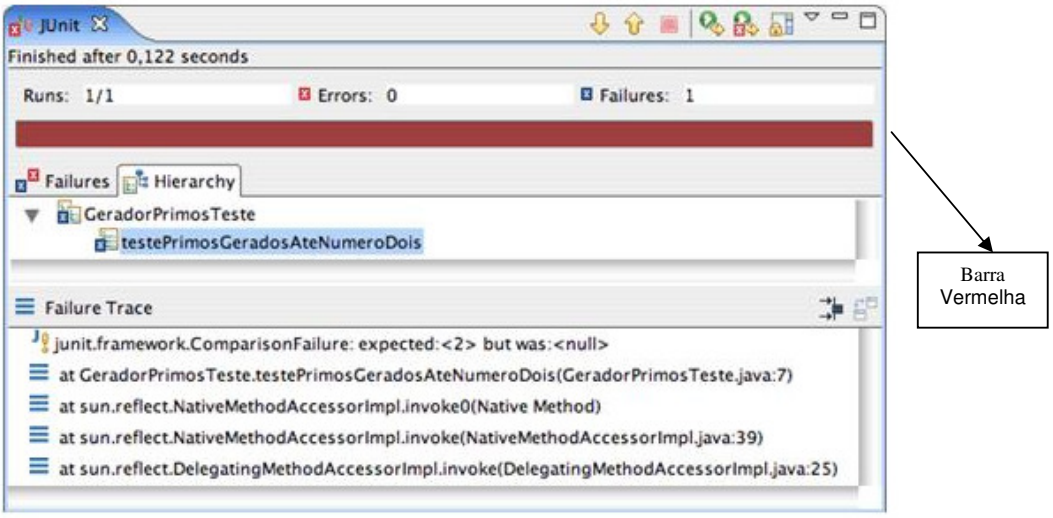

Figura 07 – Tela da ferramenta *JUnit* com barra vermelha indicando falha no teste (ARRUDA, 2011).

O próximo passo então é implementar a funcionalidade que falta para fazer todos os testes passarem. Neste momento, a barra visual deve se tornar verde. Somente se passa para a próxima etapa quando nenhum teste estiver falho.

Na última parte do ciclo é feita a refatoração onde o *design* do código é

refinado sem alterações em seu comportamento externo, mantendo todos os testes passando e a com a barra visual exibindo a cor verde (RIBEIRO, 2010).

Estes testes são automatizados para que sejam executados repetidamente. Conforme o resultado de sucesso ou falha, o progresso do desenvolvimento é julgado e desta forma, os programadores fazem continuamente pequenas decisões aumentando as funções do *software* a uma taxa relativamente constante. Todos estes testes devem ser realizados com sucesso sucessivamente antes de o novo código ser considerado totalmente implementado (BORGES, 2010).

Este ciclo é ilustrado na Figura 08:

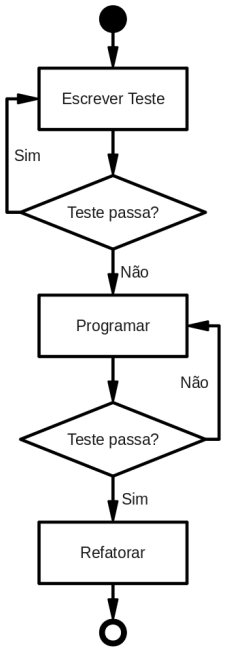

Figura 08 - Fluxograma do ciclo *Test-Driven development* (SANTOS, 2010).

De acordo com Beck (2003), o TDD segue o ciclo abaixo:

1 – Escreva um teste, pense em como a operação deve aparecer no seu código. Tome nota de como deve ser a interface, todos os elementos necessários para calcular as repostas certas.

2 – Execute este teste. Faça com que a barra vá de vermelho para o verde.

Se uma solução simples e limpa é óbvia, codifique-a, mas se ela for tomar um minuto, então tome nota e retorne ao problema principal que é tornar a barra verde em segundos.

3 – Faça isto corretamente. Agora que o sistema está funcional, dê um passo atrás remova as duplicidades introduzidas.

De acordo com Santos (2010), para praticar o TDD corretamente é necessário escrever testes de uma forma diferente, é preciso aprender como escrever testes efetivos que possam ser usados para guiar o desenvolvimento do *software*. A maioria dos desenvolvedores entende como escrever testes para testar código já escrito, mas escrever testes antes do código pode necessitar uma abordagem diferente sendo necessário se concentrar em como testar a funcionalidade de um código ao invés de sua implementação.

Há três pontos que devem ser observados em relação aos testes de TDD segundo Gaspareto (2005):

- Se forem necessárias muitas linhas de código criando objetos para uma simples função, então há algo de errado;
- Se não é possível encontrar facilmente um lugar comum para o código de iniciação, então existem muitos objetos firmemente associados;
- Testes que deixam de passar inesperadamente sugerem que uma parte da aplicação está afetando outra parte. É necessário projetar até que esta distância seja eliminada, desfazendo esta conexão ou trazendo as duas partes juntas.

É possível afirmar que o TDD é um conjunto de iterações realizadas para completar uma tarefa cujo início se dá em escolher a área do projeto ou requisitos da tarefa para melhor orientar o desenvolvimento (BORGES, 2010).

4.3 TDD no Contexto de Teste de *Software*

Existem vários níveis de testes na área de qualidade de *software* como teste de unidade, funcional, de sistema, entre outros. O TDD não tem como objetivo abordar cada aspecto desses testes, ele possibilita aumentar a qualidade que está

embutida no código desde o início e incentiva os desenvolvedores a pensar sobre testabilidade (STAMELOS, 2007).

Ainda assim, existe uma grande confusão entre TDD e testes de unidade, segundo Black et al. (2006). Testes de componentes e testes de unidade procuram defeitos e verificam o funcionamento de componentes de *software* que são separadamente testáveis, podem ser feito de forma isolada do resto do sistema e normalmente ocorrem com o acesso ao código que está sendo testado e com o apoio do ambiente de desenvolvimento.

Como já mencionado, em TDD, ao contrário do modo tradicional de desenvolvimento de *software*, onde se começa criando um projeto que irá guiar a implementação se começa criando um teste que define como uma parte do sistema deve funcionar, e portanto, testes do TDD são escritos com o objetivo de guiar o desenvolvimento do *software*, enquanto testes de unidade têm objetivo de testar a implementação já codificada de uma unidade. De qualquer forma, o código produzido é automaticamente testável no nível de unidade porque é escrito para fazer passar algum teste de unidade (SANTOS R. 2010).

Outra dúvida recorrente é se estes testes devem ser automatizados ou não, Beck (2003) afirma que em TDD deve-se escrever um novo código se um teste **automatizado** falhar, portanto, conclui-se que a automação é uma premissa para o TDD, o que pode ser justificado através da afirmação de Pedrini (2009). Ele diz que a automação de teste facilita o trabalho desta disciplina que é encontrar falhas, se escrever bons testes é difícil, mantê-los é mais ainda.

Vale lembrar que somente o TDD não garante a obtenção de níveis aceitáveis em certos aspectos do *software* final. Santos (2010), justifica essa afirmação quando diz que o código produzido é automaticamente testável em nível de unidades porque o código em TDD apenas é escrito para fazer passar algum teste de unidade. O que não garante que será testável em níveis mais altos, como testes de sistema. Logo outros níveis de testes são necessários para avaliar a qualidade do produto.

Além disso, o TDD não consegue mitigar riscos relacionados com a falta de requisitos ou com requisitos erroneamente definidos (RIBEIRO, 2010).

4.4 Benefícios da Utilização do TDD

Testes dão a chance de pensar sobre o que é preciso, independente da forma como a solução será implementada. Utilizando o TDD, testes são escritos para cada solução implementada. Dessa forma, diminui-se a probabilidade uma decisão errada. Ao mesmo tempo, há a oportunidade de experimentar várias implementações diferentes para o mesmo problema e escolher aquela mais limpa, elegante e que apresente o melhor desempenho (RIBEIRO, 2010).

Alguns dos benefícios proporcionados pelo TDD são descritos a seguir.

4.4.1 Código Simples e Manutenível

TDD utiliza uma das técnicas do XP chamada refatoraçao para conseguir a compreensão do código e gerenciar a complexidade do mesmo. Um grande programa ou sistema continuamente modificado torna-se muito complexo, sendo extremamente necessária a facilidade de manutenção (BORGES, 2007).

Além disso, grandes refatorações não mudam o comportamento do sistema, o que se conclui que, há maior confiança no código. Custos são reduzidos porque a refatoração contínua evita que o *design* se degrade com o passar do tempo, assegurando que o código continue fácil de ser entendido, mantido e modificado (RIBEIRO, 2010).

4.4.2 Melhor Entendimento dos Requisitos de S*oftware*

Ao escrever os testes, uma compreensão de como a unidade deve se comportar é construída à medida que os critérios de sucesso dos testes vão sendo estabelecidos e as assertivas codificadas. Assim, um teste define bem quais comportamentos a unidade deve ter e em um nível de detalhamento mais concreto do que possivelmente um documento de requisitos (SANTOS R. 2010).

Testes bem escritos atuam como um tipo de requisitos executáveis que ajudam a manter o entendimento compartilhado da equipe de desenvolvimento, sobre como o sistema de *software* representa os problemas do mundo real. O programador que tem um entendimento mais rápido e facilitado do que uma parte do

código faz através do código que o testa (RIBEIRO, 2010).

4.4.3 *Feedback* Contínuo

Por se tratar de ciclos pequenos de testar e então codificar, o *feedback* se torna contínuo ao programador, logo, falhas são identificadas mais rapidamente, enquanto o novo código é adicionado ao sistema. Assim, o tempo de depuração diminui compensado pelo tempo de escrita e execução dos casos de teste (BORGES, 2007).

Portanto, o TDD o ajuda a dar atenção aos problemas certos no momento certo, de forma que o *design* do *software* fica mais limpo e com muito menos defeitos. Além disso, ele encurta o ciclo de *feedback* sobre as decisões de desenho.

Ele dura apenas segundos ou minutos e é seguido pela reexecução de testes dezenas ou centenas de vezes por dia. Ao invés de se projetar um *design* e então esperar semanas ou meses para verificar o resultado, o *feedback* emerge em segundos ou minutos (RIBEIRO, 2010).

4.4.4 Diminuição do número de defeitos

Em razão do princípio de que apenas deve-se escrever código suficiente para fazer algum teste passar, os testes tendem a atingir um percentual muito alto de cobertura o que reduz a quantidade de defeitos, pois provavelmente o motivo principal para um defeito entrar em produção é que não houve teste que verificasse se um caminho de execução particular do código de fato funciona devidamente. Com TDD praticamente não há código no sistema que não seja executado pelos testes (SANTOS R., 2010).

4.4.5 Conjunto de Testes (Regressão)

Usando TDD, os testes de unidade são criados num momento onde a funcionalidade a ser implementada está mais bem definida na mente do programador e depois podem e devem ser utilizados para fazer testes de regressão. Um conjunto de testes automáticos feito por programadores reduz os custos de um *software* funcionando como uma rede de segurança de testes que capturam antes

defeitos, problemas de comunicação e ambiguidades e permitem que o *design* possa ser modificado de forma incremental (RIBEIRO, 2010).

Já na etapa de manutenção, esses testes permitem avaliar mais facilmente novos defeitos que podem ter sido inseridos no *software*. Este benefício é essencial para o desenvolvimento e controle de novos *releases*, reduzindo a injeção de novos defeitos no sistema (BORGES, 2007).

4.5 Considerações do Capítulo

Neste capítulo foi introduzido o conceito sobre a técnica de desenvolvimento *Test-Driven Development* que trata-se de uma técnica de desenvolvimento que auxilia no desenvolvimento de códigos mais simples de maneira incremental que pode auxiliar a identificar e remover defeitos no código mais cedo e de forma mais eficaz.

Os passos definidos por Beck (2003) foram listados assim como algumas vantagens que seu uso pode trazer.

Foi observado que o TDD por si só não é capaz de garantir a qualidade do *software* em produção uma vez que seu objetivo é testar o código, outros níveis de verificação e validação são necessários para cobrir outros aspectos de qualidade.

5. ANÁLISE DA APLICAÇÃO DO TDD

Este capítulo apresenta uma análise de trabalhos cujo objeto de estudo foi à aplicabilidade do TDD. Através deles é possível observar quais são as vantagens trazidas pela utilização dessa técnica, suas desvantagens assim como potenciais falhas que podem acontecer durante sua aplicação.

Com base nessas falhas, foram extraídas sugestões a serem observadas durante a aplicação do TDD com o objetivo de obter maior grau de aproveitamento do seu uso. Uma lista de boas práticas, tendo essas sugestões como origem, é apresentada, assim como a metodologia utilizada para averiguar sua real relevância.

5.1 Trabalhos Relacionados Sobre a Aplicabilidade do TDD

Nos capítulos anteriores, foi descrito que o TDD é uma prática aplicada em desenvolvimento de código que se tornou amplamente conhecida após o surgimento do método *Extreme Programming* (XP).

Diversos estudos sobre este assunto foram conduzidos. Inicialmente o objetivo mais recorrente entre esses estudos foi verificar quais as diferenças entre os resultados de um projeto aplicando o TDD e outro aplicando o *"Test-Last"* ou "testes tradicionais", como são referidos os testes aplicados quando código já está pronto, ao contrário do TDD onde o teste é o primeiro passo no desenvolvimento.

O trabalho de Gupta e Jalote (2007) é um exemplo típico desses trabalhos, que segue um formato comum de experimento controlado no qual geralmente dois grupos, compostos por estudantes em sala de aula ou profissionais da área na indústria, desenvolveram um código cada, sendo que um fez uso do TDD e o outro, meios tradicionais. Uma vez concluídos, ambos os códigos foram submetidos a testes de aceitação onde o número de defeitos encontrados nessa fase foi comparado a fim de verificar qual das abordagens é mais eficiente. Nessa avaliação também consideraram-se fatores como esforço aplicado e produtividade.

Diversos trabalhos como este foram produzidos, e a partir destes, a análise de dados coletados durante esses experimentos se tornaram o objeto dos estudos.

O trabalho de Kollanus (2010) se enquadra nesse novo tipo de trabalho onde o foco se volta para relatos de pesquisas empíricas sobre o TDD cujo objetivo foi validar se realmente há evidências que comprovem sua eficiência.

Neste tipo de trabalho, a metodologia seguida foi o levantamento de publicações anteriores provenientes de fontes confiáveis como revistas científicas, conferências técnicas, entre outras, através de diretrizes bem conhecidas de revisões de literatura.

Em uma revisão sistemática da literatura, sugere-se selecionar estudos utilizando critérios claros de inclusão / exclusão com base em questões de qualidade, como participantes, projetos de pesquisa e método de amostragem (KOLLANUS, 2010).

Uma vez que os trabalhos foram identificados, uma análise de seus dados através dos relatos fornecidos por cada um deles foi realizado sendo possível agrupar as evidências mais recorrentes.

O resultado do estudo de Kollanus (2010) sugeriu que os tópicos mais citados como resultado da aplicação do TDD foi à melhora da qualidade externa e interna do código e queda produtividade dos codificadores por terem que desenvolver testes, além do código propriamente dito.

Mais um ponto comum nesse tipo de análise foi justamente a qualidade dos dados coletados, Kollanus (2010) afirma que existem muitos resultados contraditórios e isto coloca em dúvida os fatores por trás deles.

Buchan, Li e Macdonell (2011) reforçam esta constatação dizendo que há muitos estudos investigando a qualidade das aplicações desenvolvidas com o uso do TDD comparada ao uso de testes tradicionais, mas seus resultados são variados e inconclusivos.

Kollanus (2010) e Buchan, Li e Macdonell (2011) concluem que experimentos controlados mais bem elaborados e conduzidos são necessários para melhor compreensão sobre o TDD, pois as conclusões baseadas nesses trabalhos deixaram algumas questões sobre os fatores reais colocando em dúvida a

qualidade e eficácia da técnica.

Kollanus (2010) aponta outro fator importante, que normalmente são relatadas experiências de implementação de TDD, mas muitas vezes poucos detalhes sobre o processo de desenvolvimento com TDD.

Essa afirmação torna possível assumir que há a possibilidade de haver erros na aplicação da técnica do TDD propriamente dita durante os experimentos e, consequentemente, em ambientes reais de desenvolvimento.

Quanto a isso, Buchan, Li e Macdonell (2011) são incisivos onde a técnica de TDD deve ser seguida rigorosamente para perceber os benefícios. Acrescentam que há uma série de equívocos e erros comuns na implementação de TDD na prática diária de desenvolvimento, é mais do que apenas escrever os testes primeiro. (BUCHAN, LI e MACDONELL, 2011).

Esse tipo de observação leva a um terceiro formato de estudos sobre TDD, ou seja, aquele que identifica falhas durante sua aplicação devido a diversos fatores, interferindo de forma negativa em seus resultados. Nota-se que ainda se trata de um foco menos recorrente na literatura sobre o assunto.

Dois desses trabalhos foram selecionados e são discutidos a seguir.

5.2 Falhas na Aplicação do TDD

Calsevic, Sundmark e Punnekkat (2011), observaram que os resultados obtidos através da aplicação da técnica não atingem as expectativas de melhoria de qualidade , logo o TDD não era tão utilizado quanto se esperava. A fim de identificar obstáculos que podem limitar a sua adoção nas empresas, uma revisão sistemática de estudos realizados sobre TDD foi conduzida. Qualquer trabalho cujo tema fosse TDD foi estudado, independente do foco de sua abordagem. Desses trabalhos, qualquer crítica ou apontamento de problema citados foi considerado. Foram levantados sete potenciais fatores que influenciaram de maneira negativa as empresas no momento de decidir pela adoção do TDD em seu ambiente de desenvolvimento, que são:

1. Aumento do tempo necessário para o desenvolvimento,
2. Experiência / conhecimento do desenvolvedor sobre o TDD insuficiente,
3. Falta de *design* inicial do domínio de projeto,
4. Uso de ferramentas específicas,
5. Falta de habilidade / capacidade do desenvolvedor em escrever casos de teste,
6. Adesão insuficiente aos passos do TDD,
7. Aplicação da técnica em código legado.

Já Aniche e Gerosa (2010) fizeram um levantamento de alguns dos erros mais cometidos durante a aplicação da técnica. Segundo eles o TDD é uma técnica simples, mas nem sempre os codificadores aplicam todos os passos obrigatórios de maneira correta [...] o que pode reduzir o valor que o TDD agregaria ao *software*.

Inicialmente foi feito um levantamento de dados através dos trabalhos que relatam experimentos onde o TDD era aplicado, uma das observações dos autores foi que neles não havia uma avaliação da maneira que o TDD foi aplicado, o que pode ter interferido no resultado final dos estudos, a partir desta constatação, surgiu a motivação para desenvolver um relato onde falhas no uso da técnica seria o foco. Existiam poucos trabalhos com esse tipo de abordagem.

Através da observação empírica, Aniche e Gerosa (2010) levantaram alguns dos erros mais comuns cometidos pelos codificadores quando estão inseridos em um ambiente que usa o TDD. A partir daí, gerou-se uma pesquisa que foi enviada para uma série de codificadores e com base em suas respostas, foi possível verificar se os erros eram realmente recorrentes.

A lista a seguir traz os erros cometidos durante a aplicação do TDD considerados mais recorrentes por Aniche e Gerosa (2010).:

1. Não verificar a falha do teste,
2. Esquecer-se da refatoração,
3. Refatorar outra parte do código enquanto está se trabalhando em um teste,

4. Uso de nomes ruins para os testes,
5. Não começar por testes mais simples,
6. Executar somente o teste que está falhando,
7. Escrever testes muito complexos,
8. Não refatorar o código de teste,
9. Não implementar código o mais simples possível para fazer os testes passar.

5.3 Boas Práticas Para o TDD

Uma vez obtido o conhecimento levantado pelos dois trabalhos citados na seção anterior, é possível observar que, para cada um dos cinco passos do TDD propostos por Kent Back, há alguma possível ação adicional a ser tomada.

Em cada um desses tópicos existem erros cometidos durante a aplicação do TDD, portando há a necessidade de encontrar soluções para contorná-los. Pode-se ter uma lista de boas práticas para ser observada e utilizada quando se tiver aplicando a técnica a fim de potencializar seu resultado final de maneira positiva.

A Tabela 05 traz uma associação entre o TDD propriamente dito e algumas boas práticas adicionais a partir do que foi apresentado pelos dois trabalhos da seção anterior:

Tabela 05 – TDD versus boas práticas adicionais

Passo do TDD	Descrição de boas práticas adicionais
1. Escrever o teste para uma pequena parte da funcionalidade	**Usar nomes descritivos para os testes**: é uma prática comum ler os nomes dos testes antes de implementar uma nova funcionalidade, isto faz com que os codificadores se sintam mais confiantes em relação a tarefa que irão executar em seguida. Como as boas práticas de programação sugerem, o nome de um método deve ser auto-descritivo, isto também se aplica para os métodos desenvolvidos como teste, do contrário será necessário ler o teste para entendê-lo em vez de fazer algo mais importante naquele momento. Bons nomes também fazem dos testes uma documentação aceitável uma vez que cada nome de teste descreve a funcionalidade no sistema (ANICHE e GEROSA, 2010). Suponha que a função do teste é verificar se um cálculo de porcentagem está sendo feito corretamente, um bom nome seria TesteCalculoPorcentagem, um mal exemplo seria teste1, testecalculo, etc.

	Escrever testes simples: além de fazer o código mais simples possível para que o teste passe, o teste também deve ser simples, um caso de teste é escrito para uma pequena parte da funcionalidade, quando é necessário escrever muitas linhas de código para apenas um teste, é sinal de que o trecho que será testado possui muitas responsabilidades e deve ser refatorado (ANICHE e GEROSA, 2010). Sendo assim, o caso de teste deve ser direcionado para o que está sendo avaliado, nem mais nem menos, do contrário o código que o fará passar também não será simples, contradizendo uma das premissas do TDD de código limpo.
	Codificadores devem aprimorar os conhecimentos em testes: uma vez que o TDD é uma técnica de *design* onde o desenvolvedor inicia seu trabalho criando casos de teste e então se escreve o código para que o teste passe, é esperado que ele tenha a habilidade de produzir bons casos de teste. Aparentemente não há investigações explícitas da qualidade dos testes produzidos pelos codificadores em TDD (CALSEVIC, SUNDMARK e PUNNEKKAT et al., 2011).
2. Rode os testes para verificar se há falhas	**Verificar se o teste falhou:** trata-se do segundo passo do TDD, o codificador pode pensar não ser necessário uma vez que ele acabou de escrever o teste e supostamente deve falhar, por este motivo, ele passa direto para o próximo passo que é implementar o mais simples código para que este teste passe. Essa abordagem pode levar o desenvolvedor a erros inesperados, pois uma vez que ele não viu o teste falhar, não pode estar certo de que o novo código o fez passar e a implementação do teste pode estar errada desde o início (ANICHE e GEROSA; 2010).
	Preparação da equipe: o pouco conhecimento sobre o tema pode prejudicar os resultados e até mesmo impedir a adoção da técnica. Codificadores mais experientes e/ou com maior conhecimento apresentam

3. Implemente uma pequena parte do código para que o teste passe	resultados melhores (CALSEVIC, SUNDMARK e PUNNEKKAT et al., 2011). Ministrar treinamentos, promover a programação em pares entre codificadores de diferentes graus de experiência pode melhorar esse quadro.
	Estabelecer um *design* inicial: ainda que o TDD enfatize o uso de pouco planejamento inicial e refatoração constante para manter a arquitetura, a falta desse *design* inicial pode trazer efeitos negativos principalmente em sistemas grandes e complexos. Esta é uma das grandes críticas apontadas para o TDD desde sua aparição (CALSEVIC, SUNDMARK e PUNNEKKAT et al., 2011).
	Utilizar ferramentas adequadas: a prática de TDD requer o uso de uma ferramenta que tenha suporte em forma de *framework* de automação para execução de teste. Ferramentas apropriadas para automação de teste são vitais para o sucesso do TDD (CALSEVIC, SUNDMARK e PUNNEKKAT et al., 2011)
4. Execute os testes e verifique se todos passaram	**Rodar todos os testes sempre que um novo código for implementado:** testes automáticos devem ser executados depois de cada mudança na aplicação para garantir que mudanças não introduziram erros na versão anterior do código. Quando um codificador escreve um teste falho e então o faz passar, ele deve sempre executar todo o conjunto de testes existente, pois o código que ele escreveu pode afetar outras partes do sistema ou fazer com que outro teste deixe de funcionar. Se executar somente o teste em desenvolvimento naquele momento, não será possível verificar se algo já implementado não está mais funcionando. Executar todos os testes somente no final pode requerer mais tempo para encontrar o problema devido à quantidade de linhas de código (ANICHE e GEROSA; 2010).
5. **Refatorar o código**	**Refatorar o código caso necessário:** refatoração é um processo de melhoria do código existente sem que seu comportamento seja alterado. É o último passo e fundamental no processo de TDD porque o código escrito pelo desenvolvedor no passo anterior, nem sempre é o melhor código limpo como resume Beck (2003) como sendo resultado do TDD ou código limpo que funcione (ANICHE e GEROSA; 2010).
	Refatorar o código de teste: no TDD o codificador precisa ler o código em produção e o código do teste caso algo dê errado, por este motivo, é necessário refatorar também o código do teste para que ele também seja o mais limpo possível (ANICHE e GEROSA; 2010).
	Certifique-se de que todos os testes estão passando antes de refatorar o código: os codificadores devem refatorar o código somente quando todos os testes passarem e nunca quando há algum falhando. Desta forma, ele pode detectar qual parte da refatoração precisa ser modificada para que o teste passe (ANICHE e GEROSA;

	2010).

Para verificar a real importância de cada uma das boas práticas, foi elaborado um questionário e o mesmo foi distribuído em comunidades na Internet compostas de pessoas que têm interesse pela técnica, uma delas proveniente de uma empresa de grande porte onde o TDD é o tema central das discussões, buscando a opinião de profissionais com conhecimento e experiência.

5.4 Pesquisa de Opinião Sobre As Boas práticas

O questionário distribuído é composto de dezenove questões, apresentadas no Apêndice1, todas de múltipla escolha, onde através de uma escala, quem está respondendo, escolhe um valor mais adequado à sua opinião sobre o que está sendo questionado. Um escala qualitativa foi adotada para representar as respostas dos participantes. Essa escala foi composta das seguintes opções: “desprezível”, “significativo”, “importante”, “muito importante”, “imprescindível”.

Para finalizar a pesquisa, foi solicitado que fosse informado o tempo de experiência em TDD da pessoa que estivesse respondendo, assim foram consideradas apenas respostas de profissionais de mercado com no mínimo um mês de experiência na técnica.

Foram obtidas 28 respostas em um período de quinze dias, três delas foram desconsideradas pelo fato dos participantes não terem experiência alguma em TDD, os demais já tinham trabalhado em projetos que fizeram uso da técnica durante períodos de um mês a três anos. Todas as respostas estão disponíveis no Apêndice dois. A Tabela 6 ilustra a relação de cada questão com prática sugerida.

Tabela 06 – Boas Práticas versus Questões

Descrição de boas práticas adicionais	Questões
Usar nomes descritivos para os testes.	Uma boa prática de programação é nomear métodos de classe de maneira auto-descritiva. No TDD, essa prática deve ser observada também para os métodos desenvolvidos como TESTES?
	Adotar nomes auto-descritivos também para métodos de classe destinados a testes pode influenciar o sucesso do projeto como um todo?

Escrever testes simples.	O código produzido em um projeto utilizando TDD deve ser o mais simples e limpo possível; É correto afirmar que esta boa prática também se aplica para o código de TESTE?
	Uma vez que a funcionalidade da classe esteja corretamente testada, a complexidade do código de TESTE pode influenciar no sucesso do TDD?
Codificadores devem aprimorar os conhecimentos em testes.	É preciso aprimorar os conhecimentos dos codificadores sobre como identificar cenários de teste?
	É importante que o codificador saiba que a responsabilidade sobre o teste e o correto funcionamento do código é dele?
Verificar se o teste falhou.	É coerente analisar se houve falha de um teste cujo método a ser testado ainda não tenha sido implementado?
	Qual é a possibilidade da causa dessa falha NÃO estar ligada à funcionalidade que será implementada em seguida?
	Uma vez que o método está implementado e o teste executado sem falhas, é aconselhável analisar se o código de teste está correto?
Preparação da equipe.	A falta de preparação prévia de uma equipe inexperiente em relação à técnica do TDD é um fator de risco para o sucesso do projeto?
	A falta de preparação prévia de uma equipe inexperiente em relação à técnica do TDD é um fator de risco para o sucesso do projeto?
Estabelecer um design inicial.	A literatura sobre a técnica do TDD não sugere a necessidade de um *design* inicial do código a ser implementado. É correto afirmar que um *design* inicial do código antes do início da implementação pode trazer vantagens como, por exemplo, diminuir o volume de refatorações?
	O *design* inicial do código é uma boa prática que pode aumentar a percepção sobre o número de casos que devem ser cobertos nos testes?
Utilizar ferramentas adequadas.	Uma vez que o codificador conheça a técnica de TDD, uma ferramenta com maior suporte ao desenvolvimento orientado a testes é necessária?
Rodar todos os testes sempre que um novo código for implementado.	A cada novo método implementado é importante executar todos os testes existentes para garantir a integridade do código?
	Re-executar todos os testes existentes apenas quando toda a funcionalidade estiver implementada pode

	ser mais demorado e caro?
Refatorar o código caso necessário.	Refatorar o código para que seja o mais limpo possível, é necessário mesmo que o código esteja funcionando?
Refatorar o código de teste.	Refatorar o código de TESTE é importante para a qualidade de um produto desenvolvido com o uso do TDD?
Certifique-se de que todos os testes estão passando antes de refatorar o código.	Refatorar o código do produto apenas quando todos os testes passarem, facilita a detecção da fonte de possíveis falhas introduzidas durante a refatoração?

A seguir cada questão é listada assim como seu resultado:

Questão 1: Uma boa prática de programação é nomear métodos de classe de maneira auto-descritiva. No TDD, essa prática deve ser observada também para os métodos desenvolvidos como TESTES?

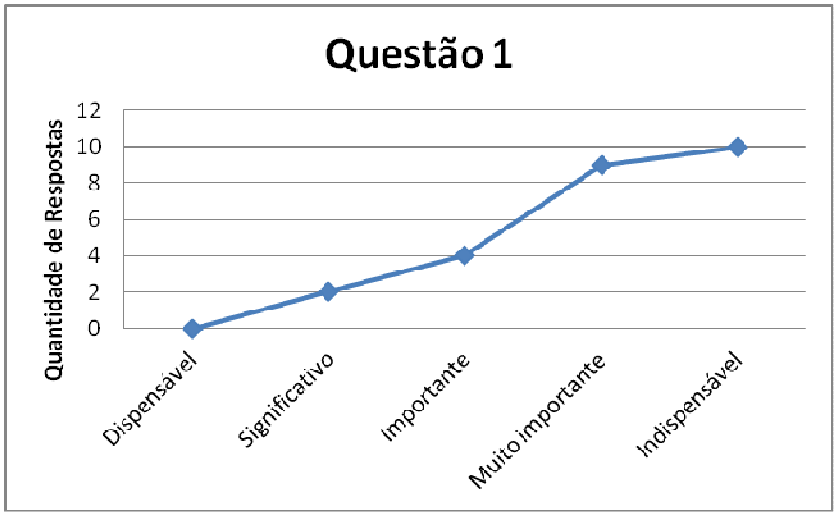

Figura 09 – Resultados da questão 1

Na Figura 9 é possível observar que das 25 repostas válidas, 19 indicam que nomear os métodos de testes de forma auto-descritivas está entre muito importante e imprescindível, conclui-se que se trata de uma prática a ser observada na aplicação do TDD.

Questão 2: Adotar nomes auto-descritivos também para métodos de classe destinados a testes pode influenciar o sucesso do projeto como um todo?

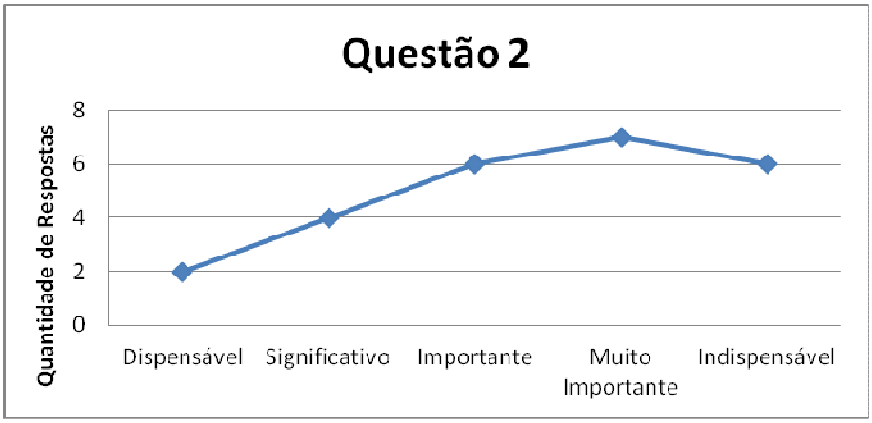

Figura 10 – Resultados da questão 2

Reforçando o resultado da questão anterior, a Figura 10 mostra que treze participantes indicam que adotar nomes auto-descritivos tem grande influência no sucesso de um projeto onde se adota o TDD.

Questão 3: O código produzido em um projeto utilizando TDD deve ser o mais simples e limpo possível; É correto afirmar que esta boa prática também se aplica para o código de TESTE?

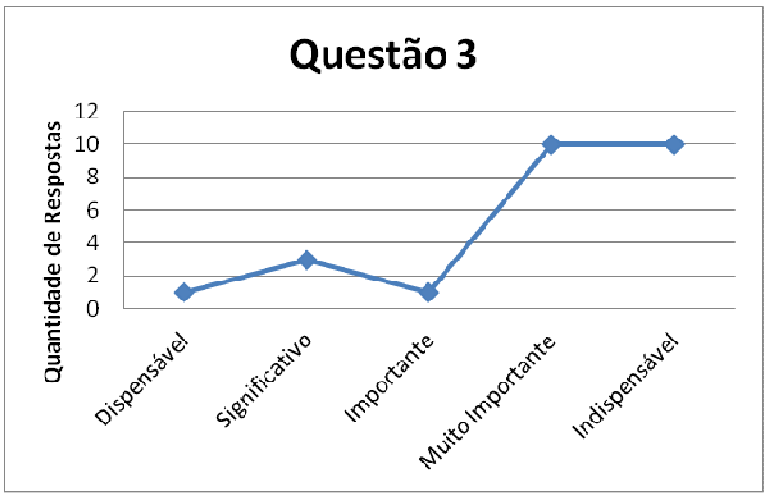

Figura 11 – Resultados da questão 3

Vinte participantes consideram a simplicidade também do código de teste de muito importante a imprescindível, como pode ser observado na Figura 11.

Questão 4: Uma vez que a funcionalidade da classe esteja corretamente testada, a complexidade do código de TESTE pode influenciar no sucesso do TDD?

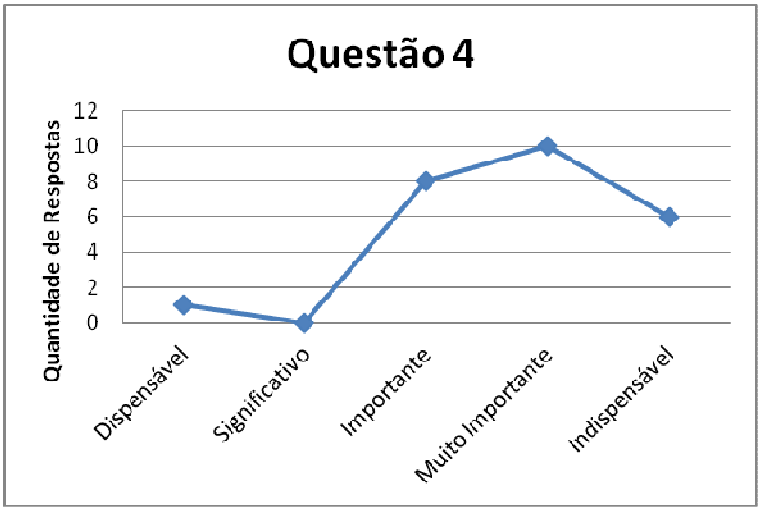

Figura 12 – Resultados da questão 4

Na Figura 12, observa-se que a questão anterior é reforçada pela opinião expressada na questão quatro, que assim como o código de produção, a complexidade do código de teste também tem influência no sucesso do TDD.

Questão 5: É preciso aprimorar os conhecimentos dos codificadores sobre como identificar cenários de teste?

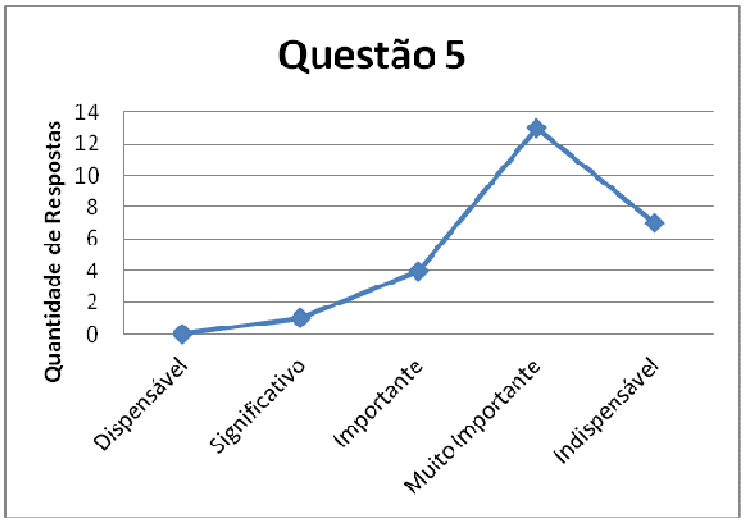

Figura 13 – Resultados da questão 5

Através do resultado desta questão exibido na Figura 13, a falta de

conhecimento no início do projeto confirma-se ser uma preocupação entre a comunidade técnica em relação à aplicação do TDD, mais da metade dos participantes considera os conhecimentos voltados para testes dos codificadores insuficientes.

Questão 6: É importante que o codificador saiba que a responsabilidade sobre o teste e o correto funcionamento do código é dele?

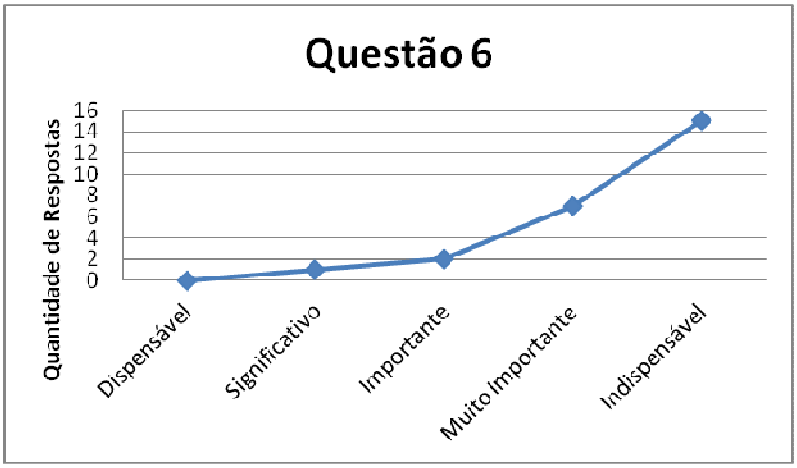

Figura 14 – Resultados da questão 6

Na Figura 14, pode-se observar que os números obtidos nessa questão mostram que é de extrema importância o codificador estar ciente de que a responsabilidade sobre o teste e o correto funcionamento do código é dele, não descartando a presença do testador, que terá sua contribuição não menos importante, verificando, entre outras coisas, se as funções implementadas atendem às necessidades do cliente.

Questão 7: É coerente analisar se houve falha de um teste cujo método a ser testado ainda não tenha sido implementado?

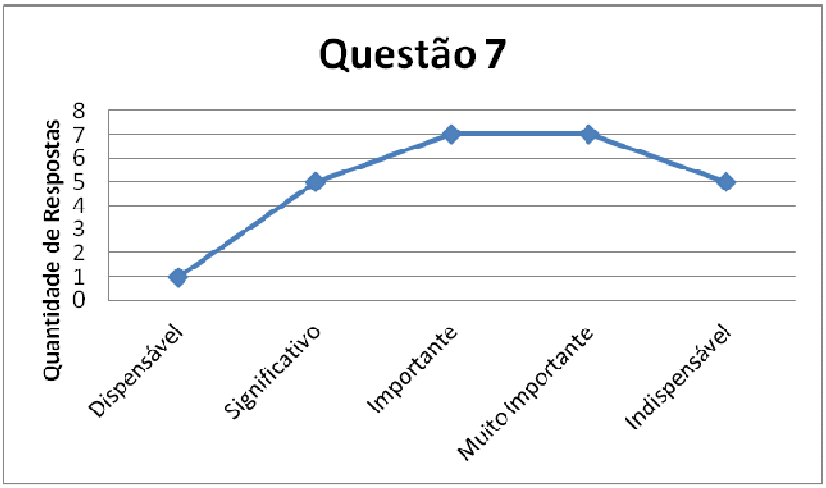

Figura 15 – Resultados da questão 7

A Figura 15 vem mostrar que esta se revelou ser uma prática que divide opiniões entre baixa e alta importância em relação a esta prática. Embora pareça óbvio que um código que testa outro que ainda não existe fatalmente irá falhar, vale ressaltar a possibilidade de o teste estar errado, logo, analisar sua falha é uma maneira de verificar se ele está correto.

Questão 8: Qual é a possibilidade da causa dessa falha NÃO estar ligada à funcionalidade que será implementada em seguida?

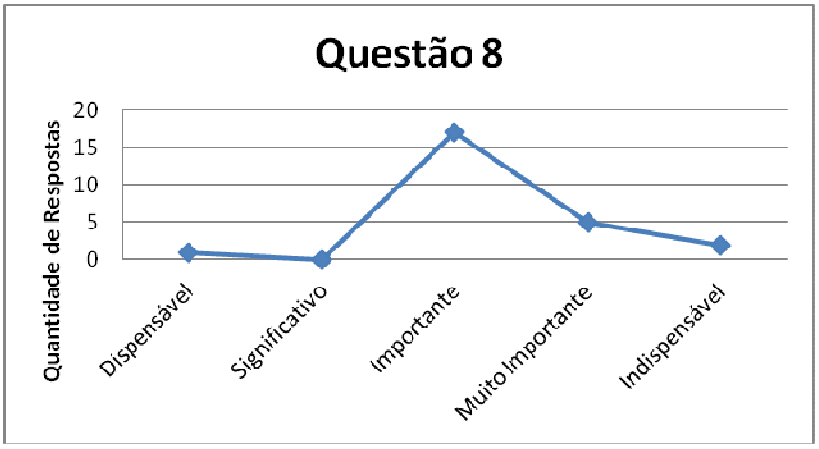

Figura 16 – Resultados da questão 8

Esta questão mostra o motivo da divisão das opiniões da questão anterior, a grande maioria dos participantes considera apenas possível a falha de um teste não estar ligada ao fato de o código a ser testado ainda não tenha sido implementado, mas poucos consideram que certamente este seria o motivo. Essa afirmação é

observada na Figura 16.

Questão 9: Uma vez que o método está implementado e o teste executado sem falhas, é aconselhável analisar se o código de teste está correto?

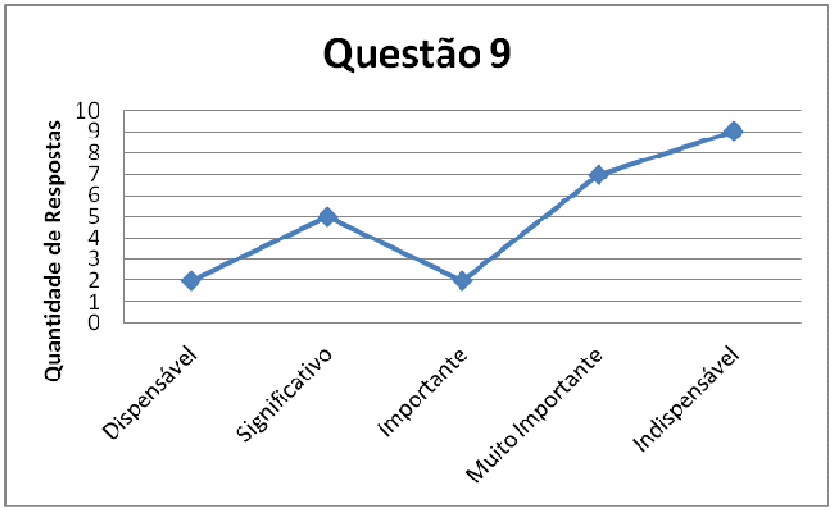

Figura 17 – Resultados da questão 9

Na Figura 17 observa que uma vez que já assumiram a responsabilidade do código do teste, aqui os codificadores reafirmam essa consciência pelo fato de acharem entre muito importante ou imprescindível analisar o código de teste.

Questão 10: A falta de preparação prévia de uma equipe inexperiente em relação à técnica do TDD é um fator de risco para o sucesso do projeto?

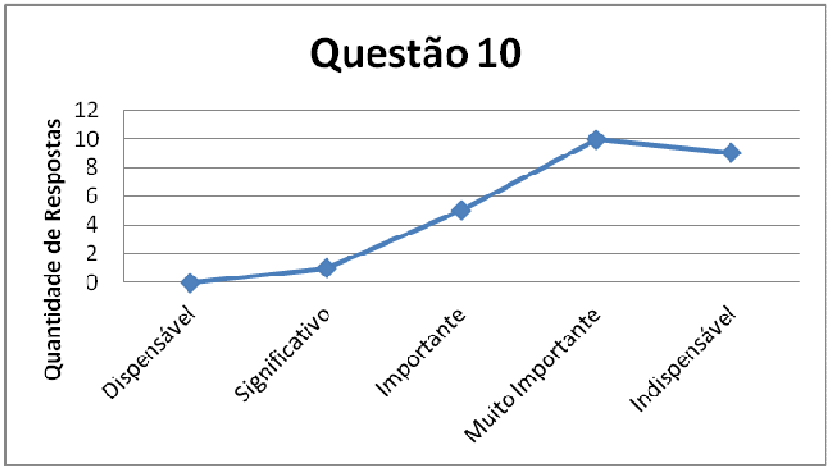

Figura 18 – Resultados da questão 10

A Figura 18 mostra que dezenove respostas entre vinte e cinco mostram a importância da preparação de uma equipe que irá utilizar a técnica de TDD pela primeira vez.

Questão 11: A programação em pares entre um codificador com conhecimento em TDD e outro menos experiente, pode trazer vantagens para o projeto?

Figura 19 – Resultados da questão 11

Com este resultado, é possível afirmar que a programação em pares entre codificadores com conhecimento e inexperientes pode auxiliar no decorrer do projeto, conforme mostrado na Figura 19.

Questão 12: A literatura sobre a técnica do TDD não sugere a necessidade de um *design* inicial do código a ser implementado. É correto afirmar que um *design* inicial do código antes do início da implementação pode trazer vantagens como, por exemplo, diminuir o volume de refatorações?

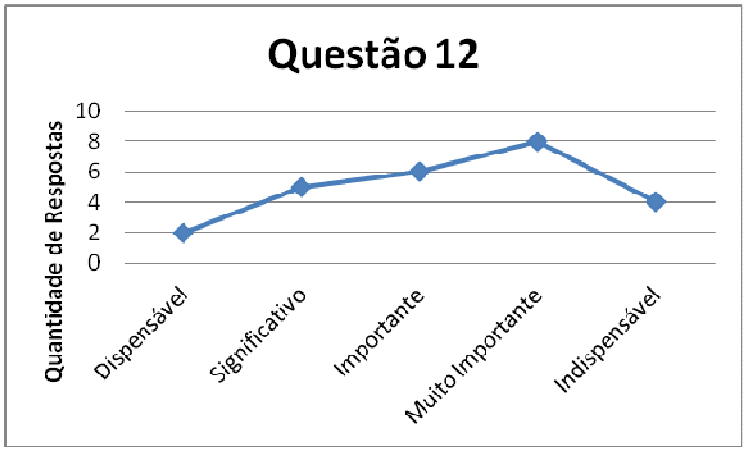

Figura 20 – Resultados da questão 12

Ainda que não seja explicitamente um dos passos do TDD, o *design* inicial do código, antes do início da codificação é uma prática importante para melhor aproveitamento da técnica, isso pode ser observado através das respostas obtidas para esta questão exibidos na Figura 20.

Questão 13: O *design* inicial do código é uma boa prática que pode aumentar a percepção sobre o número de casos que devem ser cobertos nos testes?

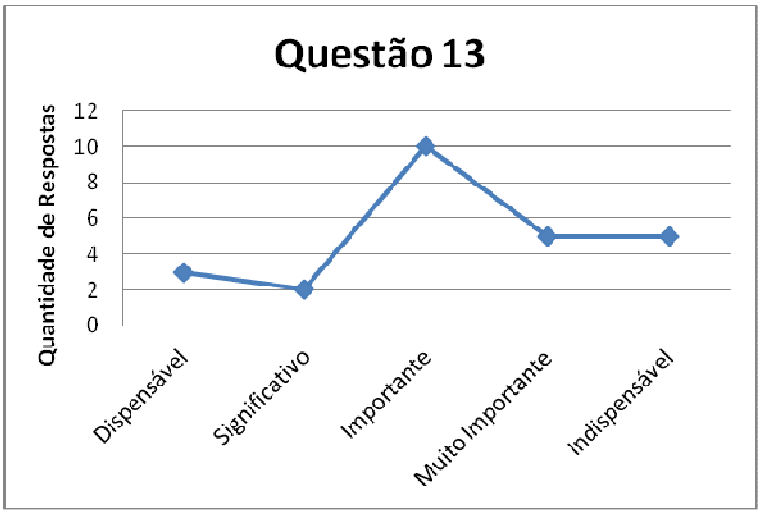

Figura 21 – Resultados da questão 13

Pode ser observado através do resultado dessa questão exibido na Figura 21, que uma das vantagens do uso de um design inicial para o código, é a melhora e aumento dos casos de testes.

Questão 14: Uma vez que o codificador conheça a técnica de TDD, uma

ferramenta com maior suporte ao desenvolvimento orientado a testes é necessária?

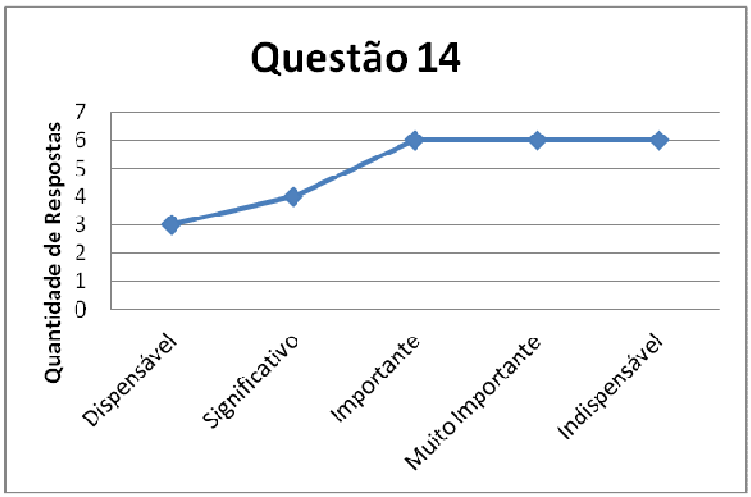

Figura 22 – Resultados da questão 14

A Figura 22 mostra que o uso de uma ferramenta que forneça suporte adequado ao TDD é considerado necessário independente da experiência do codificador, segundo o resultado desta questão.

Questão 15: A cada novo método implementado é importante executar todos os testes existentes para garantir a integridade do código?

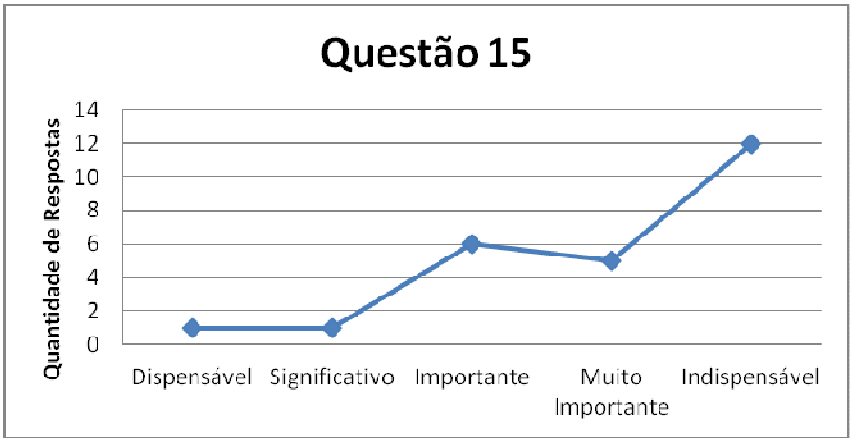

Figura 23 – Resultados da questão 15

O resultado da questão acima vem afirmar que os codificadores têm ciência e concordam, através das respostas positivas, com o que foi analisado em trabalhos anteriores, executar todos os testes a cada novo trecho de código implementado, permite que qualquer erro introduzido seja identificado mais rapidamente e demande

menos esforço para correção, como é observado na Figura 23.

Questão 16: Re-executar todos os testes existentes apenas quando toda a funcionalidade estiver implementada pode ser mais demorado e caro?

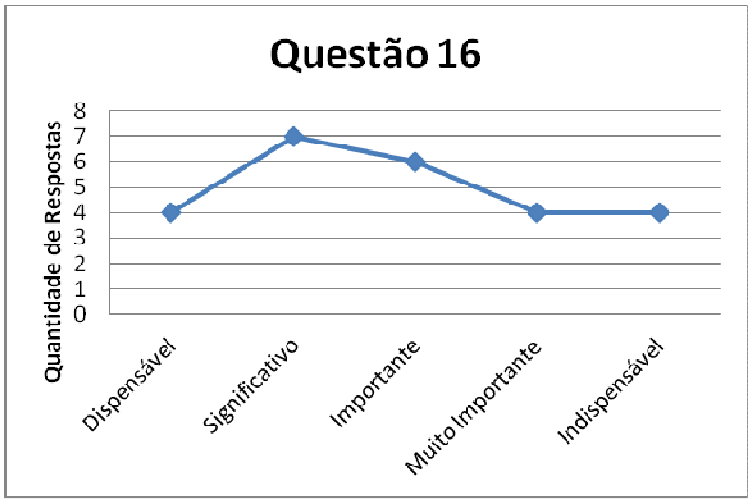

Figura 24 – Resultados da questão 16

Conforme mostrado na Figura 24, a grande maioria das respostas entre desprezível e significativo, vem mostrar que executar todos os testes somente no final da implementação não é necessariamente mais caro, mas vale lembrar de que executar testes a cada parte implementada pode minimizar impactos de falhas encontradas apenas ao final da implementação.

Questão 17: Refatorar o código para que seja o mais limpo possível, é necessário mesmo que o código esteja funcionando?

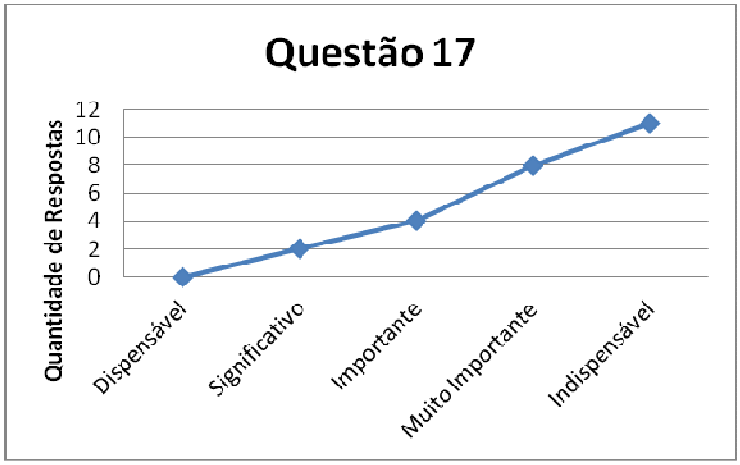

Figura 25 – Resultados da questão 17

Trata-se de um dos passos do TDD pregados por Beck (2003), poucos codificadores discordam conforme mostrado na Figura 25. Um dos objetivos da técnica é que um código limpo e de grande qualidade seja produzido, esta prática auxilia a chegar nesse resultado.

Questão 18: Refatorar o código de TESTE é importante para a qualidade de um produto desenvolvido com o uso do TDD?

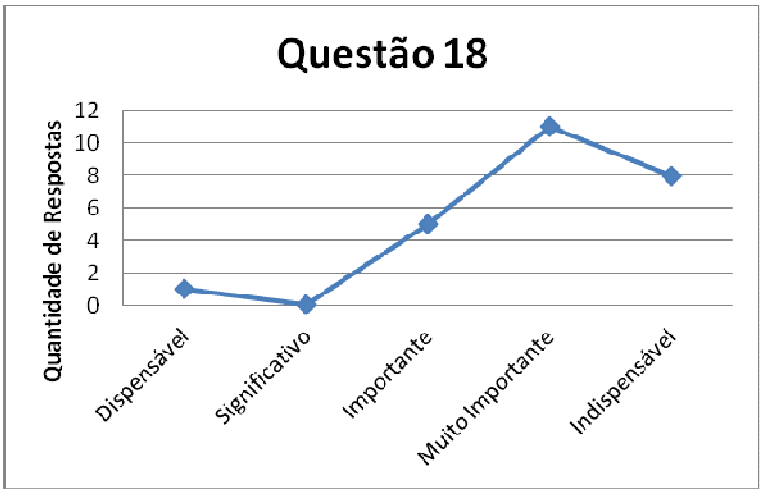

Figura 26 – Resultados da questão 18

A figura 26 mostra que dezenove de vinte e cinco participantes consideram essa prática de muito importante a imprescindível. Uma vez que o código de teste também é produto do desenvolvimento, sua refatoração também é importante para produzir um código de teste com qualidade tão elevada quanto o código de produção, fazendo com que seja mais fácil analisá-lo ou fazer manutenção quando necessário.

Questão 19: Refatorar o código do produto apenas quando todos os testes passarem, facilita a detecção da fonte de possíveis falhas introduzidas durante a refatoração?

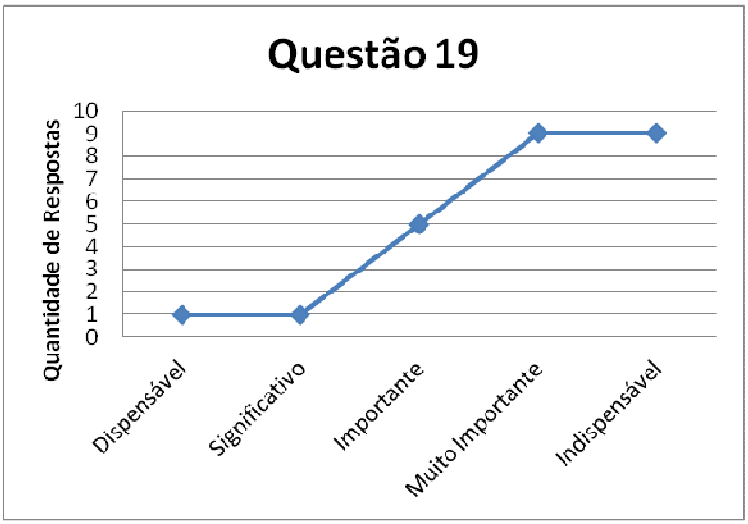

Figura 27 – Resultados da questão 19

Os codificadores acreditam que é mais fácil detectar falhas refatorando o código de produção após todos os seus testes serem implementados conforme é mostrado na Figura 27.

5.5 Considerações do Capítulo

O TDD vem sendo exaustivamente avaliado através de trabalhos acadêmicos onde, na maioria das vezes, fazem o confronto entre esta técnica e métodos tradicionais de teste. Analisando estes trabalhos é possível constatar que raramente a maneira em que se aplica a técnica é descrita, observada ou criticada.

Alguns trabalhos, porém, fizeram um levantamento de pontos a serem observados antes e durante a adoção do TDD em um ambiente de desenvolvimento. Este trabalho focou na junção desses pontos para a elaboração de uma lista de boas práticas a serem acrescentadas aos passos do TDD apresentados por Beck.

Por se tratar de boas práticas derivadas de trabalhos acadêmicos, fez-se necessário avaliar a real necessidade de seguir essas práticas, através da pesquisa previamente descrita.

Houve grande preocupação durante a elaboração das questões para que fossem claras e não induzissem as repostas dos participantes, outro fator importante seria encontrar um meio de atingir o maior número de pessoas que se

enquadrassem ao perfil adequado para participar da pesquisa, sendo este formado de profissionais com pelo menos um mês de experiência com a técnica.

Fóruns de discussão na Internet e de uma grande empresa que produz *software* foi a solução encontrada para distribuir o questionário.

Das repostas obtidas, pode se concluir que as boas práticas sugeridas são consideradas muito importantes para o sucesso da aplicação do TDD, logo é aconselhável observá-las enquanto estiver fazendo uso da técnica.

6. CONSIDERAÇÕES FINAIS

O TDD é uma técnica de desenvolvimento de software que tem como objetivo melhorar a qualidade do código através da escrita de testes anteriores à implementação da funcionalidade propriamente dita e refatorações que buscam, entre outras melhorias, a simplicidade do código final.

O objetivo do presente trabalho foi apresentar e verificar o grau de relevância de boas práticas a serem adicionadas a esta técnica. Para tanto, foi realizada uma pesquisa na literatura onde se verificou a escassez de publicações sobre a maneira como esta técnica é empregada.

Porém, diversas sugestões de melhorias ou críticas à forma que o TDD é utilizado foram analisadas de diferentes fontes e finalmente foi possível criar um conjunto de boas práticas contendo passos adicionais aos apresentados originalmente pela teoria do TDD, segundo Beck (2003).

Foi pré-suposto que a aplicação dessas boas práticas faria com que o uso do TDD, melhorasse a qualidade do software.

6.1 Contribuição do Trabalho

Dentro dos métodos ágeis conhecidos, o TDD tem sido aplicado com bons resultados. Porém, a aplicação dos métodos ágeis exige disciplina e boa observância das práticas prescritas.

Constatou-se na literatura que existem dificuldades de aplicação e entendimento das práticas do TDD que conduz, em muitos casos, a prejuízos na qualidade do software.

Baseado nisto, foi criada um conjunto de boas práticas que poderiam minimizar este cenário. Ao ser submetida a profissionais de software, foi constatado que a grande maioria concorda com a relevância das boas práticas listadas.

6.2Trabalhos Futuros

Devido ao tempo escasso não foi possível fazer uma avaliação mais rigorosa das práticas apresentadas. As conclusões acima foram tiradas a partir de vinte e cinco respostas a um questionário divulgado através da Internet.

Recomenda-se a condução de experimentos que confrontem a aplicação do TDD com as boas práticas em relação aos passos originais do TDD, tenha respostas mais conclusivas e confiáveis.

Outro trabalho seria uma extensão análoga para outros métodos ágeis tais como XP e SCRUM.

REFERÊNCIAS

Agile Aliance. **What is Agile?** Disponível em: http://www.agilealliance.org/the-alliance/what-is-agile/. Acessado em 13/09/2011.

ANICHE, M. F.; GEROSA, M. A.. **Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers.** In: *Software* Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference, May, 2010, p. 469-478.

ARRUDA, S. M. P.. **O que é TDD**. Disponível em http://pt.wikinourau.org/bin/view/GrupoJava/OQueEOTDD. Acessado em 20/11/2011

BECK, K.. **Test-Driven Development: by example.** Addison-Wesley Professional, 2003, p. 240.

BECK, K.; ANDRES, C.. **Extreme Programming Explained: Embrace Change**. Addison-Wesley Professional, 2nd Edition, 2004, p. 224.

BLACK, R.; EVANS, I.; GRAHAM Dorothy; VEENENDALL, Erik van. **FOUNDATIONS OF *SOFTWARE* TESTING - ISTQB CERTIFICATION,** Thomson Learning, 1st edition, 2006. p. 258.

BORGES, E.. N. **Conceitos e Benefícios do Test Driven Development. Disponível em** http://www.inf.ufrgs.br/~cesantin/TDD-Eduardo.pdf. Acessado em 10/11/2011.

BUCHAM, J.; LI, L.; MACDONELL, S. G. **Causal Factors, Benefits and Challenges of Test-Driven Development: Practitioner Perceptions.** In *Software* Engineering Conference (APSEC), 2011 18th Asia Pacific, December, 2011, p. 405 – 413.

CALSEVIC, A.; SUNDMARK, D.; PUNNEKKAT, S.. **Factors Limiting Test-Driven Development Practice: Results from an Online Survey with Developers.** In:

Software Testing, Verification, and Validation Workshops (ICSTW), 2011 Fouth International Conference, March, 2011, p. 337- 346.

CRISPIN, L.; GREGORY J.. **Agile Testing: A Practical Guide for Testers and Agile Teams.** Addison-Wesley Professional, 1st edition, 2009, 576 p..

COHEN, G.. **Agile Excellence for Product Managers: A Guide to Creating Winning Products with Agile Development Teams**. Super Star Press, 1st edition, 2010, 152 p.

FILHO, D. L. B.. **Experiências com desenvolvimento ágil.** Dissertação de Mestrado apresentada na Universidade de São Paulo, São Paulo – Brasil, 2008.

FOWLER, M.. The New Methodology Disponível em: http://martinfowler.com/articles/newMethodology.html. Acessado em 12/09/2011.

GASPARETO, O.. **Test Driven Development**. Instituto de Informática – Universidade Federal do Rio Grande do Sul, Porto Alegre – RS – Brasil, 2005. Disponível em http://www.inf.ufrgs.br/~cesantin/TDD-Otavio.pdf. Acessado 15/09/2011.

GUPTA, Atul; JALOTE, Pankaj. **An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development.** Em: Empirical *Software* Engineering and Measurement, IEEE Software, September, 2007, p. 285 - 294

JANZEN, D. S.; SAIEDIAN, H.. **Does Test-Driven Development Really Improve *Software* Design Quality?** Em: *Software*, IEEE Software, Abril, 2008, p. 77 – 84.

JUNIOR, L. C.. **AQUA - Atividades de Qualidade no Contexto Ágil.** Dissertação de Mestrado apresentada na Universidade Federal de São Carlos, São Carlos – Brasil, 2008.

KELLY, A.. **Changing *Software* Development: Learning to Become Agile.** John Wiley & Sons, 1st Edition, 2008, 258 p.

KETTUNEN, V.; KASURINEN J.; TAIPALE O.; SMOLANDER K. **A Study on Agility and Testing Processes in *Software* Organizations.** Em: International symposium on *Software* testing and analysis, Julho, 2010, p. 231 – 240.

KOLLANUS, S.**Test-Driven Development - Still a Promising Approach?** Em: Quality of Information and Communications Technology (QUATIC), Setembro, 2010, p. 403 – 408.

PEDRINI, J. J. C. **Test-Driven Development e Acceptance Test-Driven Development Técnicas de programação para fazer certo a coisa certa.** Monografia de Graduação. Universidade Estadual de Londrina, Paraná – Brasil, 2009.

PHAN, P-V.; PHAN, A. **Scrum in Action**. Course Technology Ptr, 2011. 284 p.

PRESSMAN, R. ***Software* Engineering: a practioner's approach.** McGraw-Hill, 7th edition, 2009. 928 p.

RIBEIRO, C. L. **A Relação Entre Desenvolvimento Orientado a Testes e Qualidade de *Software*.** Instituto de Informática. Pontifícia Universidade Católica de Goiás (PUC Goiás). Goiânia – GO – Brasil, 2010. Disponível em: www.cpgls.ucg.br. Acessado em 01/09/2011.

SANTOS, M. A.. **AGILE UBPM FOR SCRUM: Modelo de aprimoramento do gerenciamento e desenvolvimento ágil baseado na percepção de valor do usuário**. Monografia de Graduação.Universidade Federal de Lavras, Minas Gerais – Brasil, 2011.

SANTOS, R. L.. **Emprego de Test Driven Development no desenvolvimento de aplicações**. Monografia de Graduação. Universidade de Brasília, Brasília – Brasil, 2010.

SATO, D. T.. **Uso Eficaz de Métricas em Métodos Ágeis.** Dissertação de Mestrado

apresentada na Universidade de São Paulo, São Paulo – Brasil, 2007.

SFETSOS, P.; STAMELOS, I. G. **Empirical Studies on Quality in Agile Practices: A Systematic Literature Review.** In: Seventh International Conference on the Quality of Information and Communications Technology, Outubro, 2010, p. 44-59.

STAMELOS, I. G. **Agile *Software* Development Quality Assurance.** IGI Global, 2007, 268 p.

VICENTE, A. A.; DELAMARO, M. E.; MALDONADO, J. C.. **Uma Revisão Sistemática sobre a atividade de Teste de *Software* no contexto de Métodos Ágeis.** Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo São Carlos - SP – Brasil, 2010. Disponível em: http://andvicente.files.wordpress.com/2009/09/56165_vicente_clei2009_entregadissertacao.pdf. Acessado em 02/11/2011.

VILET, H. ***Software* Engineering: Principles and Practice, Third Edition**. John Wiley & Sons, 3rd Edition, 2008, 740 p.

VU, J. H..; FROJD, N.; SHENKEL-T., C.; JANZEN, D. S.; **Evaluating Test-Driven Development in an Industry-Sponsored Capstone Project.** In: Information Technology: New Generations, April, 2009, 229-234 p.

APÊNDICE 1 – Questionário distribuído para a pesquisa

As questões abaixo visam avaliar o valor de algumas práticas para o sucesso de um projeto no qual a técnica de TDD é aplicada.

Através da escala abaixo, escolha o valor mais adequado de acordo com a sua opinião sobre o que se questiona:

1. dispensável
2. significativo
3. importante
4. muito importante
5. Indispensável

Questão 1: Uma boa prática de programação é nomear métodos de classe de maneira auto-descritiva. No TDD, essa prática deve ser observada também para os métodos desenvolvidos como TESTES?

Questão 2: Adotar nomes auto-descritivos também para métodos de classe destinados a testes pode influenciar o sucesso do projeto como um todo?

Questão 3: O código produzido em um projeto utilizando TDD deve ser o mais simples e limpo possível; É correto afirmar que esta boa prática também se aplica para o código de TESTE?

Questão 4: Uma vez que a funcionalidade da classe esteja corretamente testada, a complexidade do código de TESTE pode influenciar no sucesso do TDD?

Questão 5: É preciso aprimorar os conhecimentos dos codificadores sobre como identificar cenários de teste?

Questão 6: É importante que o codificador saiba que a responsabilidade sobre o teste e o correto funcionamento do código é dele?

Questão 7: É coerente analisar se houve falha de um teste cujo método a ser testado ainda não tenha sido implementado?

Questão 8: Qual é a possibilidade da causa dessa falha NÃO estar ligada à funcionalidade que será implementada em seguida?

Questão 9: Uma vez que o método está implementado e o teste executado sem falhas, é aconselhável analisar se o código de teste está correto?

Questão 10: A falta de preparação prévia de uma equipe inexperiente em relação à técnica do TDD é um fator de risco para o sucesso do projeto?

Questão 11: A programação em pares entre um codificador com conhecimento em TDD e outro menos experiente, pode trazer vantagens para o projeto?

Questão 12: A literatura sobre a técnica do TDD não sugere a necessidade de um *design* inicial do código a ser implementado. É correto afirmar que um *design* inicial do código antes do início da implementação pode trazer vantagens como, por exemplo, diminuir o volume de refatorações?

Questão 13: O design inicial do código é uma boa prática que pode aumentar a percepção sobre o número de casos que devem ser cobertos nos testes?

Questão 14: Uma vez que o codificador conheça a técnica de TDD, uma ferramenta com maior suporte ao desenvolvimento orientado a testes é necessária?

Questão 15: A cada novo método implementado é importante executar todos os testes existentes para garantir a integridade do código?

Questão 16: Re-executar todos os testes existentes apenas quando toda a funcionalidade estiver implementada pode ser mais demorado e caro?

Questão 17: Refatorar o código para que seja o mais limpo possível, é necessário mesmo que o código esteja funcionando?

Questão 18: Refatorar o código de TESTE é importante para a qualidade de um produto desenvolvido com o uso do TDD?

Questão 19: Refatorar o código do produto apenas quando todos os testes passarem, facilita a detecção da fonte de possíveis falhas introduzidas durante a refatoração?

APÊNDICE 2 – RESPOSTAS DA PESQUISA

Nas tabelas seguintes é possível analisar cada resposta enviada da pesquisa publicada na Internet.

Data de Reposta	Participante	Questões					
		1	2	3	4	5	6
1/30/2012 0:37:31	1	4	3	4	4	5	5
1/30/2012 5:58:13	2	5	4	2	4	4	5
1/30/2012 7:39:54	3	2	2	4	3	5	4
1/30/2012 8:28:12	4	4	4	5	5	5	5
1/30/2012 9:06:39	5	5	4	5	4	5	5
1/30/2012 9:14:41	6	3	2	5	3	4	5
1/30/2012 9:20:13	7	3	4	4	5	4	4
1/30/2012 11:12:32	8	4	4	5	4	5	4
1/30/2012 11:41:56	9	4	3	5	5	4	5
1/30/2012 14:12:24	10	5	5	4	4	3	5
1/30/2012 21:50:32	11	3	4	3	3	4	4
1/31/2012 8:07:03	12	3	3	4	4	4	4
2/1/2012 16:19:53	13	5	5	5	5	4	4
2/1/2012 16:42:19	14	4	3	4	3	4	5
2/2/2012 2:01:07	15	1	1	1	1	3	1
2/2/2012 7:38:25	16	5	2	4	4	3	5
2/2/2012 9:20:48	17	5	4	4	3	4	3
2/2/2012 18:19:16	18	4	3	5	4	4	5
2/3/2012 7:56:53	19	4	4	2	5	4	5
2/4/2012 10:36:54	20	4	5	4	3	3	3
2/4/2012 12:48:09	21	5	5	5	4	3	5
2/5/2012 16:21:03	22	4	5	4	3	4	4
2/6/2012 8:49:13	23	5	1	2	1	2	2
2/6/2012 12:27:29	24	4	4	4	4	3	4
2/7/2012 11:36:29	25	3	3	1	3	4	5
2/9/2012 13:02:53	26	2	2	5	3	5	5
2/10/2012 13:13:16	27	5	5	5	5	4	4
2/13/2012 7:13:03	28	5	1	3	4	5	5

Participante	Questões							
	7	8	9	10	11	12	13	14
1	5	3	2	3	4	3	3	4
2	3	4	4	4	5	3	3	4
3	4	4	2	4	2	4	3	3
4	2	4	5	5	5	5	5	4
5	4	3	2	3	5	5	3	3
6	2	3	5	2	4	5	4	3
7	2	3	4	4	5	4	5	3
8	5	5	5	5	3	4	3	5
9	3	3	5	5	4	5	5	5
10	4	4	1	4	5	4	5	5
11	4	2	5	4	4	3	4	3
12	4	4	4	4	4	3	3	3
13	5	3	5	5	3	2	5	5
14	1	1	4	5	4	2	3	5
15	5	5	5	2	1	5	5	5
16	3	3	2	4	3	2	1	2
17	3	3	4	5	3	4	4	2
18	4	3	2	5	5	1	1	1
19	4	3	3	5	5	3	2	1
20	4	3	3	3	5	2	3	3
21	2	3	5	5	1	1	2	4
22	2	3	4	4	2	4	4	5
23	3	3	1	4	2	4	4	2
24	5	3	1	4	3	2	4	1
25	5	5	5	4	5	2	1	1
26	3	3	5	3	5	3	3	4
27	3	3	5	4	5	4	4	4
28	5	3	4	3	4	3	3	2

Participante	Questões						
	15	16	17	18	19	Quantidade de Projetos com TDD que participou	Tempo de experiência com TDD em meses
1	4	2	3	5	2	7	40
2	5	2	5	5	5	2	6
3	3	3	3	3	3	1	5
4	5	5	5	5	4	3	24
5	5	5	4	4	5	2	6
6	5	5	4	4	4	6	5
7	5	2	5	4	4	8	14
8	5	2	4	4	3	1	5
9	5	5	5	3	5	3	10
10	4	4	5	5	4	25	3
11	4	2	3	4	4	0	0
12	4	3	3	3	3	2	2
13	5	2	5	5	4	7	3
14	3	2	2	4	4	3	6
15	1	1	1	1	1	5	0
16	3	4	5	5	5	10	12
17	1	2	4	4	3	3	5
18	3	1	4	4	3	4	40
19	2	1	5	3	4	5	12
20	4	4	4	4	4	1	12
21	3	1	4	4	5	3	9
22	5	3	4	4	5	4	6
23	3	3	2	1	1	3	12
24	4	4	4	4	4	0	n.a.
25	5	1	5	3	5	10	12
26	4	3	3	4	4	3	12
27	5	4	5	5	5	7	2
28	5	3	5	5	5	3	28