Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação

TALES AUGUSTO DE VIEIRA SANTOS

O DESENVOLVIMENTO DIRIGIDO POR TESTES E COMPORTAMENTO

Belo Horizonte
2012

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação
Especialização em Informática: Ênfase: Engenharia de Software

O DESENVOLVIMENTO DIRIGIDO POR TESTES E COMPORTAMENTO

por

TALES AUGUSTO DE VIEIRA SANTOS

Monografia de Final de Curso

Prof. Rodolfo Sérgio Ferreira de Resende
Orientador

Belo Horizonte
2012

TALES AUGUSTO DE VIEIRA SANTOS

O DESENVOLVIMENTO DIRIGIDO POR TESTES E COMPORTAMENTO

Monografia apresentada ao Curso de Especialização em Informática do Departamento de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Especialista em Informática.

Área de concentração: Engenharia de Software

Orientador: Prof. Rodolfo Sérgio Ferreira de Resende

Belo Horizonte
2012

À minha família pelo apoio.

Aos professores desta instituição pelo aprendizado que obtive.

Aos colegas de curso pelo companheirismo e amizade.

À minha namorada pela compreensão e carinho nos momentos dedicados a este trabalho.

AGRADECIMENTOS

Primeiramente a Deus por acreditar em mim nos momentos mais difíceis
Aos meus pais e irmãos pelo apoio nesta complicada jornada.
Aos colegas de curso pelo companheirismo e pela troca de experiências.
E especialmente à minha namorada Janayna, por apoiar-me na conclusão deste trabalho, mesmo que para tal fosse necessário abrir mão de suas horas de descanso.

"A qualidade nunca se obtém por acaso;
ela é sempre o resultado do esforço inteligente."
John Ruskin

RESUMO

Os sistemas estão ficando cada vez mais complexos e com um número cada vez maior de informações a serem processadas e persistidas. Um dos desafios da Engenharia de Software é entregar softwares confiáveis e com alto nível de qualidade. Entretanto, superar este desafio não é uma tarefa fácil, pois a atividade que visa garantir tal nível de qualidade normalmente é comprometida pelo não cumprimento do prazo estipulado para a codificação do sistema. Contudo, existem práticas de desenvolvimento de software que sugerem um novo modelo para o planejamento e execução das atividades de teste. Este trabalho apresenta um estudo sobre algumas destas práticas de desenvolvimento e relatar como estas práticas podem melhorar a qualidade do produto de software.

Palavras-chave: teste, qualidade, tdd, bdd

ABSTRACT

Systems are becoming increasingly complex and an increasing number of information to be processed and persisted. One of the challenges of Software Engineering is to deliver reliable software with high quality. However, to overcome this challenge is not an easy task as the activity that aims to ensure this level of quality is often compromised by the failure of the deadline for the coding system. However, there are software development practices that suggest a new model for the planning and execution of test activities. This paper resents a study on some of these development practices and report how they can improve the quality of the software product.

Keywords: testing, quality, tdd, bdd

LISTA DE FIGURAS

LISTA DE SIGLAS

BDD Desenvolvimento Dirigido por Comportamento (Behavior Driven Development)
GWT Dado que-Quando-Então (Given-When-Then)
TDD Desenvolvimento Dirigido por Testes (Test Driven Development)
XP Programação Extrema (Extreme Programming)

SUMÁRIO

1 INTRODUÇÃO

A qualidade do software deveria ser planejada e acompanhada em todo o ciclo de desenvolvimento. Entretanto, as atividades de desenvolvimento que visam garantir um mínimo de qualidade as vezes são executadas apenas ao final do desenvolvimento do software. Pressman (1987) afirma que qualidade é o grau de excelência que um produto pode atingir, independentemente de seu domínio. Na Engenharia de Software (PRESSMAN, 1995) o caminho para atingir esta excelência não é fácil. Sommerville (2006) fala que a qualidade do produto está diretamente relacionada à qualidade do processo, ou seja, se o processo exige que certos padrões definidos sejam contemplados desde a concepção até a entrega do produto, este software tem grande chance de atingir um nível de qualidade satisfatório. De acordo com a ISO/IEC 9126-3, algumas características são estabelecidas a fim de se avaliar a qualidade do produto de software e, dentre estas características, destacam-se a (i) acurácia, responsável por apurar a produção de resultados ou efeitos corretos e a (ii) conformidade, responsável por checar se o produto está de acordo com as convenções, normas ou os regulamentos estabelecidos.

Os modelos de desenvolvimento de software tradicionais sugerem o planejamento, o acompanhamento e a análise destas características no final do ciclo de desenvolvimento do produto, o que pode ser dispendioso, visto que em sistemas grandes e complexos esta tarefa pode estender o tempo e o custo da fase de desenvolvimento. É comum que as atividades de teste sejam comprometidas pelos possíveis atrasos na codificação resultando num código com baixa cobertura de testes e, consequentemente, num software com a qualidade questionável.

Entretanto, existem no mercado práticas de testes de software que, aliadas a ferramentas de automatização, permitem aos desenvolvedores criar, manter e executar os teste. Pelo fato destes testes serem feitos intercaladamente com o desenvolvimento, eles tendem a ser mais assertivos e capturam falhas do software no momento da implementação. Isto leva os programadores a escreverem seus códigos dirigidos pelos testes, de forma que atendam aos requisitos

previamente definidos e garantindo que o software tenha o comportamento esperado.

1.1 Objetivo geral

Desenvolvimento Dirigido por Testes (Test Driven Development, TDD) é uma prática de teste de software rigorosamente empregada em equipes de desenvolvimento que utilizam a Programação Extrema (Extreme Programming, XP). Entretanto, uma outra prática que tem ganhado a atenção dos praticantes e estudiosos no assunto é o Desenvolvimento Dirigido por Comportamento (Behavior Driven Development, BDD). Este trabalho tem como principal objetivo estudar estas duas práticas de teste de software e identificar os pontos fortes de cada uma.

1.2 Objetivos específicos

1. Apresentar o TDD;
2. Descrever o ciclo de desenvolvimento envolvendo o TDD;
3. Apresentar o conceito do BDD;
4. Descrever as características do BDD;
5. Apresentar os benefícios destas duas práticas de desenvolvimento de software

2 DESENVOLVIMENTO DIRIGIDO POR TESTES

2.1 Introdução

O TDD é um conjunto de práticas de desenvolvimento de software que visa a criação de um código simples e funcional. Ele baseia-se na premissa de que “se você não consegue escrever o teste para aquilo que está codificando, então você nem deveria pensar em codificar” (CHAPLIN, 2001). Pode-se então observar que nenhum código funcional deve ser implementado sem que haja um teste para o mesmo. Outra característica é o processo de correção do software. Quando um erro é descoberto, casos de testes são adicionados ou modificados (quando necessário) antes de se efetuar a correção no código funcional.

Kent Beck menciona em seu livro (BECK, 2002) a importância de se escrever os testes antes do código funcional. No modo tradicional de desenvolvimento de software, à medida que os desenvolvedores implementam o código funcional, seu estresse aumenta gradativamente motivado pela complexidade do sistema e tempo para produzi-lo. Ao finalizar o código funcional o desenvolvedor possui um estresse a ponto de impedi-lo de escrever os testes para aquele código, tornando o sistema suscetível a falhas e com baixa qualidade.

Também conhecido como programação com teste em primeiro lugar (test-first programming), o TDD possui um ciclo de desenvolvimento que, apesar de simples, requer muita disciplina e maturidade por parte da equipe de desenvolvimento, visto que criar e manter os testes atualizados pode não ser uma tarefa fácil. Kent Beck (2002) detalha as etapas de desenvolvimento a serem executadas para se desenvolver utilizando TDD. Estas etapas, consideradas o “mantra” do TDD (BECK, 2002), são divididas em Vermelho, Verde e Refatoração (Red, Green e Refactoring). Em uma ferramenta proposta por Kent Beck o código não passar em um teste é indicado por uma barra vermelha, quando o código passa no teste a barra fica verde. Cada uma das etapas será descrita nas próximas seções deste trabalho.

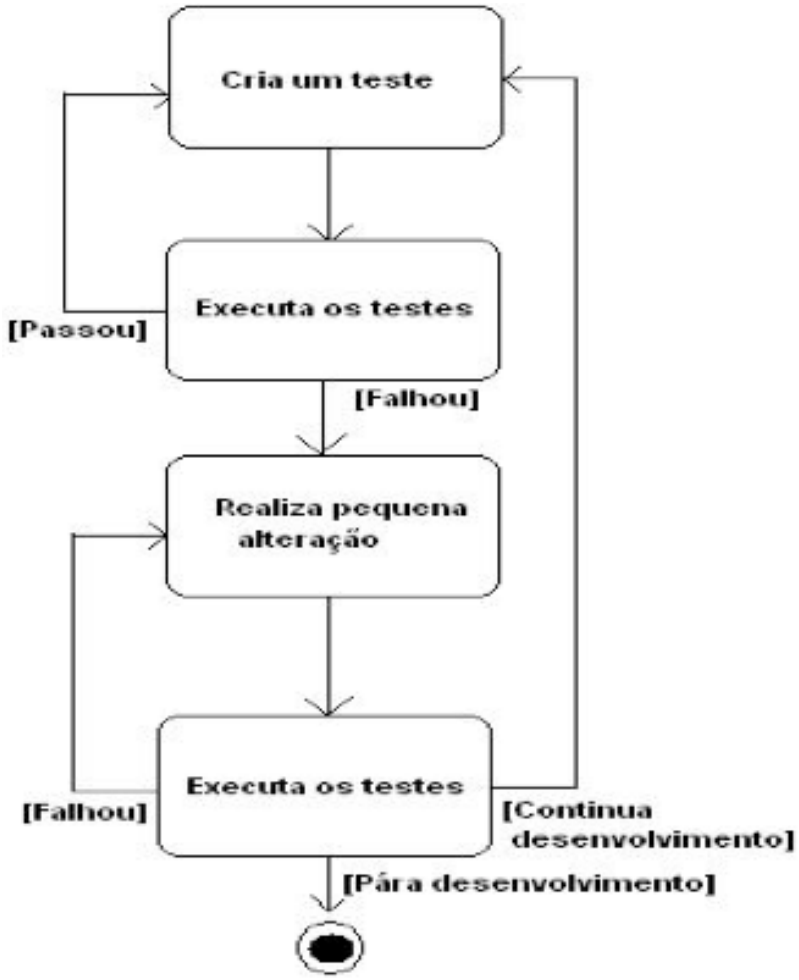

Figura 1 – Ciclo de desenvolvimento com TDD (2003-2006 Scott W. Ambler)

2.2 Ciclo de desenvolvimento do TDD

2.2.1 Criação dos testes - Vermelho

Esta etapa consiste na criação dos testes a partir dos requisitos do sistema. Os insumos necessários para a criação de um novo teste pode ser um caso de uso, histórias de usuário, caso de aceitação ou algum outro artefato que detalhe o funcionamento de uma determinada funcionalidade. Neste ponto é muito importante que o desenvolvedor entenda claramente as especificações, dado que os testes serão escritos baseados apenas nestes requisitos.

Após entender a funcionalidade um teste deve ser escrito para a mesma e, ao ser executado, deve necessariamente falhar ou não compilar, pois não existe um código funcional pronto que o teste passe com sucesso. Este procedimento certifica que seu teste irá executar o código funcional corretamente e que não está funcionando por acidente ou utilizando algum outro código.

Os testes são escritos baseados na real utilização das unidades de código (classes, métodos e funções), ou seja, similar ao código funcional, considerando as interações destas unidades. Em seguida, um esboço do código funcional deve ser escrito apenas para fazer com que o código compile de acordo com o que foi escrito nos testes. Ao ser executado, é esperado que o teste falhe, pois nenhuma implementação foi escrita para que o teste passe com sucesso.

Pode-se então notar que codificando os testes antes da implementação do código funcional o raciocínio do desenvolvedor é canalizado para as regras de negócio de uma determinada funcionalidade, e não para o código funcional propriamente dito. Em outras palavras, é uma forma de se pensar em modelagem antes de se escrever código funcional.

2.2.2 Implementação do código - Verde

Neste ponto tem-se os testes e apenas o esboço do código funcional escritos. Esta etapa do ciclo de desenvolvimento do TDD consiste na implementação do código de produção propriamente dito. Ou seja, será escrito apenas o necessário para que o teste passe com sucesso ao ser executado. Kent Beck (2002) fala que esta implementação deve ser a mais simples possível: *“clean code that works”*.

Uma prática muito importante neste momento é a constante certificação de que as outras partes do sistema continuam funcionando perfeitamente. Para isso, para cada pequeno progresso na implementação do código funcional, todos os outros testes devem ser executados.

É importante salientar também que os testes podem sofrer alterações neste momento com o intuito de cobrir condições, exceções ou laços sem saídas (loops) não previstos na escrita do teste.

2.2.3 Refatoração

Esta parte do ciclo de desenvolvimento utilizando TDD será abordada de forma mais profunda, pois este assunto, cada vez mais, é objeto de estudos da área de engenharia de software e, portanto, merece uma abordagem mais incisiva neste trabalho.

Normalmente os sistemas tendem a ser grandes e complexos, requerendo possíveis correções e melhorias contínuas. Estas manutenções podem eventualmente diminuir a legibilidade do código, tornando-o mais complexo e de difícil manutenção. A refatoração, ou *refactoring*, é uma prática de desenvolvimento de software que visa, dentre outros objetivos, melhorar o grau de entendimento do código, tornando-o manutenível (LEHMAN e BELADY, 1985).

Ward Cunningham e Kent Beck foram as duas primeiras pessoas que reconheceram a importância da refatoração através do projeto *Smalltalk. Smalltalk* é um ambiente de desenvolvimento dinâmico que permite a escrita de softwares altamente funcionais e rapidamente, pois seu ciclo de compilação-execução é pequeno. Ward e Kent trabalharam no desenvolvimento de um processo de desenvolvimento de software capaz de se adequar em ambientes como este. Kent se refere a este estilo de desenvolvimento de Programação Extrema ou *Extreme Programming* (SOURCE, MAKING;).

Martin Fowler (1999) define o processo de refatoração como um conjunto de pequenas alterações, onde cada uma delas visa a melhoria de aspectos estruturais do sistema mantendo-se o comportamento externo do mesmo sem alterações. Contudo, a refatoração de um sistema ou parte dele depende de uma boa análise de viabilidade para reduzir custos excessivos e efeitos colaterais no sistema. As necessidades da refatoração também devem ser levantadas nesta análise, e algumas são listadas por Martin Fowler (1999), tais como:

- prover melhorias arquiteturais *(design)* do software tornando-o manutenível;
- tornar o software fácil de ser entendido (escrever para pessoas e não para compiladores).
- ajudar na depuração a fim de encontrar as falhas com facilidade.

Levantadas as necessidades de se fazer a refatoração no sistema, deve-se então considerar o melhor momento para fazê-lo. Martin Fowler (1999) destaca algumas situações que sugerem o momento propício para a refatoração:

- ao adicionar novas funcionalidades até que o código seja entendido;
- ao encontrar falhas no sistema;
- ao efetuar revisões no código (inspeção).

Entretanto, o processo de refatoração pode ocasionar em alguns problemas em sua aplicação. Martin Fowler (1999) menciona que sistemas nos quais existe um alto acoplamento entre a aplicação e o bancos de dados pode trazer dificuldades no processo de refatoração. Outro ponto a ser considerado neste processo são as interfaces. As interfaces têm por finalidade esconder a sua implementação e disponibilizar apenas os serviços das mesmas, assim, as alterações em sua implementação pode ser feita seguramente (desde que sua interface não mude). Entretanto, quando existe a necessidade de se refatorar os serviços de uma interface, ocorre o risco de se gerar efeitos colaterais em outras partes do sistema. Neste caso, uma boa cobertura de testes de unidade irá transparecer rapidamente os códigos afetados após a alteração, e caso não se tenha esta cobertura, este será mais um indicativo de que a refatoração possivelmente acarretará em problemas no sistema. Martin Fowler (1999) é enfático ao afirmar que a refatoração deve ser evitada caso o prazo de entrega do software (*deadline*) esteja perto.

O processo de criação dos testes dentro do ciclo de desenvolvimento do TDD eventualmente gera trechos de códigos redundantes e é nesta etapa do ciclo que estes códigos serão refatorados objetivando remover estas redundâncias e, consequentemente, melhorando a qualidade do código. Normalmente o processo de refatoração não deve quebrar os testes de unidade já existente. Todos os testes devem ser executados a cada pequena alteração feita no código funcional bem como nos testes, garantindo-se que nenhuma modificação resultou em efeitos colaterais em outras partes do projeto (WELLS, 2011).

2.3 Considerações

Alguns aspectos devem ser observados quando da adoção do TDD. Kent Beck (2002) diz que os testes devem ser escritos apenas para o código que um determinado desenvolvedor está escrevendo. Ou seja, testes para código de outras pessoas deve ser evitado a menos que se tenha uma boa razão para fazê-lo.

Outro ponto muito importante na escrita do teste é o que realmente deve ser testado. Kent Beck (2002) menciona que estruturas condicionais, estruturas de repetição, operações e polimorfismo são pontos de partida para a escrita dos testes. E como mensurar a qualidade dos testes? Kent Beck (2002) lista alguns indicadores que respondem a esta pergunta:

- se for necessário inicializar um grande conjunto de objetos para fazer uma simples asserção, então isso é um indicador de que os objetos estão muito grandes e devem ser particionados;
- se não for possível localizar facilmente um lugar comum para o código de inicialização, então existem muitos objetos fortemente acoplados;

2.4 TDD e os testes tradicionais

Os métodos de desenvolvimento tradicionais planejam e executam as atividades de testes e integração do software no final do ciclo do desenvolvimento (Figura 2). Sistemas grandes e complexos requerem um processo de validação e verificação eficiente já que todo o software deverá ser testado em uma única etapa no processo. Entretanto, na maioria dos casos, a fase de testes é reduzida ou até mesmo desconsiderada, já que a fase de codificação ultrapassa o tempo planejado para ele (CRISPIN e GREGORY, 2009).

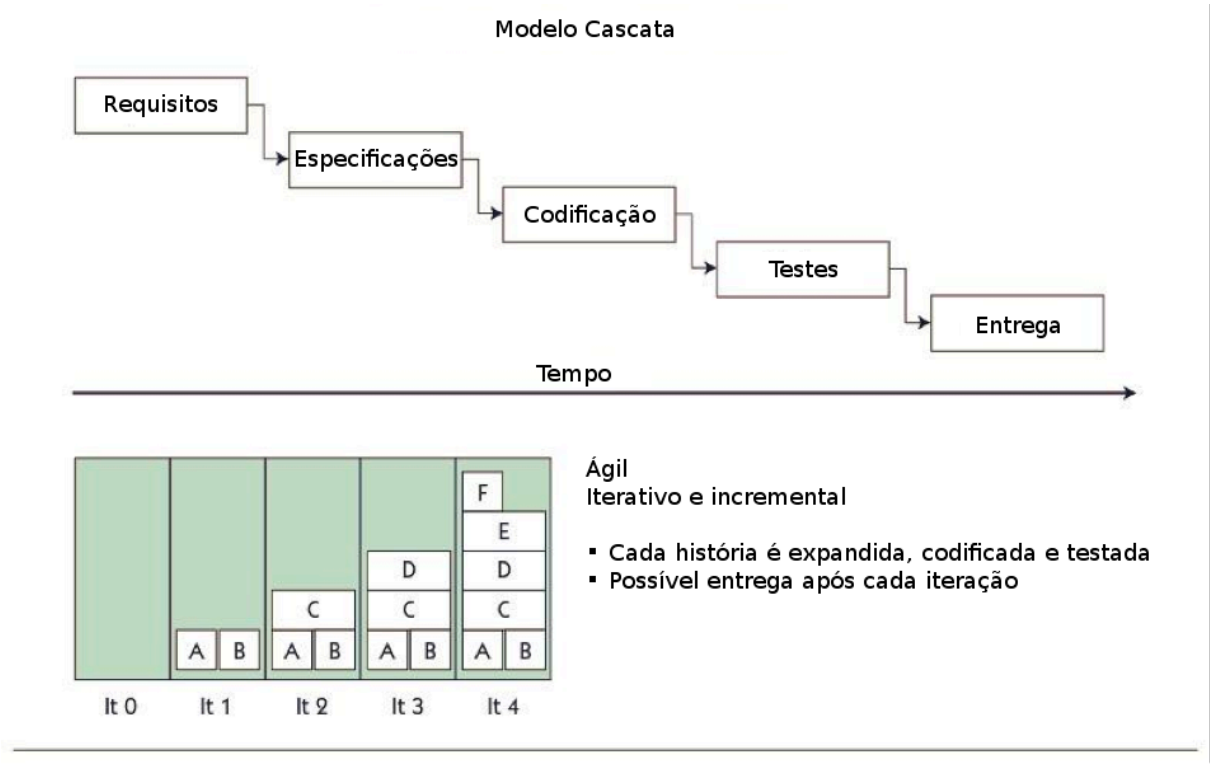

Figura 2 – Teste tradicional versus Teste ágil (CRISPIN e GREGORY, 2009).

É sabido que 40% dos custos do software é destinado às atividades de teste e integração. Em alguns casos este número pode exceder o custo da fase de desenvolvimento (SOMMERVILLE, 2004). A figura 3 mostra os custos por atividade dos modelos tradicionais.

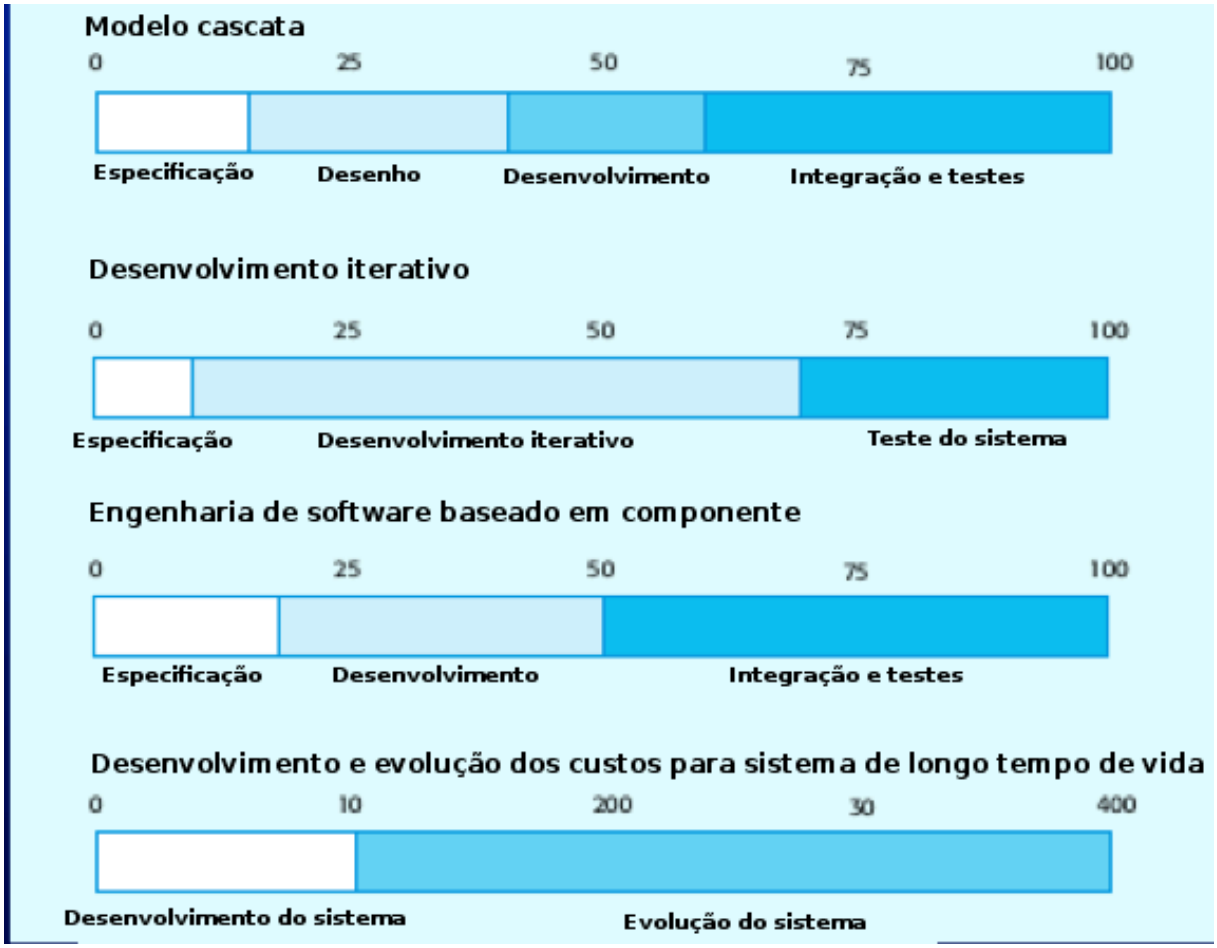

Figura 3 - Custos por atividade (SOMMERVILLE, 2004)

Este cenário motivou Muller e Hagner (2002) a conduzir uma pesquisa comparativa entre o teste tradicional com o modelo TDD. A comparação foi feita utilizando dois grupos de desenvolvedores que executaram as tarefas propostas na pesquisa.

O estudos permitiram mensurar alguns aspectos importantes da Engenharia de Software como tempo de resolução do problema, confiança dos resultados produzidos, reuso de código e qualidade dos testes. Inicialmente foi concedido às duas equipes as especificações das tarefas e algumas classes com métodos apenas declarados e documentados, cabendo aos desenvolvedores implementá-los no decorrer da pesquisa. Ao final do experimento foi possível observar que:

- o desenvolvimento utilizando TDD não foi necessariamente mais rápido que o desenvolvimento tradicional, isto é, a solução para o mesmo problema não foi atingida mais rápida;
- a reutilização de códigos escritos pelo grupo que utilizou TDD foi mais assertivo;

Assim, chegou-se a conclusão de que apesar de o tempo de desenvolvimento dos métodos de ambas equipes foi o mesmo, o código escrito pelo grupo que utilizou o TDD teve menos erros significantes quando reusado. Os pesquisadores, portanto, puderam concluir que o modelo TDD aumenta significativamente a qualidade do software e proporciona maior compreensão do código.

Pesquisas também foram feitas objetivando mensurar o retorno dos investimentos quando utilizado o TDD no desenvolvimento de sistemas. Muller e Padberg (2003) chegaram a alguns resultados utilizando um modelo econômico. Os pesquisadores, então, chegaram nas seguintes conclusões:

- o cálculo do retorno do investimento utilizando TDD independe do tamanho do projeto, do número de desenvolvedores e seus respectivos salários;
- o retorno do investimento depende de um desenvolvimento mais lento e uma alta qualidade do código;
- o esforço para corrigir uma linha de código, ou a produtividade de um desenvolvedor utilizando o processo de desenvolvimento convencional, tem pouco impacto no retorno do investimento do TDD.

2.5 Benefícios

A prática TDD gera vários benefícios quando utilizado adequadamente. A pequena granularidade do ciclo TDD fornece ao desenvolver uma retroalimentação (feedback) contínua, ou seja, as falhas são encontradas rapidamente assim que novas funcionalidades são adicionadas ou alteradas, diminuindo-se consideravelmente o tempo de depuração (BECK, 2002).

As pesquisas relacionadas indicaram que, apesar de o tempo de desenvolvimento utilizando TDD ser igual ao tempo de desenvolvimento utilizando métodos tradicionais, a quantidade de erros significantes é menor quando adotadas as práticas do TDD (MULLER e HAGNER, 2002). Muller e Hagner também destacam que a qualidade final do código é melhor do que os modelos tradicionais, pois as falhas são capturadas e corrigidas ao longo do desenvolvimento e não no final do ciclo do desenvolvimento.

Devido ao fato dos testes serem escritos antes do código funcional, os programadores tendem a escrever mais testes de unidade, ou seja, quanto maior a cobertura dos testes, mais seguro e confiável será o software (BECK, 1999). Os programadores tendem a entender melhor os requisitos, dado que o seu foco inicial é escrever apenas os testes para o mesmo (ERDOGMUS, MORISIO e TORCHIANO, 2005).

3 DESENVOLVIMENTO DIRIGIDO POR COMPORTAMENTO

3.1 Introdução

Desenvolvimento Dirigido por Comportamento (Behaviour Driven Development, BDD), assim como TDD, é uma prática utilizada nos processos ágeis que tem ganhado a atenção dos pesquisadores e praticantes nos últimos anos (MULLER e PADBERG, 2003). Criado originalmente por Dan North (2006), BDD foi uma resposta a algumas falhas encontradas nas práticas do TDD, tais como:

- onde iniciar o processo;
- o que testar e o que não testar;
- quanto de teste ainda falta;
- que nome dar aos testes;
- como entender o porquê um teste falha

Muller e Padberg (2003) ainda acrescenta que o TDD possui uma maior preocupação com o estado das unidades do sistema do que o comportamento desejado do mesmo, e que o código de teste é altamente acoplado à implementação dos sistemas. O modelo do TDD bem como as ferramentas que o suportam, força os desenvolvedores a pensar em termos de testes e asserções ao invés de especificações.

BDD foca no comportamento do sistema, certificando que tudo aquilo que foi especificado na história de usuário está implementado no código fonte. Os requisitos possuem uma descrição textual com palavras-chaves indicando como o sistema deve se comportar dado um contexto, um evento e uma ação (CARVALHO, SILVA e MANHAES, 2010). Assim, os especialistas de negócio comunicam-se com a equipe de desenvolvimento utilizando uma mesma linguagem, evitando-se possíveis erros de compreensão e interpretação dos requisitos do sistema. O principal objetivo do BDD é obter requisitos executáveis do sistema (SÓLIS e WANG, 2011).

BDD propõe o uso de uma linguagem ubíqua para que os negócios e as pessoas de tecnologia se refiram ao sistema em uma mesma maneira. BDD inicia com a modelagem dos requisitos do sistema em forma textual utilizando uma linguagem ubíqua onde os requisitos são descritos utilizando modelo de histórias de usuário (CARVALHO, SILVA e MANHAES, 2010). O principal objetivo das histórias de usuário é explicitar o valor do negócio de cada parte do sistema àqueles que irão desenvolver ou se envolver de alguma forma na construção do software. Criadas as histórias de usuário, um conjunto de possíveis roteiros deve ser escrito descrevendo-os no formato *Dado que-Quando-Então* (Given-When-Then, GWT) (CARVALHO, SILVA e MANHAES, 2010).

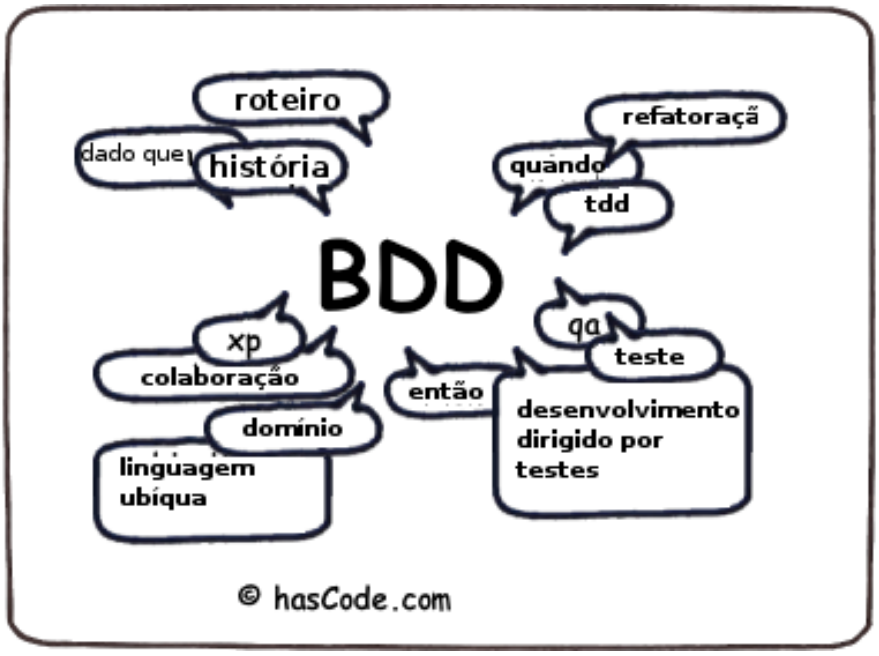

Figura 4 - Características do BDD
Fonte: http://hascode.com

3.2 Linguagem ubíqua

Os especialistas de negócio possuem um entendimento tecnicamente limitado sobre as terminologias utilizadas pelos desenvolvedores no desenvolvimento do sistema. A equipe de desenvolvimento, por outro lado, deve

traduzir as regras de negócio definidas pelos especialistas de negócio, e transformá-la em código (EVANS, 2003). No processo de análise dos requisitos os desenvolvedores podem interpretar os requisitos destorcendo o que foi solicitado inicialmente.

Este problema eventualmente é percebido nas empresas de desenvolvimento de software, onde o resultado final do produto nem sempre é aquilo que realmente foi pedido pelos clientes por causa de falhas na comunicação com os fornecedores.

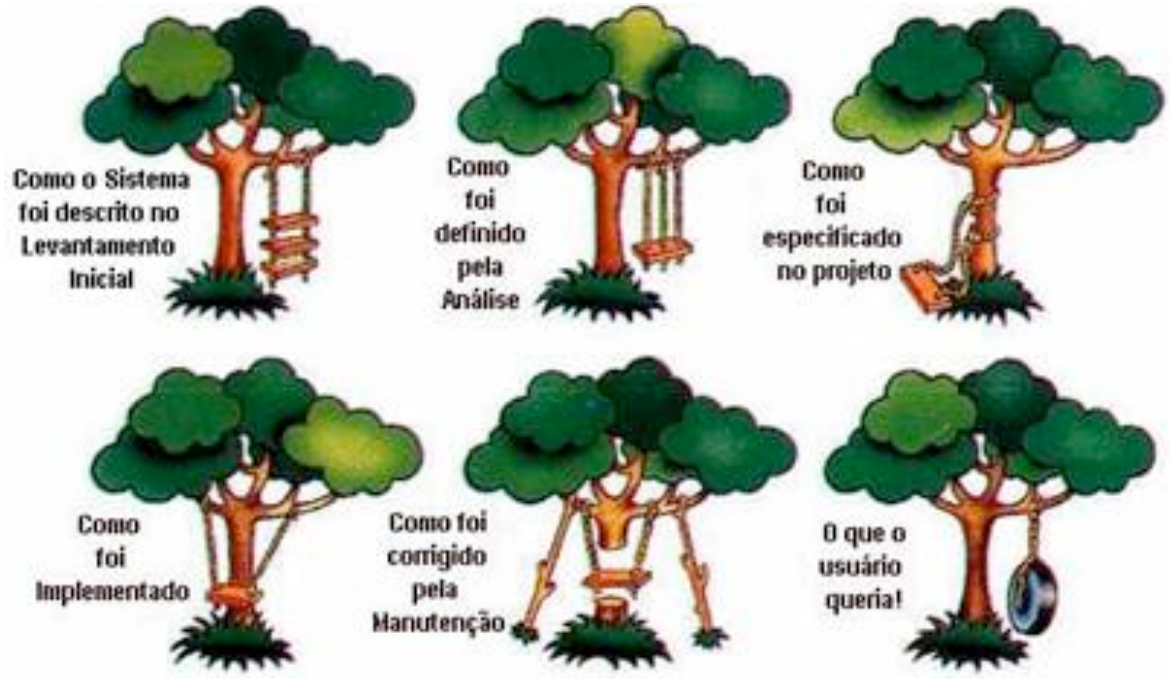

Figura 5 – Diferentes interpretações ao longo do ciclo de desenvolvimento de um software.

Fonte: http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035

Segundo Evans (EVANS, 2003), uma linguagem comum deve ser utilizada para que os especialistas de negócio descrevam os requisitos de forma que os desenvolvedores entendam sem que haja problemas de interpretação e entendimento daquilo que está sendo solicitado.

A linguagem ubíqua é a uma das principais características do BDD (MULLER e PADBERG, 2003). Ela é estruturada de acordo com o modelo do

domínio na qual contém termos que serão utilizados para definir o comportamento do sistema e permite que especialistas de negócio e os desenvolvedores conversem numa mesma linguagem sem ambiguidades. No BDD, a linguagem ubíqua é formada a partir de um conjunto de palavras pré-determinadas a fim de identificar as regras de negócio. Estas regras são descritas seguindo um modelo estruturado de forma a explicitar o valor do negócio para os usuários e desenvolvedores (NORTH, 2006). Este modelo é denominado histórias de usuário, o qual será melhor descrito na seção 3.3 deste trabalho.

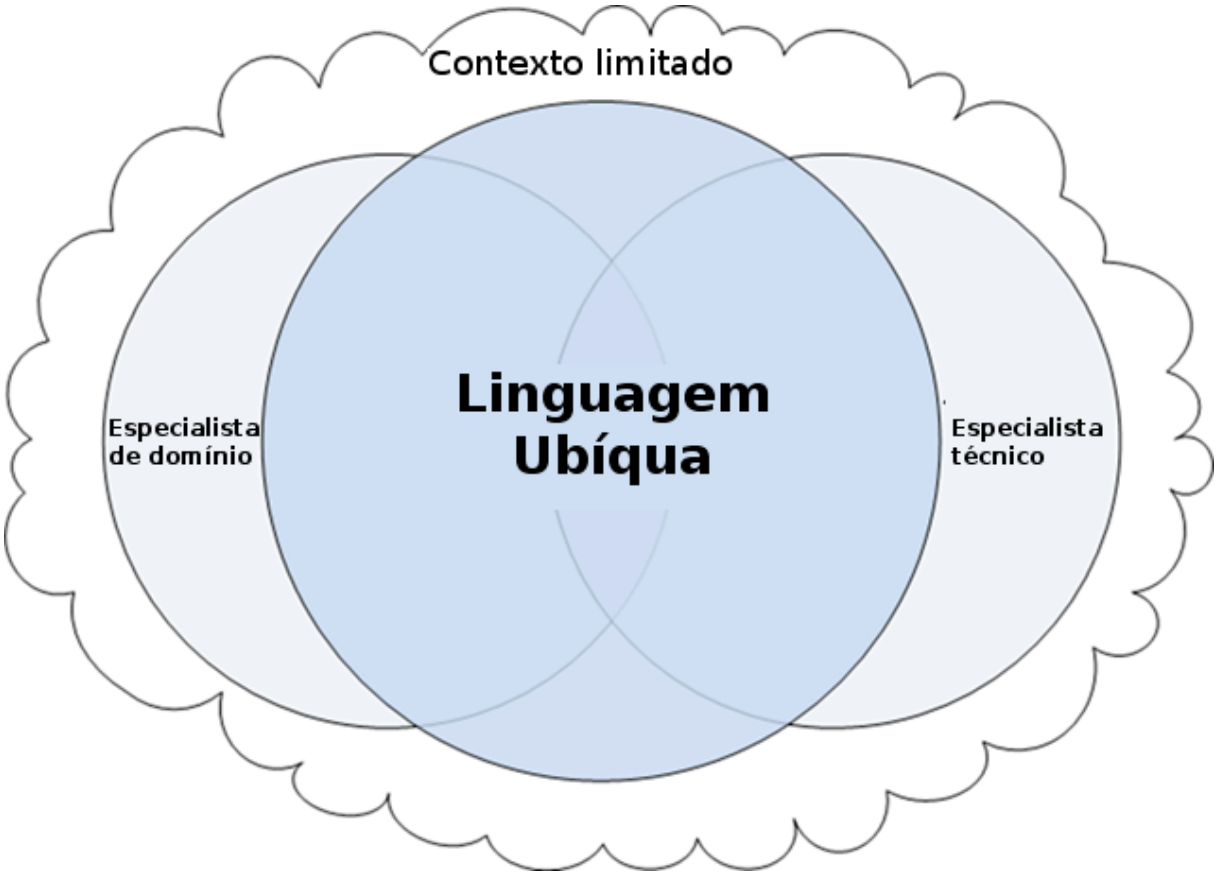

Figura 6 – Intersecção do dialeto do domínio de negócio com desenvolvedores resulta na linguagem ubíqua

Fonte: http://userinexperience.com/?p=308

3.3 Histórias de usuário

Uma interação no processo de desenvolvimento do XP é iniciada pela escrita das histórias de usuário. Elas são criadas pelos especialistas do negócio e passadas para os desenvolvedores iniciarem a codificação dos testes e, posteriormente, o código funcional (BREITMAN e LEITE, 2002). Kent Beck (1999) define as histórias de usuário como “algo que os clientes necessitam que o sistema faça” e ainda afirma que as histórias de usuário “precisam ser estimáveis idealmente entre uma a cinco semanas de desenvolvimento e devem ser testáveis”. Cohn (2004) define uma boa história de usuário utilizando o acrônimo em inglês INVEST: Independente, Negociável, Valiosa, Estimável, Pequena (Small) e Testável.

As histórias são decompostas em tarefas, e para cada tarefa é estimado o esforço e tempo para fazê-la.

No BDD as histórias de usuário fornecem o contexto das funcionalidades entregadas para o sistema. Elas descrevem as interações entre os usuários e o sistema e qual é o valor de negócio que justifique a existência desta funcionalidade. As histórias de usuário são compostas por três partes (SÓLIS e WANG, 2011):

- qual é o papel do usuário que irá interagir na funcionalidade;
- o que o usuário deseja fazer na funcionalidade;
- quais os benefícios o usuário terá se o sistema prover esta funcionalidade.

[Título da história]
Como [papel]
Desejo [funcionalidade]
Porque [benefícios]

A descrição dos termos definidos entre chaves deve ser escrito utilizando a linguagem ubíqua. O título da história descreve uma atividade feita pelo usuário dado um determinado papel. A funcionalidade fornecida pelo sistema permite ao usuário executar a atividade, e logo em seguida, o usuário obtém o benefício da funcionalidade. Este modelo permite identificar claramente o que o sistema deve suportar. É importante observar que este formato exige que os benefícios sejam

explicitados para que todos os envolvidos saibam os reais valores da funcionalidade e ainda analisar se deve ser ou não implementada neste momento (BREITMAN e LEITE, 2002). É uma forma de se priorizar o que deve ser produzido no sistema.

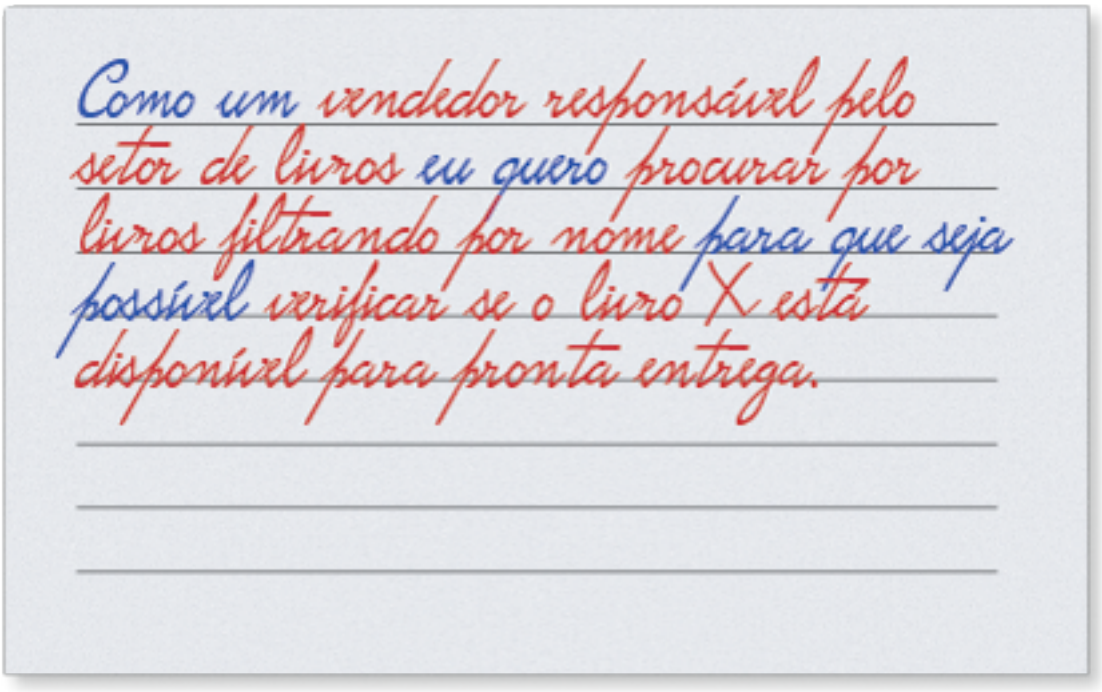

Figura 7 - Exemplo de história de usuário
Fonte: http://blog.scrumhalf.com.br/2011/10/user-stories-o-que-sao-como-usar/

No contexto do BDD cada história de usuário pode derivar em vários roteiros, que por sua vez são usados como critérios de aceitação.

3.4 Roteiros

Roteiros descrevem como o sistema que implementa uma funcionalidade deve se comportar quando está em um determinado estado e um determinado evento acontece. O resultado esperado é uma ação que muda o estado da aplicação ou uma resposta do sistema. Os roteiros podem ser escritos por qualquer pessoa envolvida no sistema, desde que se mantenha o mesmo formato e, o mais importante, o valor do negócio seja bem explicitado. No momento da discussão para

a criação dos roteiros, os envolvidos debatem entre si analisando se o resultado esperado do roteiro está de acordo com os eventos. Isto ajuda a revelar outros roteiros e esclarecer os requisitos.

A vantagem de se escrever os requisitos em forma de roteiros é que estes podem ser automatizados por ferramentas como a XReq (SOGILIS, 2012).

Roteiro 1: [Título do roteiro]
Dado que (Given) [Contexto/Estado]
Quando (When) [Evento]
Então (Then) [Resultado]

O termo “Dado que” define o estado antes que um usuário ou uma ferramenta externa inicie uma interação com o sistema. Nunca deve ser falado de interação do usuário nesta parte do roteiro. O evento descreve a ação chave que o usuário ou sistema irá executar culminando na transição de estado da máquina. Os resultados oriundos desta transição devem ser avaliados para saber se os benefícios/valores da funcionalidade foram atingidos ao executar todos estes passos. Um exemplo que melhor exemplifica o formato de um roteiro pode ser visto assim:

Roteiro 1: Itens reembolsados devem retornar para o estoque
Dado que um cliente compra um jumper preto e eu tenho três jumper pretos no estoque
Quando ele retorna com o jumper preto para reembolso
Então eu devo ter quatro jumpers pretos no estoque

A convenção GWT (Given-When-Then) conecta os conceitos humanos de causa e efeito aos conceitos de software entrada-processamento-saída. Esta convenção simplesmente traduz uma transição de estado. Uma combinação de vários roteiros de uma regra de negócio pode ser representado pela Máquina Finita de Estado, e, consequentemente, em Petri Net (CARVALHO, SILVA e MANHAES, 2010).

Embora exista no mercado vários padrões para modelagem de negócio, eles são formados por construtores básicos. Porém, mapeando estes construtores em notações GWT é possível construir estes padrões utilizando a linguagem ubíqua

do BDD (CARVALHO, SILVA e MANHAES, 2010). As figuras 8 e 9 ilustram como é feito o mapeamento de um simples diagrama de estado em notação GWT.

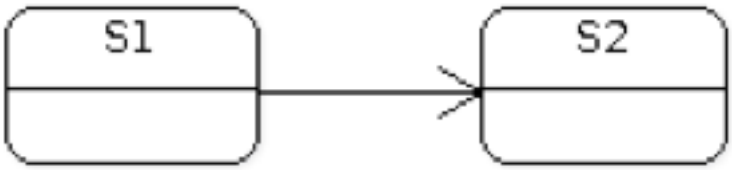

Figura 8 - Transição de estado (CARVALHO, SILVA e MANHAES, 2010)

Figura 9 – Representação GWT da figura 8 (CARVALHO, SILVA e MANHAES, 2010)

A sequência acima define o estado inicial como S1. Após a realização do evento EV1 o sistema passa para o estado S2. Este mapeamento é bem simples pois tem apenas o propósito de exemplificar como é feita esta conversão. Outros exemplos podem ser encontrados nas referências bibliográficas deste trabalho.

3.5 Benefícios

A abordagem BDD ainda é nova na área de desenvolvimento de software por isso não existem muitas pesquisas com a finalidade de se avaliar seus reais benefícios. Contudo, através dos estudos realizados nestes trabalho, pôde-se observar que esta prática de desenvolvimento possui algumas características interessantes.

BDD é uma técnica de desenvolvimento de software que propõe o uso de uma linguagem ubíqua para que negócios e pessoas de tecnologia se refiram ao sistema de uma mesma maneira. Esta linguagem permite que i) clientes

especifiquem os requisitos a partir da visão de negócio, ii) os analistas de negócio traduzam os requisitos em histórias de usuário e roteiros e iii) os desenvolvedores implementem os comportamentos esperados do sistema. Este modelo encoraja a colaboração de todos os participantes (fornecedores, analistas, desenvolvedores e etc) do projeto facilitando a comunicação entre eles (LAZAR, MOTOGNA e PÂRV, 2010).

Assim como o TDD, o modelo do BDD sugere que o código deve fazer parte da documentação do sistema, agregando em sua abordagem aspectos de desenvolvimento ágil. A escrita dos métodos deve indicar a finalidade do método, facilitando a leitura e entendimento do código (SÓLIS e WANG, 2011).

Outra característica do BDD é a automação dos requisitos. A utilização de histórias de usuário e roteiros seguindo um modelo pré-determinado possibilita que ferramentas especializadas leiam e interpretem estes artefatos resultando em trechos de códigos executáveis. Isto permite que testes de aceitação e testes de regressão sejam executados também de forma automática.

4 CONCLUSÃO

A prática de teste de software no final do ciclo de desenvolvimento ainda está presente em muitas empresas de desenvolvimento. Os modelos tradicionais possuem uma parcela de “culpa” neste cenário, pois eles sugerem que as atividades de testes sejam executadas ao final do ciclo de desenvolvimento do produto, dando margem para que possíveis atrasos na fase de codificação comprometam as atividades de teste.

Este cenário foi um dos fatores que motivou o surgimento das metodologias de desenvolvimento ágil, em resposta a estas falhas dos modelos tradicionais. O XP (BECK, 1999) é atualmente o modelo de desenvolvimento ágil que mais tem ganhado destaque no mercado. Ele é tão rigoroso ao ponto de exigir que nenhum código funcional possa ser escrito sem que haja testes para o mesmo.

Também conhecido como programação teste-primeiro, o TDD possui em seu ciclo de desenvolvimento etapas que forçam o desenvolvedor escrever os testes do sistema antes de programar o código funciona. Este procedimento evita que o programador pense apenas em codificação, focando-o inicialmente em desenho (design). Vale observar também que o TDD mantém o código simples e funcional, resultado da técnica de refatoração do código. Quando bem utilizado, o TDD pode agregar valores consideráveis no software, tais como: melhoria na qualidade do código, redução de erros, redução de tempo na captura de falhas e etc.

O BDD é uma técnica de desenvolvimento ágil tendo como principal finalidade a criação de requisitos de negócio executáveis utilizando uma linguagem ubíqua. BDD foca no comportamento do sistema ao posso que o TDD foca no correto funcionamento de pequenas unidades. Esta diferença foi um dos motivos resultou na criação do BDD.

Ambas as práticas apresentadas requerem um forte relacionamento com o time de negócio, assim sendo, o perfil da empresa influenciará consideravelmente na escolha da prática de desenvolvimento a ser adotada. Vale salientar também que estas práticas requerem um nível de conhecimento apreciável por parte do time de desenvolvimento.

Os estudos realizados neste trabalho permitem, então, concluir que ambas as práticas de desenvolvimento analisadas são boas opções para empresas que buscam a qualidade de seus softwares.

5 REFERÊNCIAS

BECK, K. **Extreme Programming Explained:** Embrace Change. US. ed. Boston: Addison-Wesley Professional, 1999.

BECK, K. **Test-Driven Development by Example**. 1a. ed. Boston: Addison-Wesley Professional, 2002.

BOB, U. The Truth About BDD. **Object Mentor**, 11 Nov 2008. Disponivel em: <http://blog.objectmentor.com/articles/2008/11/27/the-truth-about-bdd>. Acesso em: 15 Maio 2012.

BREITMAN, K.; LEITE, J. Managing User Stories. **PUC-Rio**, 2002. Disponivel em: <www-di.inf.puc-rio.br/~julio/tcre-site/p6.pdf>. Acesso em: 23 Maio 2012.

CARVALHO, R.; SILVA, F.; MANHAES, R. Cornell University Library. **Mapping Business Process Modeling constructs to Behavior Driven Development Ubiquitous Language**, 25 Jun 2010. Disponivel em: <http://arxiv.org/abs/1006.4892>. Acesso em: 13 Dez 2011.

CHAPLIN, D. **Test First Programming**. TeckZone. [S.l.]: [s.n.]. 2001.

COHN, M. **User Stories Applied:** For Agile Software Development. 1a. ed. Boston: Addison-Wesley Professional, 2004.

COIN, K. Behavior Driven Development. **Wikipédia**, 21 Outubro 2005. Disponivel em: <http://pt.wikipedia.org/wiki/Behavior_Driven_Development>. Acesso em: 28 Maio 2012.

CRISPIN, L.; GREGORY, J. **Agile Testing:** A Practical Guide for Testers and Agile Teams. 1a Edição. ed. [S.l.]: Addison-Wesley Professional, 2009.

ERDOGMUS, H.; MORISIO, M.; TORCHIANO, M. On the Effectiveness of the Test-First Approach to Programming. **IEEE Transactions on Software Engineering**, 2005.

EVANS, E. **Domain-Driven Design:** Tackling Complexity in the Heart of Software. 1a. ed. [S.l.]: Addison-Wesley, 2003.

FOWLER, M. **Refactoring – Improving the style of existing code**. 1a. ed. [S.l.]: Addison-Wesley, 1999.

GELPERIN, D.; HETZEL, W. **Software Quality Engineering**. Fourth International Conference on Software Testing. Washington: [s.n.]. 1987.

ISO/IEC 9126-3. **Software Engineering - Product Quality**. [S.l.]. 2003.

LAZAR, I.; MOTOGNA, S.; PÂRV, B. Behaviour-Driven Development of Foundational UML Components. **Electronic Notes in Theoretical Computer Science**, p. 91-105, 10 Agosto 2010.

LEHMAN, M.; BELADY, L. **Program Evolution:** Processes of Software Change. 1a Edição. ed. [S.l.]: Academic Press, 1985.

MULLER, M.; HAGNER, O. Experiment bout test-first programming, v. 149, n. 5, p. 131-136, Outubro 2002.

MULLER, M.; PADBERG, F. **About the Return on Investment of Test-Driven Development**. 5 International Workshop on Economic-Driven Software Engineering Research. Portland: International Conference on Software Engineering. 2003. p. 23-31.

NORT, D. Introducing BDD. **DanNorth.net**. Disponivel em: <http://dannorth.net/introducing-bdd/>. Acesso em: 21 Maio 2012.

NORTH, D. Behavior Driven Development. **Behavior Driven Development**, 2006. Disponivel em: <http://behavior-driven.org>. Acesso em: 20 Maio 2012.

PRESSMAN, R. **Engenharia de software:** a practioner's approach. Nova Iork: Macqraw, 1987.

PRESSMAN, R. **Engenharia de Software**. 6a Edição. ed. São Paulo: Makron Books, 1995.

SÓLIS, C.; WANG, X. **A Study of the Characteristics of Behaviour Driven Development**. 37th EUROMICRO Conference on Software Engineering and Advanced Applications. Oulu: University of Limerick Institutional Repository. 2011. p. 383-387.

SOGILIS. XReq. **XReq**, 2012. Disponivel em: <http://xreq.forge.open-do.org/>. Acesso em: 28 Maio 2012.

SOMMERVILLE, I. **Software Engineering**. 7a. ed. [S.l.]: Addison Wesley, 2004.

SOMMERVILLE, I. **Software Engineering**. 8a. ed. [S.l.]: Addison-Wesley, 2006.

SOURCE, MAKING. Where Did Refactoring Come From. **Source Making - Teaching IT Professionals**. Disponivel em: <http://sourcemaking.com/refactoring/where-did-refactoring-come-from>. Acesso em: 26 Maio 2012.

WELLS, D. A Gentle Introdction. **Extreme Programming**, 2011. Disponivel em: <http://www.extremeprogramming.org/>. Acesso em: 25 Abril 2012.