BRENO HENRIQUE DUARTE DE OLIVEIRA

Práticas de *Behavior Driven Development* em *Scrum* para Entrega Contínua de Valor.

São Paulo
2015

BRENO HENRIQUE DUARTE DE OLIVEIRA

Práticas de *Behavior Driven Development* em *Scrum* para Entrega Contínua de Valor.

Monografia – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para conclusão do curso de especialização em Tecnologia de Software MBA-USP.

São Paulo
2015

BRENO HENRIQUE DUARTE DE OLIVEIRA

Práticas de *Behavior Driven Development* em *Scrum* para Entrega Contínua de Valor.

Monografia – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para conclusão do curso de especialização em Tecnologia de Software MBA-USP.

Área de Concentração: Tecnologia de Software.

Orientadora: Profª Drª. Selma Shin Shimizu Melnikoff.

São Paulo

2015

RESUMO

OLIVEIRA, Breno Henrique Duarte de. **Práticas de Behavior Driven Development em Scrum para Entrega Contínua de Valor**. 2015. 57p. Monografia – Programa de Educação Continuada da Escola Politécnica da Universidade de São Paulo, Curso de Especialização em Tecnologia de Software MBA-USP, 2015.

Este trabalho tem o objetivo de apresentar um guia de aplicação de boas práticas de *Behavior-Driven Development - Desenvolvimento Orientado a Comportamento* - BDD para entrega contínua de valores para cliente. Trata-se de uma pesquisa que aborda a importância das mudanças no desenvolvimento de software que surgiram com o aparecimento do Manifesto Ágil, de *Test Driven Development - TDD – Desenvolvimento Orientado a Testes* e de *Behavior-Driven Development – BDD - Desenvolvimento Orientado a Comportamento.* Este último assunto busca uma forma de evitar as falhas no sistema, através do envolvimento de teste de aceitação desde as atividades iniciais de planejamento e desenvolvimento de *software*. Os conceitos selecionados foram aplicados ao processo de desenvolvimento de uma empresa real, com foco na entrega contínua de valor para o cliente, e os resultados obtidos são apresentados.

Palavras-chave: Desenvolvimento Orientado a Comportamento (Behavior-Driven Development – BDD); Entrega contínua; Integração contínua.

ABSTRACT

OLIVEIRA, Breno Henrique Duarte de. **Práticas de Behavior Driven Development em Scrum para Entrega Contínua de Valor**. 2015. 57p. Monografia – Programa de Educação Continuada da Escola Politécnica da Universidade de São Paulo, Curso de Especialização em Tecnologia de Software MBA-USP, 2015.

This paper aims to present an application guide of good practice of Behavior- Driven Development - Behavior Driven Development - BDD for continuous delivery of value to customers. This is a research focusing on the importance of changes in software development that emerged with the appearance of the Agile Manifesto of Test Driven Development - TDD - Test-Driven Development and Behavior- Driven Development - BDD - Driven Development Behavior . This last issue is seeking a way to avoid system crashes, through acceptance testing of involvement from the initial planning activities and software development. The selected concepts have been applied to the development of an actual business process , focusing on continuous delivery value to the client, and the results are presented.

Keywords: Behavior Driven Development- BDD; Continuous Delivery; Continuous Integration.

SUMÁRIO

1. INTRODUÇÃO

Este capítulo apresenta a motivação, o objetivo, a justificativa, a metodologia utilizada para o trabalho e a estrutura da monografia.

1.1 Motivação

Diante da crescente pressão do mercado por inovação e exigências dos clientes, os projetos de *software* têm buscado excelência na qualidade dos sistemas, através de técnicas para melhorias nas diversas fases do seu desenvolvimento.

A evolução na área é notória e necessária, uma vez que, para proporcionar qualidade aos resultados dos processos organizacionais, os sistemas de *software* precisam atender aos requisitos iniciais de sua criação e, ao mesmo tempo, estar aptos à adaptação, caso mudanças eventualmente ocorram. Sistemas duráveis, flexíveis e, sobretudo, de fácil manutenção são, portanto, essenciais a esses propósitos.

O Manifesto Ágil, publicado em 2001 foi um marco no desenvolvimento de *software*, pois traz mudanças nos valores do processo de desenvolvimento de software orientado a processos e utilizando regras da ciência para um modelo dirigido por questões sobre pessoas e suas interações, sobre um *software* e como este pode ser concebido (KENT et al 2001). Como consequência, surgiram os métodos ágeis como XP (KNIBERG, 2007), *Scrum* (SCHWABER; SUTHERLAND, 2013) e vários outros, que atraíram os desenvolvedores para a nova abordagem. No entanto, a implantação desses processos não é imediata, sendo necessário haver mudanças de cultura e treinamento (MARTIN; SHORE; WARDEN, 2007).

Um dos conceitos interessantes introduzidos pelos métodos ágeis de desenvolvimento de software foi o Desenvolvimento Dirigido por Testes (Test-Driven Development – TDD), que resulta em um processo de desenvolvimento guiado pelos testes, ou seja, os testes são criados inicialmente, onde eles não passam pois não há o código de implementação, depois o desenvolvedor implementa o código o código da maneira mais simples possível; dessa forma, o desenvolvedor passa, pelo código implementado, melhorando-o e os testes já implementados anteriormente devem continuar como aceito.

Essa prática combina a técnica da definição de testes antes da programação, o projeto e a refatoração de código, e permite a obtenção de códigos mais próximos à real necessidade, pois o desenvolvedor está focado apenas no que é necessário fazer para que o teste do código tenha sucesso. Como resultado, a necessidade de alterações no código é menor que no caso de *software* desenvolvido através de modelo convencional, no qual as eventuais falhas de adequação de projeto são percebidas apenas nas fases finais do desenvolvimento. Além disso, faz parte do TDD, o registro das informações do teste no código, o que permite documentar os erros deletados, para que eles sejam evitados em um novo projeto.

Como evolução do TDD surgiu o Desenvolvimento Orientado a Comportamento (*Behavior-Driven Development* – BDD), proposto por North, para tornar a interação, entre a equipe de desenvolvimento e o cliente, mais produtiva, incorporando os conceitos de análise e teste de aceitação ao TDD (NORTH, 2006). O BDD incorpora as características do TDD, como o ciclo de criação de testes e refatoração de código de forma incremental, porém o foco dos testes é no comportamento do *software*. Dessa forma, o teste tem a finalidade de facilitar a compreensão do funcionamento do sistema, evitando falhas no entendimento do software e também do negócio.

No BDD, os trechos de especificação do comportamento do sistema são definidos de tal forma que o seu teste possa ser automatizado. Os testes definidos são claros e compreensíveis, pois o BDD utiliza de uma linguagem ubíqua que permite que pessoas de diversos perfis possam definir os testes desejados (SOLÍS; WANG, 2011). Dessa forma, o BDD incorpora as características do modelo de negócio para o qual o *software* se destina, pois permite transformar uma ideia em um requisito implementável. O teste do código é preparado para ser executado para avaliar o requisito específico do ponto de vista das pessoas de negócio, desenvolvedores e testadores, criando um comum entendimento do projeto.

Os testes projetados e realizados sistematicamente permitem ampla cobertura de verificação, o que os torna capazes de fornecer tanto a qualidade técnica dos sistemas quanto a satisfação e aceitação dos usuários. Por outro lado, os testes realizados dessa forma evidenciam a possibilidade de adaptação do

sistema às possíveis mudanças impostas pelo mercado agressivo às organizações, devido à possibilidade de repetir os testes que ficam documentados.

Para Humble e Farley, o foco no teste de aceitação, desde o início do desenvolvimento do sistema, aumenta a probabilidade de sucesso do sistema (HUMBLE; FARLEY, 2013):

> O foco em teste de aceitação, como uma forma de mostrar que a aplicação atende a seus critérios para cada requisito, tem um benefício adicional. Faz com que todos os envolvidos na entrega – clientes, testadores, desenvolvedores, analistas, pessoal de operações e gerentes de projeto – pensem sobre o que significa obter sucesso para cada requisito (HUMBLE; FARLEY, 2013, p. 26).

Ainda, em relação aos testes em métodos ágeis, a prática de integração contínua favorece a liberação de produtos com valor. De acordo com o artigo de Continuous Integration de Martin Fowler (FOWLER, 2000):

> Integração Contínua é uma prática de desenvolvimento de software em que os membros de uma equipe realizam a integração frequentemente. Geralmente cada pessoa realiza a integração pelo menos diariamente, podendo ocorrer várias integrações por dia. Cada integração é verificada por um processo de compilação automatizado (incluindo testes) para detectar erros de integração o mais rápido possível. Muitas equipes consideram que essa abordagem leva a uma redução significativa dos problemas de integração e permite que uma equipe desenvolva software coeso mais rapidamente. (FOWLER, 2000)

1.2 Objetivo

Considerando que as práticas citadas na seção anterior possuem relacionamento entre elas e que o BDD é uma prática ainda em desenvolvimento, torna-se necessário discutir a aplicação conjunta em um processo de desenvolvimento.

O BDD possibilita a validação do *software* através da definição de testes relacionados com seus requisitos; porém, para se ter resultados eficazes, é necessário que o desenvolvedor execute os testes sistematicamente e

constantemente. Com isso, ocorre a integração contínua que causa um *feedback* constante durante o desenvolvimento do *software*.

Dessa forma, este trabalho tem como objetivo, definir um guia de boas práticas para a aplicação de BDD em desenvolvimento de *software* que tenha como meta a integração contínua.

Para isso, optou-se em utilizar um processo ágil, no caso o *Scrum*, e selecionaram-se as tarefas, consideradas relevantes do BDD, que foram atribuídas para os papéis existentes no *Scrum*. Assim, as estórias do *Scrum* constituem o comportamento do sistema do ponto de vista do *Scrum* e o desenvolvimento de cada estória resulta no teste a ser executo no servidor de integração contínua.

1.3 Justificativa

Os métodos ágeis apresentam um diferencial para desenvolvimento de software, através de valores e práticas que consolidam o processo iterativo e incremental, apresentado no modelo espiral na década de 70 (PRESSMAN, 2011). Com uma abordagem "evolucionária" à engenharia de software, capacita o desenvolvedor e o cliente a entender e reagir aos riscos em cada fase evolutiva. Essas práticas, no entanto, não são facilmente compreendidas e aplicadas no processo de desenvolvimento e tem causado efeitos contrários aos esperados.

Apesar disso, e mesmo com a resistência de alguns setores e empresas, os métodos ágeis são cada vez mais utilizados e uma de suas exigências é a necessidade de maior grau de envolvimento e de comunicação entre membros da equipe, para garantir a negociação e o andamento do projeto (SHORE, 2007).

Uma das práticas interessantes dos métodos ágeis é a integração contínua (FOWLER, 2000) e, para sua viabilização, é de fundamental importância que todos os envolvidos no desenvolvimento estejam comprometidos com testes de integração planejados de forma a ocorrer continuamente, com qualidade na liberação das suas partes. Todos devem se envolver com código de todos, realizando melhorias contínuas através da refatoração. Com isso, obtém-se um processo de entrega de

novas funcionalidades e retificações com valor ao cliente (HUMBLE; FARLEY, 2013).

Uma das dificuldades da integração contínua é unificar e coordenar as atividades de requisitos, projeto, implementação e testes das iterações, para tornar o processo eficiente e confiável durante sua execução. Outra dificuldade é conseguir com que desenvolvedores, testadores e pessoal de operação trabalhem juntos de forma eficiente (HUMBLE; FARLEY, 2013).

Outra prática relevante nos métodos ágeis é o TDD, que trouxe uma grande contribuição ao propor a elaboração do procedimento de teste anteriormente ao projeto e codificação. Porém, ao operacionalizar esse conceito, constatou-se que não era fácil estabelecer um procedimento para identificar como começar os testes e definir o seu escopo, assim como os dominar, compreender e identificar as falhas (NORTH, 2006).

O BDD foi uma tentativa do North, para melhorar a aplicação do conceito do TDD, e integrou as atividades de análise de requisitos e teste de aceitação ao TDD (BECK, 2002). Para isso, o modelo e as regra de negócio foram introduzidos na visão de testes de aceitação. A linguagem para definição de testes é próxima à linguagem de usuários, que não possuem conhecimento em desenvolvimento de software, permitindo, assim, a sua participação na descrição de testes.

Por outro lado, com o BDD, o cliente pode escrever a especificação das estórias e, se as ações não forem coordenadas com a equipe de desenvolvimento, a solução pode não ser adequada, devido à sua falta de conhecimento técnico. Para que o resultado seja beneficiado com a diversidade de ideias, é importante que haja participação de grupos heterogêneos de pessoas trabalhando juntos. O cliente precisa de informações técnicas que o auxiliem na criação de cenários antes não pensados; caso contrário, a solução pode ser mais complexa do que necessária. Sendo assim, é necessário que a equipe de desenvolvimento o cliente trabalhem colaborativamente (PUFAL; VIEIRA, 2013).

Vale salientar que, no BDD, é recomendável o uso de ferramentas, como *Cucumber* (WYNNE; HELLES,2012), para viabilizar a sua implantação. Nesse caso, as estórias são escritas diretamente utilizando-se a linguagem da ferramenta e esse procedimento reforça a necessidade de uma interação efetiva do cliente com a

equipe de desenvolvimento, para juntos elaborarem possíveis ideias para o desenvolvimento do sistema (PUFAL; VIEIRA, 2013).

Dessa forma, o BDD, conta com a participação do cliente, para identificar as necessidades reais do cliente, para compor os cenários que descrevem o comportamento do software; com isso, facilita ao cliente visualizar o uso do software, apresenta a forma como o software é construído e aproxima o desenvolvimento de software aos negócios.

É necessário apontar a importância de organizar a construção do software ao cliente, pois mostra que o software pode ter qualidade desde o primeiro conjunto de testes que é executado em um ambiente próximo ao de produção, eventualmente em máquinas com custos mais acessíveis. Dessa forma, cria-se uma confiança maior no processo de implantação no ambiente de produção. Além disso, o cliente se conscientiza de que as falhas, detectadas na versão em teste, não avançam para os próximos estágios, devido ao teste contínuo, e o *software* final será de fato um apoio para melhorar o seu negócio.

Do ponto de vista da entrega contínua de software o processo deve considerar desde o início do desenvolvimento; as necessidades do cliente devem ser identificadas e agregadas aos critérios claros que resultem no sucesso do software. Essa prática causa evolução nos processos e no uso das ferramentas, até alcançar o estágio final, ou seja, a entrega contínua do *software* de valor.

Assim, o software está diretamente vinculado com os objetivos do negócio do cliente e torna-se imprescindível que este esteja sempre atualizado e disponível para a continuidade do negócio.

1.4 Metodologia

Para o desenvolvimento desse trabalho, inicialmente fez-se o levantamento bibliográfico e o estudo do material relacionado com métodos ágeis de desenvolvimento de *software*, com foco na integração contínua, TDD e BDD.

Selecionou-se o *Scrum* como método ágil para dar suporte à incorporação do BDD, por ser um dos métodos ágeis mais utilizados no mercado. Sobre o *Scrum*,

foram definidas as práticas do BDD e da entrega contínua, para detalhar o processo de teste.

Para a parte prática do trabalho, o método resultante foi aplicado em uma empresa. Foram utilizadas ferramentas para definir um ambiente que fornecesse apoio a esse experimento: Cucumber, Git e GO – Continuous Delivery. O *Cucumber* foi utilizado como ferramenta de auxílio para a escrita das estórias do BDD; o Git, como controle de versão do código; Go, como servidor de integração contínua.

O Cucumber, em particular, foi selecionado para possibilitar a escrita de funcionalidades do sistema em uma linguagem natural, tão essenciais na comunicação do cliente e a equipe. Há ainda de incluir que o processo possibilita a contínua entrega de valores ao cliente, através do apoio do Git e Go, assim validando constantemente as funcionalidades inseridas no software (HUMBLE; FARLEY, 2013).

A aplicação do método resultante ocorreu na empresa M com o intuito da evolução das entregas do *software* em ciclos constantes; observou-se que o *software* foi entregue constantemente com maior aderência aos requisitos e uma melhor cobertura de testes automatizados, para garantir a qualidade do *software*.

1.5 Estrutura da Monografia

Além da INTRODUÇÃO apresentada, esta monografia possui de mais três capítulos.

No Capítulo 2 TESTE NO DESENVOLVIMENTO ITERATIVO, são apresentados os conceitos e as técnicas relacionados com os testes no contexto de métodos ágeis de desenvolvimento de software.

No Capítulo 3 BOAS PRÁTICAS DO *BEHAVIOR DRIVEN DEVELOPMENT* NO *SCRUM*, é apresentada a utilização das boas práticas do BDD no contexto do *Scrum*, a aplicação do método resultante a uma parte do sistema referente ao processo de compra do *e-commerce* e a análise dos resultados obtidos.

No Capítulo 4 CONSIDERAÇÕES FINAIS são apresentados a conclusão, as principais contribuições deste trabalho e os trabalhos futuros.

2 TESTE NO DESENVOLVIMENTO ITERATIVO

O objetivo desse capítulo é apresentar os conceitos e as técnicas que permitam dar suporte ao entendimento desse trabalho. São apresentados os pontos relevantes do método *Scrum* e as práticas relevantes do TDD e BDD.

2.1Método *Scrum*

Os métodos ágeis passaram a ser definidos e utilizados após a publicação do Manifesto Ágil, elaborado por especialistas em desenvolvimento de software, que buscavam solucionar os frequentes problemas ocorridos em processos e em produtos gerados.

2.1.1 Principais Tópicos sobre *Scrum*

Essa seção apresenta o Manifesto Ágil e os conceitos relevantes do *Scrum*. Esse método foi selecionado por ser um dos métodos ágeis mais utilizados no mercado.

2.1.2 Manifesto Ágil

Em fevereiro de 2011, 17 profissionais da área de *software* reuniram-se para discutir as alternativas para processos utilizados na época, considerados não adequados por serem pesados e orientados por documentos. Das práticas adotadas por esses participantes, entre eles Kent Beck, Robert C. Martin e Martin Fowler, surgiram o Manifesto para Desenvolvimento Ágil de *Software*, conhecido como Manifesto Ágil que apresenta quatro valores e doze princípios, que constituem a base para o desenvolvimento ágil de software *(*KENT et al., 2001).

Os autores declaram que estão identificando maneiras melhores de desenvolver software, através dos seus trabalhos, para ajudar os outros

profissionais nessa tarefa. Os valores apresentados do Manifesto Ágil são os seguintes *(*KENT et al., 2001):

1. Indivíduos e interação entre eles, mais que processos e ferramentas.
2. *Software* em funcionamento, mais que documentação abrangente.
3. Colaboração com o cliente, mais que negociação de contratos.
4. Responder a mudanças, mais que seguir um plano.

Para eles, mesmo que os itens à direita dos valores apresentados sejam importantes, os itens à esquerda devem ser mais considerados.

Os princípios do Manifesto Ágil são os seguintes *(*KENT et al., 2001):

1. A maior prioridade é satisfazer o cliente, através de entregas com antecedência e contínua, de *software* com valor.
2. Acolher as mudanças nos requisitos mesmo na etapa tardia do desenvolvimento. Os processos ágeis aproveitam a mudança para obter vantagens competitivas ao cliente.
3. Entregar *software* em funcionamento com frequência, entre 2 semanas a 2 meses, dando preferência a períodos mais curtos.
4. Pessoas relacionadas com negócio e desenvolvedores devem trabalhar em conjunto ao longo do projeto.
5. Construir projetos ao redor de indivíduos motivados, dando a eles o ambiente e suporte necessários, e confiar na realização do seu trabalho.
6. O método mais eficiente e eficaz, para transmitir informações para uma equipe de desenvolvimento, é através de uma conversa frente a frente.
7. Software em funcionamento é a medida principal do progresso.
8. Processos ágeis promovem um ambiente sustentável. Os patrocinadores, desenvolvedores e usuários, devem ser capazes de manter passos constantes indefinidamente.
9. Contínua atenção à excelência técnica e ao bom *design* aumenta a agilidade.
10. Simplicidade - a arte de maximizar a quantidade de trabalho não realizada – é essencial.
11. As melhores arquiteturas, requisitos e *designs* emergem de equipes auto-organizadas.

12. Em intervalos regulares, a equipe reflete em como ser mais efetiva; então, se sintonizam e ajustam seu comportamento de acordo.

O Manifesto Ágil, através de seus valores e seus princípios sintetiza a filosofia dos autores.

2.1.3 *Scrum*

Denominado de *framework* por Ken Schwaber e Jeff Sutherland (SCHWABER; SUTHERLAND, 2013), o *Scrum* permite o uso de diversos processos e técnicas para desenvolver e manter produtos complexos, entregando produtos com mais alto valor possível. O processo resultante da sua aplicação é classificado como ágil, devido às características aderentes ao Manifesto Ágil.

O *Scrum* apresenta três papéis principais: *Scrum Master, Product Owner* e *Scrum Team*.

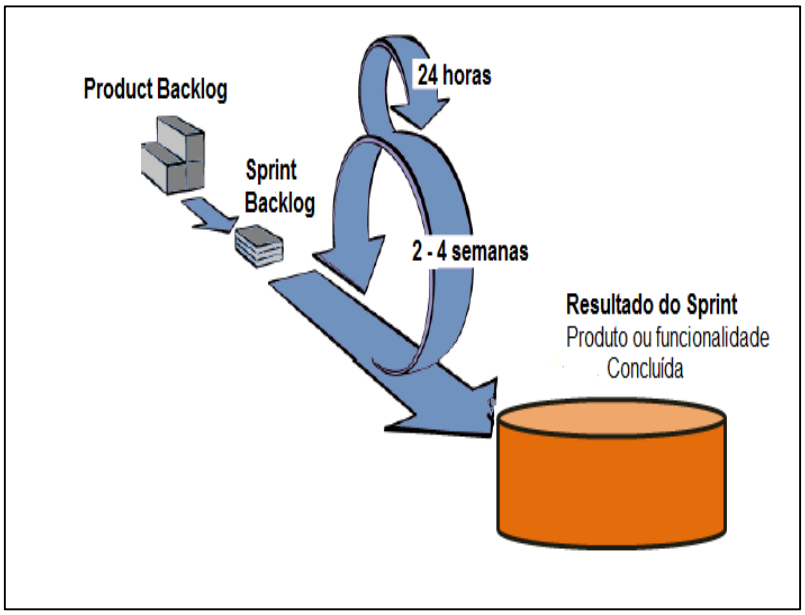
O *Scrum Master* tem o papel de liderar os processos e, para isso, deve fazer com que os participantes entendam e incorporem os valores, os princípios e as práticas do *Scrum*. Além disso, o *Scrum Master* desempenha o papel de facilitador, removendo interferências externas, resolvendo problemas e atuando na equipe para realizar melhorias.

O *Product Owner* é o representante do negócio e das partes interessadas; sua responsabilidade é definir as funcionalidades, a serem construídos, os recursos necessários, bem como, manter a comunicação entre os interessados, para que todos tenham os mesmos objetivos no projeto.

O *Scrum Team* é uma equipe multifuncional (entre 5 e 9 pessoas) responsáveis pela concepção, construção e testes do produto, ou seja, atuam na análise, no projeto, na implementação e no teste. A equipe se auto-organiza para atingir a meta estabelecida pelo *Product Owner*.

O fluxo de processo do *Scrum* está representado na figura 1.

Figura 1 — Ciclo da metodologia Scrum



O *Product Backlog* é o conjunto de funcionalidades priorizadas do *software* a ser desenvolvido e é definido pelo *Product Owner*. Cabe salientar, que o *Product Backlog* é um documento em constante desenvolvimento e evolução, por conta de mudanças que possam ocorrer no negócio ou devido à melhor compreensão do produto pelos participantes do projeto.

No *Scrum*, a execução do projeto é dividida em ciclos chamados *Sprints* e cada um corresponde a uma iteração, cuja duração é em média de 2 a 4 semanas, com reuniões diárias de 15 minutos no máximo, denominadas *Daily Meeting*, para acompanhar o trabalho. Na execução de cada *Sprint*, deve-se criar uma versão de valor tangível para o cliente ou usuário.

Para planejar cada *Sprint*, é realizada uma reunião em que o *Product Owner*, o *Scrum Team* e o *ScrumMaster* definem o *Sprint Backlog*, que contém as funcionalidades a serem implementadas durante o *Sprint*.

Definidos os itens prioritários do *Product Backlog*, estabelece-se a sequência priorizada dos itens, a partir de fatores como custo, risco e valor.

No final do *Sprint*, são realizadas duas atividades fundamentais:

1. *Sprint Review,* cujo objetivo é rever os itens concluídos e verificar se o produto atende à expectativa de entrega.

2. *Sprint Retrospective*, cujo objetivo é identificar o que está funcionando adequadamente, o que precisa ser melhorado e quais as ações a serem tomadas para melhorarias. O *Sprint Retrospective* é realizado no final da *Sprint Review* para verificar as melhorias nas formas de trabalho para o próximo *Sprint*.

O resultado do *Sprint* é um produto ou uma funcionalidade. No final do *Sprint* , é gerado um documento chamado *Definition of Done – DoD,* que é um acordo feito pelo *Scrum Team*, no qual é apresentado de maneira clara, o conjunto mínimo de passos necessários para a entrega do resultado com qualidade.

2.2 *Extreme Programming*

O *Extreme Programming* (*XP*) foi criado por *Kent Beck*, em um projeto crítico de folha de pagamento para a empresa *Chrysler*; nele Beck selecionou um conjunto de práticas que haviam se mostrado eficientes separadamente em outros projetos e as aplicou juntas e potencializadas e, isso foi a origem do o *XP*. Beck pode perceber que a revisão de código, testes, integração rápida, *feedback* do cliente, *design* simples, entre outras práticas, eram atividades que contribuíam para a maior qualidade do produto. Então, sua proposta foi intensificar a utilização delas ao extremo, fazendo, por exemplo, revisão constante do código através de programação em pares, intensificando o uso testes automatizados, antecipando a criação dos testes com testes antes mesmo da implementação do código e permitindo um acompanhamento constante do projeto com o cliente presente (BECK, 2002).

O TDD é uma prática aplicada na comunidade que atua com metodologias ágeis e foi introduzida inicialmente através do *XP,* por Kent Beck em 2002.

O TDD utiliza a filosofia de que o desenvolvimento deve ser feito do teste para o código e é realizado em ciclos, que cobrem pequenos trechos, definindo-se o teste para uma função desejada, antes da sua implementação.

Com isso, o TDD procura integrar a escrita e a verificação do código de forma simultânea e, assim, viabiliza a agilidade no ciclo de desenvolvimento, através do melhor entendimento das funções a serem codificadas.

No entanto, por ser uma tecnologia relativamente nova, é necessário treinar os desenvolvedores, o que demanda tempo para aprendizado. Além disso, do ponto de vista da produção de código, o TDD dá a falsa impressão de diminuir a produtividade; mas, por outro lado, o código gerado tende a ter maior qualidade, evitando erros e correções que ocorrem na implantação.

O mecanismo do TDD é apresentado na Figura 2.

Figura 2 — TDD – Test Driven Development

O ciclo do TDD é dado na seguinte forma:

1. O desenvolvedor cria um teste para um código inexistente, fazendo com que o teste recém-criado não passe na sua execução.
2. Nessa etapa o desenvolvedor escreve o código de implementação da maneira mais simples, para o teste, de maneira a fazer o teste ser executado com sucesso.
3. O desenvolvedor volta à implementação para refatorar o código, de forma a evoluir a implementação ou fazer com que outros cenários do teste também sejam executados com sucesso.

Segundo Kent Beck (BECK, 2002), o ciclo de TDD pode ser feito em pequenos passos para obter resultados rapidamente e uma compreensão sobre o que se está sendo desenvolvido.

A adoção do TDD vem se tornando cada vez mais popular em empresas de desenvolvimento de *software*, pois o método faz com que o desenvolvedor crie um teste automatizado para determinado cenário, que até então eram manuais, sujeitos a falha humana (BECK, 2002). Sua adoção no início é mais complicada para desenvolvedores, pois em geral, o desenvolvedor não compreende o que é escrever um teste que falhe na sua execução (NORTH, 2006).

Observe que o uso da prática de TDD ajuda a equipe a garantir que os requisitos funcionam como esperado, e também auxilia o desenvolvedor a identificar problemas de código na sua implementação.

2.3 BDD

O BDD é uma abordagem de desenvolvimento de *software* proposta por Dan North em 2003, em resposta às dificuldades dos desenvolvedores ao utilizar o TDD. As dúvidas na aplicação do TDD eram várias e aconteciam desde como começar os testes, o que deveriam ou não testar, até como compreender porque os testes falharam. Para auxiliar os desenvolvedores, North propôs o BDD, que inicia o desenvolvimento do sistema a partir da visão do teste do seu comportamento, ressaltando as funcionalidades que agreguem maior valor aos interessados (NORTH, 2003).

2.3.1 Conceitos do BDD

Para reconhecer o valor das funcionalidades para o negócio, o BDD faz com que os desenvolvedores discutam sobre exemplos concretos do comportamento do sistema com os profissionais de negócio. Dessa forma, o BDD propicia uma interação mais intensa entre analistas de negócio, desenvolvedores e testadores de *software*, pois os requisitos devem ser expressos de forma que facilite os seus testes, e de forma compreensíveis para a equipe de desenvolvimento e interessados da área de negócio. As ferramentas de BDD permitem a conversão dos requisitos em testes automatizados, o que ajuda o trabalho do desenvolvedor para verificar as funcionalidades e para documentar o sistema (SMART, 2014).

A interação entre a equipe de negócio e de tecnologia é possibilitada no BDD pela existência de uma linguagem comum (ubíqua) para especificar o comportamento do sistema. Essa linguagem permite que (LAZAR; MOTOGNA; PÂRV, 2010):

1. Clientes especifiquem os requisitos do ponto de vista de negócio;
2. Analistas de negócio definam exemplos concretos (cenários e testes de aceitação) que tornem claro o comportamento do sistema;
3. Desenvolvedores implementem o comportamento do sistema usando TDD.

Ainda, podem-se citar os três princípios do BDD (NORTH, 2003):

1. O suficiente é suficiente: Não se deve automatizar todo o processo de negócio, mas o que descreve o comportamento esperado do produto pelo cliente isso é suficiente para desenvolver a solução e, mais do que isso, é desperdício de esforço.
2. Entregar valor para os clientes: deve-se entregar somente o que tem valor para o cliente. Se não agregar valor para o cliente, ou não potencializar o valor entregue, deve-se descartar essa parte.
3. Tudo é comportamento; independentemente do nível de teste, a funcionalidade sempre é descrita como comportamento: Tudo que um *software* faz pode ser descrito como comportamento e explicado para qualquer pessoa que tenha o domínio do negócio.

North (2006) esclarece que o BDD acrescenta vantagens ao TDD através de fatos como comunicação entre as equipes, compartilhamento de conhecimento, documentação dinâmica, visão do todo, além de criar uma conexão entre a definição do negócio e a criação dos testes.

O desenvolvimento de um *software* através de um processo guiado por testes torna a manutenção bem mais acessível no futuro, pois a cada nova funcionalidade ou refatoração de código, o desenvolvedor executa os testes, já criados anteriormente, garantindo que o que já foi desenvolvido antes não tenha o comportamento alterado pelas novas funcionalidades ou refatoração (BECK, 2002).

2.3.2 Práticas do BDD

O BDD tem, como característica, ser guiado para e pelos os valores do negócio que motiva ou solicita a produção do *software*. Apesar de ter sua filosofia semelhante à de TDD, de elaborar o teste antes da codificação, o foco do BDD é no comportamento do *software* e não nas funções do seu *software.* Dessa forma, deixa os participantes do projeto em situação mais confortável para pensar no sistema como um todo e elaborar os cenários de teste mesmo antes do seu desenvolvimento. O BDD permite definir ideias acerca do seu funcionamento, amadurecê-las através de cenários de teste e transformá-las em requisitos que serão implementados e testados. Essa dinâmica torna o processo de desenvolvimento mais simples e eficaz (NORTH, 2006).

Para isso, precisa-se de uma forma para descrever os requisitos para que os participantes, tais como especialistas de negócios, desenvolvedores e testadores tenham um entendimento comum do *software* a ser desenvolvido. No BDD, a descrição de um requisito é feita através de uma estória, que descreve um requisito e seu benefício no negócio, a qual deve obter a concordância dos participantes.

A descrição de requisitos é feita a linguagem ubíqua, que tem a característica de poder ser utilizada com certa facilidade tanto pelo cliente quanto pela equipe de desenvolvimento. Esse recurso é inspirado na técnica do *Domain-Driven Design* – DDD, para melhorar o diálogo entre os especialistas de domínio e de aplicação (AVRAN; MARINESCU, 2007).

Para que processo do BDD seja executado com eficácia, é necessário envolver as pessoas desde o desenvolvedor ao cliente e descrever exemplos do comportamento da aplicação ou unidade de código, para esclarecer requisitos e outros cenários. Uma vez com as estórias definidas, criam-se os testes automatizados de rápida execução para o desenvolvedor executar os devidos cenários de teste constantemente, ou seja, o desenvolvedor executa esses testes diversas vezes ao dia.

Outra prática sugerida pelo BDD é a utilização de simuladores de teste (*mocks*, *stubs*, *fakes*, *dummies*), que são unidades auxiliares para permitir a colaboração com módulos e códigos que ainda não foram escritos.

2.4 Integração e entrega contínua de *software*

As interações dos métodos ágeis, bem como as técnicas contidas no TDD e BDD, têm a preocupação de realizar a Integração Contínua, ou seja, uma prática de desenvolvimento de software na qual os membros da equipe integram seu trabalho constantemente, para serem executados em um servidor que detecta através dos *builds* (incluindo teste) possíveis erros o quanto antes.

A entrega contínua, por sua vez, é uma prática que permite liberar os produtos de software com frequência, o que agiliza o feedback entre o desenvolvedor e a equipe, além de oferecer maior segurança para o cliente. Essa prática é frequentemente citada como o primeiro princípio dos doze princípios do manifesto ágil: “Nossa maior prioridade é satisfazer o cliente através da entrega contínua e adiantada de software com valor agregado” (KENT et al., 2001) .

O BDD utiliza o método de entrega contínua, ao entender uma mudança que não considera os critérios estabelecidos pelo cliente, leva os interessados a buscar soluções (HUMBLE; FARLEY, 2013), sendo assim é possível corrigir o problema assim que ele ocorre, isso permite disponibilizar sempre o software em estado funcional, ou seja, pronto para ser instalado em ambiente de produção.

A entrega contínua é a habilidade do BDD para se adaptar, bem como, para responder às mudanças e, assim, garantir a sobrevivência do projeto. Para isso, se faz necessário verificar se o *software* realmente apresenta o valor esperado, o que nem sempre acontece, pois, em geral, espera-se que a maioria ou todas as funcionalidades do sistema estejam implementadas, para então detectar e solucionar possíveis erros, o que demanda uma quantidade razoável de trabalho, (HUMBLE; FARLEY, 2013).

A integração frequente, para incorporar as mudanças que ocorrem durante o desenvolvimento de *software* promove a integração contínua. Para Fowler (2000), é uma prática de desenvolvimento de *software* em que os membros de uma equipe integram o sistema frequentemente e, normalmente, cada pessoa deve integrar pelo menos uma vez por dia.

Cada procedimento construído é testado, caso seja identificado possíveis erros eles são corrigidos. Esse método evita problemas relativos à integração, bem como possibilita o desenvolvimento do software coeso e de forma rápida. Por isso, a integração deve ser feita de maneira contínua e com testes de aceitação constantes

até que os requisitos se completem, ou seja, até que o código que satisfaz as necessidades do cliente seja aprovado em todos os testes.

Os códigos são utilizados como um repertório, que juntamente com um sistema de controle de versões, acompanha os requisitos e viabiliza versões diferenciadas dos arquivos.

Assim como o repositório de códigos possui sua funcionalidade dentro do desenvolvimento do software, a integração só se faz contínua se todos os envolvidos no projeto consigam instalá-lo e executar todos os testes em diferentes máquinas. Para isso, um script de build automatizado desempenha um papel relevante dentro do desenvolvimento do software, que informa como compilar códigos-fontes, instalar junto com suas dependências, executar testes, assim como, notificar a equipe problemas ou falhas dentro do processo.

Segundo Fowler (2000), o ponto principal é usar o mesmo ambiente, para que seja possível evitar problemas antes do *software* passar para produção. Tantos problemas relacionados a versões de dependências quanto ao do sistema operacional. Essa prática mostra-se eficiente, pois a equipe não tem surpresas na hora de implantar o sistema no ambiente de produção, pois o ambiente é idêntico ao ambiente em que o *software* foi desenvolvido e testado.

As práticas na equipe, como compartilhar versões de código, instalar, executar testes e validar sistema a qualquer momento, evitam surpresas no momento da implantação e, consequentemente, todas as novas funcionalidades do software podem evoluir juntas são fundamentais para a integração contínua.

A integração contínua integra sintetiza-se em ter um *feedback* rápido para assegurar um *software* quando pronto possa ser colocado em produção.

2.5 Importância do *feedback*

Ultimamente denota-se um crescimento de interesse em Testes de *Softwares,* confirmando-se a necessidade de agilidade do desenvolvimento de sistemas de

software e, assim, promover uma mudança evolucionária no ciclo de vida do projeto, cujo ambiente é estável e acolhedor.

Mesmo com o crescimento nos negócios de Testes de *Software*, permanece a sua essência de manter a interação entre os envolvidos, de forma a produzir o que é interessante para todos. Assim, todos contribuem na documentação, independente do modelo de negócios e, com isso, o processo do *software* é dirigido e modificado com a finalidade de atender as necessidades dos envolvidos.

Os envolvidos no projeto de *software* criam equipes ou comunidades que incluem os usuários aos desenvolvedores e, muitas vezes, o cliente atua junto para compor o melhor *software*.

Dentro da pesquisa realizada, foram encontrados poucos estudos que possam contribuir para a documentação detalhada de Testes de S*oftwares*, em projetos que envolvam a contratação de uma ou mais desenvolvedores por um curto período de tempo. A prestando serviço ao cliente deve ser de maneira integrada com valor de negócio, assim como, a cooperação deve ser no sentido de desenvolvimento funcional do software, tornando-o útil ao cliente.

É importante ressaltar que o cliente deve orientar ou criar as hipóteses em relação às correções e às funcionalidades mais úteis para os usuários. Para Humble e Farley (2013), o objetivo deve ser a entrega o *software* com qualidade suficiente para gerar valor aos seus usuários.

A entrega de um *software* com qualidade é, algumas vezes, considerada uma arte, mas deveria ser considerada como uma disciplina de engenharia. Para isso, deve ocorrer o envolvimento do cliente no projeto de s*oftware* e o resultado depende da experiência que a equipe têm no relacionamento com clientes. Mas o sucesso de um projeto de s*oftware* depende da habilidade do fornecedor de responder rapidamente à demanda das funcionalidades pelos clientes e da colaboração do cliente para responder ou entregar rapidamente às solicitações.

Sendo assim, a agilidade com que ocorre um ciclo de entrega em um projeto de software é muito importante, pois a disponibilidade rápida de uma manutenção solicitada pelo cliente faz todos os envolvidos perceberem a utilidade da entrega rápida como foco de negócio.

Nesse contexto, o processo de mudança de software pode ser visto com certo grau de confiança, porque a descoberta de problemas nos produtos de entrega

rápida e a busca de soluções em sequência, minimiza o tempo do ciclo, bem como, possibilita gerar uma nova versão do sistema.

O curto período entre as entregas faz com que o intervalo entre as conversas dos envolvidos seja pequeno, aumentando com isso o *feedback.* Esse mecanismo traz melhorias contínuas e possibilita a entrega rápida do *software* ao cliente.

Quanto mais frequentes forem as entregas ao cliente, o processo leva a um *feedback* mais rápido, principalmente se o teste de cada mudança ou correção for feito de maneira mais automatizada possível. Em testes ágeis, para cada mudança ou correção realizada, o *feedback* deve ser enviado para a equipe responsável que deve recebê-lo, compreendê-lo e atuar sobre o sistema para atender ao *feedback,* se for necessário.

Para qualquer mudança realizada, o sistema completo deve ser testado para garantir que não ocorreram efeitos colaterais. Os testes variam de acordo com cada tipo de sistema e verificam o funcionamento do seu código, seu comportamento, se está dentro do esperado, o valor do negócio esperado, se atende às necessidades dos usuários, Para garantir o resultado, o ambiente de teste devem ser o mais próximo possível do ambiente produtivo. Segundo Humble e Farley (2013):

> Um dos elementos de nossa abordagem é a necessidade de *feedback* rápido. Garantir esse f*eedback* em mudanças exige atenção ao processo de desenvolvimento do código. Os desenvolvedores devem realizar *commits* frequentes para o sistema de versionamento e separar o código em componentes para gerenciar equipes maiores ou distribuídas. Criar novos *branches* deve ser evitado na maioria das vezes.

Ainda, a resposta rápida ao *feedback* torna-se fundamental para que todos estejam envolvidos no processo de desenvolvimento do *software* e busquem a melhoria contínua na sua entrega, o que requer muito planejamento e disciplina.

2.6. Considerações finais

No ambiente competitivo do mercado de *software,* é necessário investir em processos de desenvolvimento de *software* que resultem em produtos de maior qualidade para atender as necessidades dos usuários de forma mais eficaz.

Nesse contexto, os métodos ágeis foram bem aceitos pela sua capacidade de aproximar os clientes de fornecedores, através da maior interação entre eles e da possibilidade da liberação de entregáveis com maior frequência.

Ao agregar as abordagens de BDD, os métodos ágeis têm a qualidade dos entregáveis melhorada, pelo fato de trazer uma cultura que agrega valores relevantes aos clientes. Com isso, ocorre um rápido ciclo de *feedback* a cada entrega, favorecendo a qualidade do *software* entregue e mais valor é entregue com maior frequência ao cliente.

Esse resultado pode ser obtido, pois a visão do teste é elaborada a partir do comportamento do sistema, uma vez que o BDD é voltado para testes de aceitação, viabilizando a criação de cenários para novas funcionalidades e garantindo a entrega dentro do que realmente o cliente espera, ou seja, entrega contínua de valores.

3. BOAS PRÁTICAS DO BEHAVIOR DRIVEN DEVELOPMENT NO SCRUM

Este capítulo apresenta a seleção de um conjunto de boas práticas do BDD para ser aplicado a um processo ágil, sua aplicação em uma empresa real de pagamentos digitais e a discussão dos resultados obtidos.

3.1 Desafios dos profissionais de *software*

Com a utilização da Internet nos negócios, as fronteiras se expandiram e o reflexo disso chegou ao mercado. Uma empresa, por exemplo, que atuava apenas no seu bairro ou na sua cidade, pode vender, através do uso da Internet, para qualquer lugar do Brasil ou do mundo. Essa situação passou a gerar a necessidade de um grande volume de desenvolvimento de sistemas que, por sua vez, demandavam evolução para se adaptar às novas condições geradas pelo negócio.

Dessa forma, um grande desafio que os profissionais da área de desenvolvimento de *software* enfrenta é transformar boas ideias em sistemas, adicionar novas características aos sistemas existentes e entregar produtos com alta qualidade aos usuários, dentro de um prazo previsto (HUMBLE; FARLEY, 2013). Para isso, uma das características relevantes de um processo de desenvolvimento, que responda a essa demanda, é possuir uma forma de realizar os testes de modo sistemático.

Atualmente, o desenvolvimento de software tem várias atividades, tais como teste, implantação e instalação, realizadas manualmente e seus resultados devem ser documentados (HUMBLE; FARLEY, 2013).

Na implantação e instalação de software, os ambientes, em geral, são criados manualmente, por uma equipe de operação, que realiza seguintes tipos de passos:

1. Instalar o *software* de terceiros do qual a aplicação depende;
2. Carregar a aplicação no ambiente de produção;
3. Configurar os servidores Web, servidores de aplicação ou outros componentes do sistema criados por terceiros;

4. Copiar os dados de referência para os servidores em questão;
5. Iniciar a aplicação por partes, se o sistema for distribuído ou orientado serviços.

Esses passos geram uma série de informação que deve ser registrada para que possa ser utilizada posteriormente na implantação e na instalação de novas versões.

3.2. Resumo do problema

Em relação à evolução de um *software*, a adição de novas características não deve alterar o comportamento das demais que não foram alteradas. Para garantir isso, o impacto das alterações solicitadas deve ser avaliado e a sua inclusão deve ser feita seguindo os procedimentos estabelecidos. Além disso, devem-se executar todos os testes contidos no plano de teste, a fim de verificar a não existência de efeitos colaterais (CHELIMSKY et al., 2009). Esse procedimento deveria, em princípio, fazer parte da rotina diária de um desenvolvedor; porém, muitas vezes, o procedimento é descartado por uma série de motivos.

Um deles é a falta de recursos ou conhecimento dos desenvolvedores para analisar o impacto das alterações, pois os documentos atualizados podem não estar disponíveis (NORTH, 2006).

Outro motivo frequente é a falta de cultura de elaborar planos de teste, o que obriga os desenvolvedores a definir novos conjuntos de teste a cada alteração de software. Dessa forma, além do retrabalho para definição dos testes a serem executado, o resultado depende da experiência do profissional encarregado (NORTH, 2006).

Ainda, deve-se considerar o tempo para realizar os testes; às vezes pode ser necessário refazer parte do teste de integração, realizando testes em interfaces, acesso a banco de dados, ou até uma integração com *software* de terceiros (BECK, 2002). Em geral, o prazo estimado para a alteração não é suficiente e, muitas vezes, torna a execução do plano de teste inviável, pois se faz necessário alterar a

configuração de ambiente, o que pode ser muito demorado.

Além disso, após a evolução do sistema, as alterações realizadas devem ser atualizadas nos seus documentos para que se possa garantir próximas adições e alterações das características do sistema. A atualização desses documentos é uma tarefa complexa, que consome tempo significativo e necessita da colaboração de diversos tipos de profissionais e, se não for bem controlada, os documentos geralmente ficam incompletos ou ultrapassados (HUMBLE; FARLEY, 2013).

Para novos sistemas, cuidados devem ser tomados ao longo do desenvolvimento, para que os sistemas apresentem características que deem suporte ao teste, tais como arquitetura, componentes com alta coesão e baixo acoplamento, planos e procedimento de teste, entre outras (BECK; FOWLER, 2000).

Uma falha, em qualquer ponto do processo de desenvolvimento ou evolução, pode causar grande impacto ao usuário final e, como consequência, trazer prejuízos financeiros às organizações.

3.3 Melhorias do Processo de Teste através do BDD

Dentre os assuntos citados na seção 3.2, o processo de teste é um ponto importante para a melhoria da qualidade de software e, por esse motivo, foi selecionado como o foco do trabalho. A proposta consiste em incorporar os conceitos e as práticas do BDD, que agregam os aspectos de negócio ao TDD, a um processo ágil, que utiliza a prática do TDD e da Integração Contínua. Foi selecionado o método *Scrum*, por ser um dos mais utilizados no mercado.

3.3.1 Seleção das Práticas do BDD

O BDD foi analisado e foram selecionadas, as práticas mais relevantes para serem incorporadas no processo do *Scrum* e devem ser informados aos participantes do projeto, para uniformizar a visão de todos. As seguintes práticas do

BDD foram selecionadas:

1. Maior envolvimento das pessoas interessadas no processo de desenvolvimento;
2. Uso de exemplos para melhorar o entendimento do comportamento de uma aplicação;
3. Automação de cenários de testes desses exemplos para obter um *feedback* rápido do cliente;
4. Testes de regressão para garantir que comportamentos anteriores não sejam alterados por novos comportamentos.

3.3.2 Incorporação das práticas do BDD no *Scrum*

Para incorporar as práticas do BDD ao *Scrum*, foram feitas substituição ou detalhamento das suas atividades originais, ou inserção de novas atividades. O novo processo obtido foi denominado de *Scrum*-BDD.

Os dois papéis do Scrum: *Product Owner* e *Scrum Team*, descritos na seção 2.1.1, tiveram a sua responsabilidade ajustada da seguinte forma:

1. O Product Owner deve definir dos comportamentos indispensáveis para o *software* a ser desenvolvido, com a utilização de exemplos;
2. O *Product Owner* deve elaborar a especificação de testes para os cenários;
3. O *Scrum Team* deve criar testes automatizados dos cenários descritos pelo *Product Owner;*
4. O *Scrum Team* deve preparar o servidor para a Integração Contínua, para o rápido *feedback.*

Os comportamentos indispensáveis, do ponto de vista do BDD, são as funções que o sistema deve possuir e correspondem às funcionalidades do *Product Backlog* do *Scrum*. Então, o tratamento de uma funcionalidade deve ser o mesmo dispensado a um comportamento indispensável do BDD e a definição do *Product Backlog* deve ser feita seguindo as recomendações do BDD, utilizando palavras chaves que permitem, a quem está escrevendo, elaborar um texto que possa ser compreendido por qualquer integrante da equipe. No BDD um desenvolvedor ou profissional do setor de qualidade ou até mesmo o cliente pode escrever as estórias,

correspondentes às funcionalidades, passo-a-passo, seguindo o modelo da figura 3.

Figura 3 — Modelo de escrita de uma funcionalidade.

Funcionalidade: ...
Como um (a) …
Quero ....
Com o objetivo ...

O *Sprint Backlog,* que contém as tarefas a serem realizadas nas próximas iterações, deve incluir novos tipos de tarefa e alterar as tarefas originais do *Scrum*, para acomodar a descrição dos cenários, a elaboração da especificação dos testes, e a geração dos testes automatizados baseados nessas descrições.

A descrição de cenários do BDD, é feita seguindo os modelos das figuras 4 e 5.

Figura 4 — Modelo de escrita de um cenário.

Cenário: ...
Dado ...
Quando ...
Então...

Figura 5 — Modelo de escrita de um cenário mais complexo.

Cenário: ...

Dado ...

Quando ...

Então…

E

Cenário: ...

Dado ...

E ...

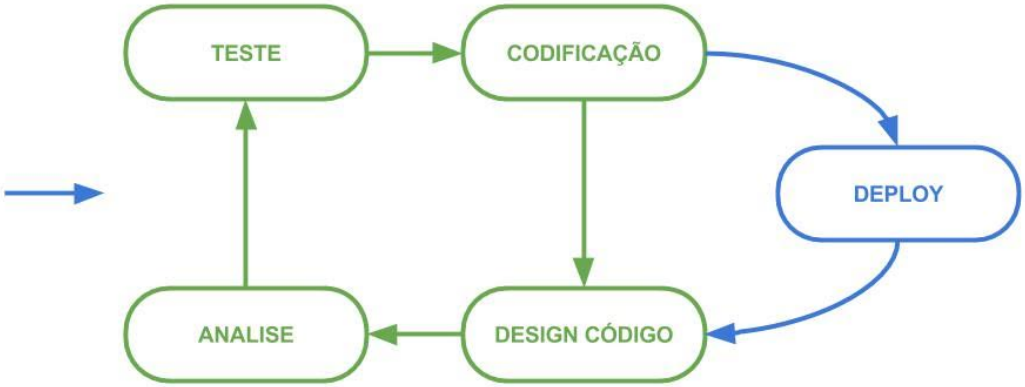
Quando ...

Então ...

E ...

Com a inclusão das práticas de BDD, a execução de um Sprint passa a ser feita conforme apresentada na figura 6.

Figura 6 — Diagrama de funcionamento.



A elaboração da especificação de testes para os cenários deve ser feita através de uma estória, ou seja, da descrição de um requisito e seu benefício no negócio, e de um conjunto de critérios de como o requisito será implementado, o qual tem o de acordo de todos os participantes do projeto (NORTH, 2006). A utilização de ferramentas apropriadas, durante o desenvolvimento, permite melhor elaboração e a automação dessas estórias, pois permitem a escrita das estórias em texto puro.

Para automatizar e executar as estórias é necessário preparar o servidor de Integração Contínua, no entanto, essa é uma atividade já existente no *Scrum*, pois os métodos ágeis pressupõem esse tipo de prática. Os testes serão realizados no servidor de Integração Continua e, caso sejam executados com sucesso, o servidor de Integração Continua deve disponibilizar o software para instalação no ambiente de produção.

A atribuição das tarefas incluídas ou alteradas aos papéis do *Scrum*-BDD pode ser vista na Tabela 1.

Tabela 1 – Atribuição das tarefas novas e alteradas aos papéis do *Scrum*

Papéis do *Scrum*	Tarefas
Scrum Master	Especialização do *Scrum* com as tarefas do BDD incluídas Disseminação do *Scrum*-BDD

Product Owner	Definição dos comportamentos indispensáveis Descrição dos cenários para os comportamentos Elaboração da especificação de testes para cenários
Scrum Team	Criação dos testes automatizados para a especificação de testes Preparação do servidor para Integração Contínua

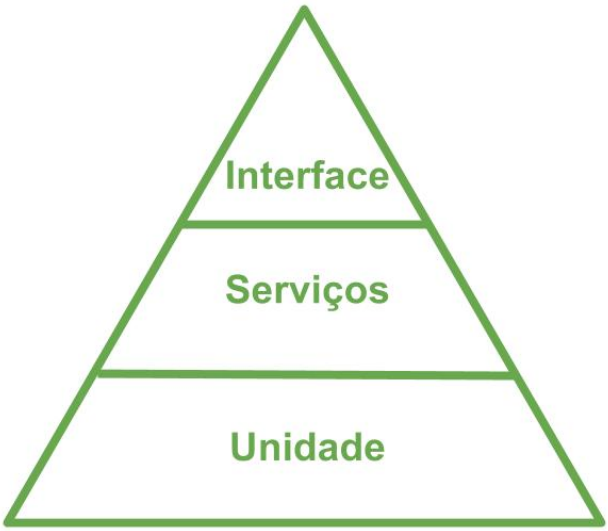
O *Scrum Master*, como líder do processo, é responsável pela especialização do *Scrum* para incorporar as atividades do BDD e pela disseminação da prática do *Scrum*-BDD. Para isso, deve ter um bom conhecimento do BDD.

O *Product Owner* deve ter a visão de negócio e do projeto, para definir o *Product Backlog,* e coordenar a execução dos *Sprints*. Devido a esse perfil, foi atribuído a ele a definição dos comportamentos indispensáveis e a descrição dos cenários. Para isso, ele deve incorporar as particularidades do BDD para a descrição das funcionalidades, as quais podem melhorar o próprio entendimento das funcionalidades, uma vez que a visão do teste de aceitação está sendo incorporada.

O *Scrum Team* deve os testes automatizados para os cenários dos comportamentos elaborados pelo *Product Owner,* visando testes automatizados, para o *Sprint*, no servidor de Integração Contínua, que deve ser preparado inicialmente. Os testes incluem a verificação dos comportamentos implementados anteriormente, para garantir que suas características sejam preservadas. Uma vez especificados os testes, os comportamentos do software são codificados e testados. No final de um *Sprint*, o *Scrum Team* tem um produto para ser colocado em produção.

Do ponto de vista de tipos de teste, os usuários de métodos ágeis frequentemente utilizam a estratégia da pirâmide de teste, referenciada por Martin Fowler (SOLÍS; WANG, 2011XXXX), Essa estratégia foi também adotada no *Scrum*-BDD pois, para o constante *feedback*, é necessário que os testes sejam executados constante e rapidamente (figura 7).

Figura 7 — Pirâmide de teste.



Essa estratégia considera os testes de unidade, testes de serviço e teste através da interface de humano-computador (IHC). Os testes através da IHC é, em princípio, uma boa solução, pois permite testar o software a partir do comando na IHC. No entanto, como em geral a IHC contém software de terceiros, nem sempre é fácil realizar os testes em ferramentas automatizadas. A dificuldade decorre da necessidade de realizar configurações diferentes da ferramenta de teste para conjuntos de teste e também da obtenção de licenças de software de terceiros. Dessa forma, em muitos casos, o teste de serviços, sem o uso da IHC pode ser uma solução que agiliza os testes. Esse tipo de teste é realizado na camada intermediária de teste de serviços e é denominado de teste subcutânea, sendo realizado na camada imediatamente abaixo da camada de interface (SOLÍS; WANG, 2011).

Pode-se ainda comentar que o processo de teste automatizado tem, no preparo dos testes, a geração de especificação executável dos comportamentos do *software* e, pela sua própria natureza, passa a ser um documento dos comportamentos que estarão sempre atualizados.

A entrega frequente de produtos de alta qualidade dos *Sprints* faz com que cada versão tenha, em geral, um acréscimo de funcionalidades, sem impactar no

funcionamento da anterior. Isso faz com que o risco associado à liberação de uma nova versão fique reduzido significativamente e, no caso da necessidade de reverter às mudanças, o processo é facilitado. A entrega frequente também causa uma realimentação mais rápida do cliente, agilizando o desenvolvimento.

3.4 Aplicação das boas práticas do BDD no ambiente *Scrum*

Nesta seção, é feita a aplicação do *Scrum*-BDD na estrutura corporativa de uma empresa real de pagamentos digitais, denominado de empresa M, a qual é responsável por fornecer meios de pagamento para lojas virtuais.

Com expansão do negócio através da Internet surgiu, como desafios, a necessidade de criar uma forma para que o cliente possa pagar pelo produto ou pelo serviço de maneira não presencial, ou seja, sem que o cliente estivesse presente na loja no ato da compra. O resultado foi a instituição do pagamento digital, que é feito pela Internet, por meio de uma transação bancária ou por cartões de crédito. Dessa forma, o cliente tem a comodidade de pagar pelo pedido, independentemente de onde estiver, e a loja virtual, por sua vez, deve ter a segurança do recebimento do valor pago pelo cliente.

As plataformas de pagamento via Internet disponibilizam esse tipo de serviço, a fim de facilitar a realização de transações financeiras rápidas e seguras. As empresas, que fornecem essas plataformas, garantem às lojas virtuais uma disponibilidade de cerca de 99,7%. Isso significa que, em um dia, o *software* desse tipo pode ter uma indisponibilidade de cerca de 4 minutos e 19 segundos, em uma semana, 30 minutos e 14 segundos, em um mês, 2 horas e 11 minutos e, em um ano, 1 dia 2 horas 17 minutos e 50 segundos.

3.4.1 Descrição do ambiente de teste da empresa M

Dado este cenário, pode-se observar que o *software* de pagamento digital deve ser de alta disponibilidade e empresas que fornecem esse tipo de *software* devem possuir processos desenvolvimento que garantam a qualidade das transações financeiras. Em particular, os processos de teste e instalação devem ser feitos com planejamento cuidadoso, pois uma pequena indisponibilidade pode gerar um enorme prejuízo financeiro a essas empresas e às lojas virtuais que utilizam suas plataformas.

Atualmente, no mercado de pagamentos digitais, existem inúmeras opções de pagamentos disponíveis que permitem uma conexão direta com o autorizador da transação financeira. Por exemplo, se a loja virtual precisar aceitar Diners, Mastercard e Visa, basta integrar-se diretamente ao adquirente, usando seus protocolos, Cielo E-commerce, Redecard e outros adquirentes.

Da mesma forma, é possível, à loja virtual, conectar diretamente ao banco e fornecer, aos clientes, as opções de pagarem suas compras através de transferência bancária, boleto e, em alguns casos, através de financiamento. Este último tipo de serviço é provido, por exemplo, por Banco do Brasil, Banrisul, Bradesco e Itaú.

A empresa M está nesse mercado desde 2008 e fornece, em uma única integração, uma plataforma com diversos meios de pagamentos digitais, necessários para uma loja virtual.

É apresentado, a seguir, o ambiente de desenvolvimento de *software* da empresa M, relacionado com os testes.

A empresa utiliza a ferramenta *Cucumber* que permite descrever as funcionalidades do sistema a ser desenvolvido, através de uma linguagem específica de domínio, compreensível pelas pessoas da área de negócio. A descrição do comportamento é feita através de texto em linguagem natural, que é inserido na estrutura de programa, para posterior codificação. Como as funcionalidades podem ser descritas em português, essas descrições auxiliam na

documentação do software, por serem mais precisos que comentários inseridos posteriormente (WYNNE; HELLES,2012).

A empresa M utiliza *Scrum* e seu *Product Owner* é responsável por escrever as funcionalidades que são detalhadas e codificadas pelo *Scrum Team.*

Para controle de versão de código, a empresa optou por utilizar o Git, um sistema distribuído de código livre e aberto, para controle de versão de projetos de portes diversos. O Git é uma ferramenta amplamente utilizada no mercado e empresas como *Google, Facebook, Microsoft, Twitter, Netflix, Gnome, Eclipse,* entre outras utilizam no seu desenvolvimento de *software*.

O Git mostrou-se ser uma ferramenta importante para empresas que trabalham com diversas equipes no mesmo código do *software*, pois permite os desenvolvedores trabalharem em diferentes partes do software, sem quem um interfira o outro.

Para a aplicação do conceito de Integração Contínua, a empresa utiliza o GO – Continuous Delivery, um sistema de código aberto para integração contínua e gerência de versões, desenvolvido pela empresa *ThoughtWorks*.

O GO automatiza e contribui de maneira a agilizar o ciclo de testes e a gerar os artefatos compilados para a entrega contínua. A ferramenta realiza a execução de testes automatizados e, caso algum teste venha a falhar, dispara um e-mail para o *Scrum Team*, com os dados sobre as falhas ocorridas em algum teste. Outro recurso do GO é gerar os pacotes de versão para a instalação do software. A cada nova funcionalidade instalada no software, o GO dispara uma execução dos testes e, quando todos os testes forem executados com sucesso, o GO gera um pacote instalável da versão do *software,* considerada uma versão estável.

Após o GO ter gerado o pacote do *software,* a empresa M, ainda conta com o ambiente de homologação que, em geral, é muito semelhante ao de produção, sem, no entanto, realizar operações financeiras reais.

A versão estável do *software* é instalada no ambiente de homologação, para que o *Product Owner* realize seus testes para validar as funcionalidades em relação

aos requisitos. Uma vez que as novas funcionalidades adicionadas ao *software* foram validadas pelo *Product Owner*, o *software* encontra-se disponível para a instalação no ambiente de produção.

O sistema operacional dos servidores de instalação da empresa é Linux, devido a sua fácil integração com as tecnologias descritas anteriormente. O Linux também é um sistema operacional amplamente difundido, estável e seguro.

3.4.2 Aplicação das boas práticas de BDD na empresa M

Nesta seção, é descrito um experimento na empresa M, para analisar a inclusão das boas práticas de BDD no *Scrum* que já era utilizada pela sua equipe de desenvolvimento. A empresa é uma empresa de médio porte com cerca de 80 (oitenta) colaboradores sendo desses apenas 20 (vinte) envolvidos na parte de tecnologia da empresa. Inicialmente, vale salientar que empresa M vem tentando, por inúmeras vezes, aplicar métodos ágeis como *Scrum* à risca. Porém, nunca foi bem-sucedida, devido a diversos motivos, tais como falta de uma boa compreensão da equipe envolvida sobre o processo de testes, ou seja, o que era necessário testar, como e onde incluir as atividades de testes dentro do processo ágil. Além, disso, muitas vezes, existia a falta de uma visão mais clara sobre as novas funcionalidades que estão sendo adicionadas ao *software*.

Antes da implantação das práticas do BDD, a preparação da entrega de software na empresa M era feita de maneira manual e complemente dependente do desenvolvedor.

O desenvolver recebia as tarefas de maneira imprecisa, sem uma descrição detalhada da funcionalidade e, por muitas vezes, o entendimento ficava sujeito à sua interpretação, e atribuía significado não esperado para a nova funcionalidade.

O processo de execução dos testes era totalmente dependente do desenvolvedor, ou seja, era necessário que o desenvolvedor definisse os testes e o ambiente de execução de testes ficava no seu computador, onde as alterações feitas eram testadas. Isso mostrou ser um processo falho, pois caso o

desenvolvedor não tivesse o cuidado necessário com o *software*, os testes eram realizados de forma adequada e o desenvolvedor não tinha a motivação para continuar evoluindo os testes.

Após a implementação da nova funcionalidade, o desenvolvedor ainda era o responsável por gerar o pacote do *software*, disponibilizando-o para a equipe de infraestrutura, que recebia esse pacote, sem nenhum tipo de versionamento do *software* e, manualmente, instalava nos ambientes desejados.

Por muito tempo a empresa M utilizou esse processo, completamente dependente de seus desenvolvedores; porém, com o crescimento da equipe, o processo complemente manual e dependente dos desenvolvedores começou a mostrar suas fraquezas. Por muitas vezes, foram instaladas versões de *software* em que, por falta de testes planejados, outras funcionalidades pararam de funcionar ou tiveram seu comportamento alterado. Neste processo, ainda existia uma grande dificuldade para voltar para versões estáveis do *software*, pois não existia controle de versões, dificultando a identificação da versão que estava sendo instalada no ambiente de produção.

Com isso pode-se observar que o processo anterior era muito frágil e suscetível a erros, gerando grandes problemas para a empresa M.

Para resolver esses problemas, as equipes de desenvolvimento optaram pela utilização do *Scrum*, para gerenciar melhor a inclusão das novas funcionalidades, decorrentes da crescente demanda de mercado. Porém, apenas utilização do processo *Scrum* não foi suficiente para atingir as metas de qualidade, pois as equipes não conseguiam elaborar bons cenários de testes para as funcionalidades. Sendo assim, ao executar as próximas iterações do processo, as equipes de desenvolvimento observaram que as funcionalidades existentes foram comprometidas devido às mudanças atuais.

Esse cenário contribuiu para que fossem entregues versões com funcionalidades comprometidas do *software* ou causando impacto em outras funcionalidades já existentes, e entregando um *software* com erros ou comportamentos não esperados.

A introdução do *Scrum*-BDD na empresa M é ilustrada através da descrição parcial dos passos realizados em um *Sprint* de um desenvolvimento. Para o ponto

de partida, considera-se que o *Product Owner*, o *Scrum Master* e o *Scrum Team* já tenham realizado a cerimônia de definição de *Sprint*. Para a maior clareza da aplicação, descreve-se o desenvolvimento da funcionalidade de identificação de bandeira de cartão de crédito, que é adicionada ao *Checkout* de Pagamentos da empresa M.

O *Checkout* é um módulo do *software* da empresa M, para o qual o usuário da loja virtual é direcionado para fechar a compra de seus itens. De forma resumida, corresponde ao procedimento que vai desde o cadastro do usuário, caso ele ainda não seja cliente, passando pela seleção do endereço de entrega até o cálculo de frete, concluindo com o pagamento dos itens.

Inicialmente foram escritas as estórias da funcionalidade do *Sprint* pelo *Product Owner,* que é o especialista do domínio de pagamentos digitais. O *Product Owner* é responsável por escolher as estórias que agreguem valor ao cliente. Duas delas foram selecionadas para ilustrar a aplicação das boas práticas:

> Eu como empresa M, gostaria de identificar a bandeira do cartão de crédito, quando o usuário informar o número.

> Eu como empresa M, gostaria de não aceitar cartão com a data de expiração já vencida.

Feito isso, o *Product Owner* escreveu a funcionalidade correspondente às estórias.

Como empresa M utiliza a ferramenta Cucumber, a descrição das funcionalidades é feita em linguagem natural, ou seja, em linguagem que o especialista do domínio e os desenvolvedores possam compreender mais facilmente. As Figuras 8 e 9 apresentam os documentos de funcionalidade das estórias selecionadas, na forma recomendadas pelo BDD.

Figura 8 — Documento de funcionalidade identificação de bandeira.

```
Funcionalidade: Identificação de bandeira de cartão
  Como um intermediador de pagamento
  Com o objetivo de melhorar a conversão do checkout
  Eu devo identificar a bandeira do cartão do crédito

  Cenário: Identificando uma bandeira de cartão válido
    Dado que eu esteja realizando um pagamento
    Quando eu seleciono "Cartão de Crédito" como forma de "pagamento"
    E eu preencho "Número de Cartão" com "<cartão>"
    Então eu devo ver "<bandeira>"

  Cenário: Cartão não identificado
    Dado que eu esteja realizando um pagamento
    Quando eu seleciono "Cartão de Crédito" de "Cliente"
    E eu preencho "Número de Cartão" com "0000"
    Então eu deveria ver "Cartão de Crédito Inválido"

    Exemplos:
    | cartão              | bandeira   |
    | 4012001037141112    | Visa       |
    | 5453010000066167    | MasterCard |
    | 6362970000457013    | Elo        |
```

Figura 9 — Documento de funcionalidade validação de data.

Funcionalidade: Não aceitação de cartão com a data de vencimento expirada
Como um intermediador de pagamento
Com o objetivo de melhorar a conversão do *checkout*
Eu não devo aceitar cartão a data de expiração vencida

Cenário: Cartão com data de expiração vencida
Dado que eu esteja realizando um pagamento
E que hoje seja dia 22/01/2015
Quando eu seleciono "Cartão de Crédito" como forma de "pagamento"
E eu preencho "Número de Cartão" com "4012001037141112"
E eu preencho "Expiração do Cartão" com "12/2014”
Então eu deveria ver "Cartão de Crédito vencido"

Cenário: Cartão dentro da data de expiração
Dado que eu esteja realizando um pagamento
Quando eu seleciono "Cartão de Crédito" de "Cliente"
E eu preencho "Número de Cartão" com "4012001037141112"
E eu preencho "Expiração do Cartão" com "12/2020”
Então eu deveria conseguir prosseguir no *checkout*

As funcionalidades 8 e 9 contém as informações necessárias para que o *Scrum Team* comece a preparar seus testes e sua implementação. Caso ainda apareçam mais cenários de testes, o *Product Owner* modifica o arquivo de funcionalidade, adicionando os novos cenários, sem o comprometimento do trabalho em andamento.

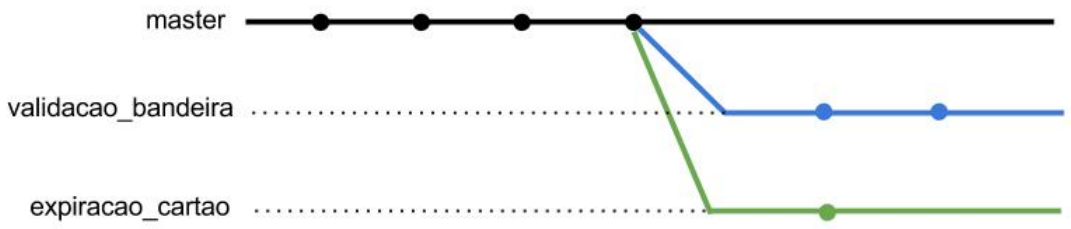
Os documentos de funcionalidade também têm a utilidade como documentação do *software*, pois descrevem o seu comportamento e os seus cenários.

Uma vez que os documentos de funcionalidade estavam prontos, o *Scrum Team* iniciou o preparo dos testes automatizados e a codificação da funcionalidade.

A empresa M utiliza, como controle de versão de código, a ferramenta *Git,* que faz o papel de unir as partes do projeto (código fonte), de modo que cada membro da equipe faça uma parte e, utilizando este sistema é possível unir, as diferentes alterações de cada desenvolvedor em um único código final. Para isso o Git trabalha com o conceito de *branches*, ou seja, uma ramificação do código que possibilita os desenvolvedores continuarem evoluindo o código sem que outro desenvolvedor interrompa, e então eles podem unir o código em um processo chamado *merge*, quando as suas partes estiverem funcionando.

Para cada nova funcionalidade do *Sprint*, cria-se em uma nova *branch*, cujo nome deve ser próximo ao da funcionalidade que será desenvolvida. Assim, uma nova *branch* é criada a partir do trecho de código principal, o *branch master* que corresponde à versão estável do *software*. Os testes da integração contínua são realizados sempre a partir da *branch master*. Na figura 10, pode-se observar os *branches* expiração_cartao e validação_bandeira, correspondentes às duas funcionalidades descritas anteriormente.

Figura 10 — Versionamento de código.



Cada funcionalidade é implementada por um desenvolvedor separadamente e a ferramenta *Git* auxilia na distribuição de código entre as *branches*. Cada desenvolvedor cria uma nova *branch*, como também cria um novo ponteiro para que possa se mover em relação à *branch master*. A ferramenta Git permite que dois desenvolvedores trabalhem em funcionalidades diferentes, porém, nos mesmos trechos de código e evitando conflitos das funcionalidades, uma vez que cada um deles está trabalhando em uma *branch* separadamente. Dessa forma, é possível

que cada desenvolver trabalhe em seu trecho de código, sem que um interfira na funcionalidade do outro.

Após a geração dos documentos de funcionalidade, é feita a preparação de testes automatizados pelo *Scrum Team*, utilizando a ferramenta *Cucumber;* os documentos de funcionalidade foram interpretados e foram gerados os documentos de implementação de testes. Nesse documento, são definidos os *steps* pelo desenvolvedor, os quais correspondem aos métodos ou funções de programas orientados a objetos ou procedurais. Cada *step* pode ter zero ou mais argumentos e a sua definição começa com uma preposição ou advérbio (Dado, Quando, Então, E, Mas).

Os documentos de implementação de testes das funcionalidades de validação data de expiração do cartão de crédito e validação de bandeira de cartão de crédito encontra-se respectivamente nas Figuras 11 e 12.

Figura 11 — Documento de implementação de testes – funcionalidade validação de bandeira de cartão de crédito.

```
Dado /^que eu esteja realizando um pagamento$/ do
  visit "/checkout"
end

Quando /^eu seleciono "Cartão de Crédito" como forma de "pagamento"$/ do
  click_button "creditcard"
end

E /^eu preencho "Número de Cartão" com "<cartão>"$/ do
  fills_in "creditcard_number", with: "4012001037141112"
end

Então /^Então eu devo ver <bandeira>$/ do
  response.should have_tag("p", text: /Visa/)
end
```

Figura 12 — Documento de implementação de testes – funcionalidade validação data de expiração do cartão de crédito.

```
Dado /^que eu esteja realizando um pagamento$/ do
  visit "/checkout"
end

E /^que hoje seja dia 22/01/2015$/ do

end

Quando /^ eu seleciono "Cartão de Crédito" como forma de "pagamento"$/ do
  click_button "creditcard"
end

E /^eu preencho "Número de Cartão" com "4012001037141112"$/ do
  fills_in "creditcard_number", with: "4012001037141112"
end

E /^eu preencho "Expiração do Cartão" com "12/2014"$/ do
  fills_in "creditcard_expiration_date", with: "12/2014"
end

Então /^eu deveria ver "Cartão de Crédito vencido"$/ do
  response.should have_tag("p", text: /Cartão de Crédito expirado/)
end
```

O documento de implementação de testes contém a especificação para os testes automatizados para cada um dos cenários de uma funcionalidade. Conforme a recomendação do BDD, primeiro deve-se criar um teste que não passa, para

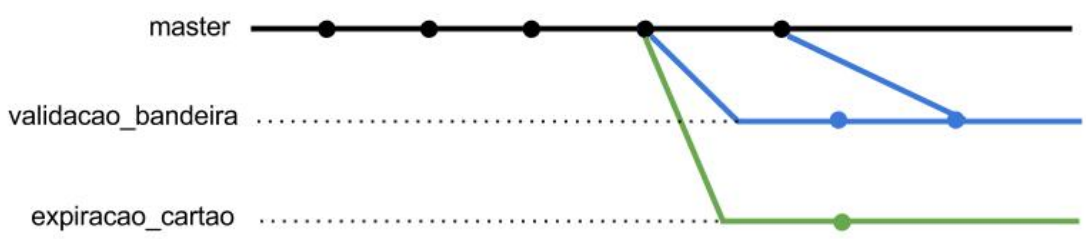
depois criar a implementação da funcionalidade, seguindo o mesmo ciclo recomendado por Kent Back (BECK, 2002). Então, ao realizar os testes, se o desenvolvedor receber a informação de que os testes, referentes a pagamento com o cartão de crédito, não estão passando, é porque a implementação final do código ainda não foi elaborada.

Com o entendimento dos diversos cenários, o desenvolvedor está mais seguro para criar código necessário para atender a um requisito. Uma vez que, o desenvolvedor tenha realizado a implementação necessária, os respectivos testes realizados e a funcionalidade é considerada concluída, o código correspondente deve ser inserido na *branch master.*

Para isso, o desenvolvedor deve trazer a versão do *branch master* para seu *branch* e realizar as mudanças necessárias, para obter a nova versão. A ferramenta *Git* (CHACON, 2014) fornece, ao desenvolvedor, o recurso de *merge* de código entre os *branches*, para que não haja conflito com os outros desenvolvedores que estão trabalhando no mesmo *software*.

O processo de *merge* está representado pela figura 1*3.* O desenvolvedor que estiver com uma versão do código de validacao_bandeira concluída, solicita o processo de merge e, caso não ocorram conflitos, pode liberar a sua funcionalidade.

Figura 13 — Merge do master para validação_bandeira



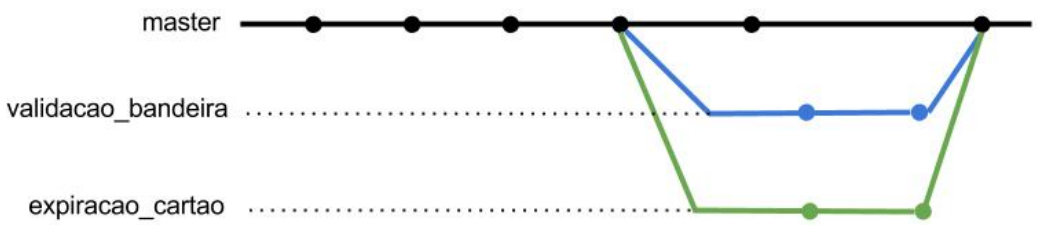
A operação de *merge* é representada pela linha à direita que une a *branch* da validação_bandeira para a *branch master*.

Após o *merge,* a nova funcionalidade fica disponível para ser incorporada nos testes no servidor de Integração Contínua.

Quando o código desenvolvido nas *branches* é colocado na *branch master*, as *branches* correspondentes devem ser removidas, encerrando, dessa forma o seu ciclo de vida.

A figura 14 mostra o encerramento do ciclo de vida das *branches*, em seu processo de desenvolvimento.

Figura 14 — Encerramento de branch



Com o encerramento da funcionalidade no processo de *merge* do Git, é iniciado o processo de integração contínua. Para a empresa M, o servidor *GO* (CHACON, 2014) realiza operações, como a execução de teste e empacotamento do *software*.

A execução dos testes é realizada na empresa M através da utilização da ferramenta *Go Continuous Delivery* no servidor de Integração Contínua. Essa ferramenta é integrada com a ferramenta *Git* e, assim, quando ocorre mudança de código na *branch master* do *Git*, isso é identificado pelo *Go* que inicia seu processo.

Sendo assim, a identificação automática da mudança do *software* causa a execução automática dos testes pelo *Go*; isso garante que as funcionalidades acrescentadas ao *software,* não tenham alterado os comportamentos já existentes.

Para a validação de software, o *Go* realiza os seguintes passos:

- **BuildApp:** o *Go* baixa o código fonte da ferramenta *Git* e, se necessário, realiza a compilação;
- **TestUnit:** execução de todos os cenários de testes existentes dentro do software;
- **VendorizeGems:** realiza o empacotamento das bibliotecas de terceiros utilizado pelo software;

- **CodeQualityAndSecurity:** realiza a análise estática do código, onde o foco é encontrar possíveis vulnerabilidades de segurança;
- **PackageApp:** realiza o empacotamento de uma versão estável do *software*;
- **PushAppPackage:** envia a versão de *software* estável para o servidor de repositórios do *software*, disponibilizando-o para instalação nos servidores de produção.

A empresa M considera que uma versão estável de *software* é gerada quando o código passa por todos os passos sem falhas.

Caso venha a acontecer alguma falha, em qualquer passo do processo, o Go não disponibiliza a versão de *software* e comunica, através de um e-mail para *Scrum Team*, alertando a instabilidade do *software*. A nova versão do *software* não é liberada até que a falha encontrada seja corrigida, garantindo apenas que a última versão estável do *software* esteja disponível para instalação nos ambientes.

Quando o *Go* disponibiliza a versão estável do *software*, o *Scrum Team* faz a instalação no ambiente de homologação, para avaliação final do *Product Owner*. Com a aprovação da nova funcionalidade pelo *Product Owner*, de acordo com o processo do *Scrum*, tem-se a versão pronta para ser instalada nos servidores de produção.

O processo de instalação do *software* em ambiente de produção ainda está em estudo na empresa M. Busca-se uma forma que seja segura e automatizada, através do uso de uma ferramenta apropriada.

O desenvolvimento de *software*, através das práticas de BDD no método *Scrum*, permite disponibilizar uma versão de *software* ao final de cada estória, agregando, assim, valor ao cliente de maneira contínua e validado pelos especialistas do domínio. Dessa forma, as entregas ocorrem continuamente, melhorando de fato desempenho do método *Scrum*, diminuindo o tempo das entregas entre *Sprints* e mostrando a evolução do *software* com novas funcionalidades.

3.5 Considerações Finais

As boas práticas do BDD no *Scrum* auxiliam a empresa M na entrega de *software* com qualidade e também contribuem para que as versões de *softwares* sejam entregues com maior frequência no ambiente de produção. Enfatiza-se que houve uma mudança de comportamento na sua equipe de desenvolvimento, o que permitiu aplicar as boas práticas com o sucesso. O envolvimento e a interação das pessoas da equipe de desenvolvimento causaram o sucesso desse experimento.

O processo ainda está em fase de amadurecimento, porém sua importância já foi reconhecida na empresa, de maneira que novas funcionalidades estão sendo concebidas através do seu uso.

4. CONSIDERAÇÕES FINAIS

Este capítulo descreve as conclusões obtidas com este trabalho, suas principais contribuições e recomendações, bem como os trabalhos futuros.

4.1 Conclusões

O primeiro princípio do Manifesto Ágil apresenta diz que “Nossa maior prioridade é satisfazer o cliente através da entrega contínua e adiantada de *software* com valor agregado”.

A estratégia do desenvolvimento ágil de *software* é possibilitar ao cliente avaliar o *software* em desenvolvimento e agregar valor ao negócio. Para isso, é fundamental planejar a entrega contínua de partes prontas do *software*, para que o cliente avalie se a funcionalidade atende as suas necessidades, como também, antecipar ao cliente os benefícios do software.

O BDD, como uma prática do desenvolvimento ágil, transforma uma ideia para um requisito que possa ser implementado e testado para a produção simples e eficaz, desde que o requisito seja específico o suficiente para que todos compreendam.

Também vale à pena salientar que não há muito tempo para a integração e a entrega de valor para o cliente, visto que o desenvolvimento ágil permite mostrar o que está ou não funcionando durante sua construção e obter novos recursos com *feedback* rápido, isso torna o ambiente corporativo mais colaborativo no ciclo de desenvolvimento.

O cliente hoje procura resultados e investir no desenvolvimento de *software*, que seja durável, flexível e, sobretudo, de fácil manutenção, ou seja, a entrega do *software* funcionando o mais breve possível.

Sendo assim pode-se considerar que o trabalho atingiu parcialmente o objetivo, uma vez que a utilização do scrum-BDD auxiliou ao desenvolvedor

compreender melhor as regras de negócio e melhorar a qualidade do *software* entregue. O uso de histórias e cenários deixou claro para desenvolvedor e *Product Owner* o que realmente era necessário a ser desenvolvido. Porém os testes de aceitação por muitas vezes demandam mais tempo para serem executados e como foi visto anteriormente é importante que eles possam ser executados rapidamente.

4.2 Contribuições

Atualmente, o uso da Internet possui uma demanda caracterizada por acesso a serviços *online* e buscas com acesso de forma rápida e com qualidade.

O atendimento de tais expectativas torna-se uma demanda necessária para o desenvolvimento de *software* e, neste sentido, o grande desafio é a operacionalização de uma série de pesquisas existentes na área. O presente trabalho busca ser um guia de estudo e boas práticas de BDD, dentro de processo *Scrum*, em uma prática de integração e entrega contínua, no qual, espera-se contribuir para a descoberta de elementos, que desencadeiem processos para o desenvolvimento ágil de *software*.

Para isso, apresenta-se uma forma de implantação de uma metodologia ágil focada em comportamento do *software* em um ambiente corporativo e mostra-se um exemplo. Através das práticas do BDD em um processo de entrega contínua, com aplicação do conjunto das técnicas no ambiente de desenvolvimento da empresa M, observa-se como o *software* é entregue e validado constantemente dentro do processo. Trata-se de uma prática na qual os testes são escritos antes da implementação final do código, permitindo-se aumentar a qualidade do código produzido.

Sendo assim, a principal contribuição do presente trabalho foi a aplicação do processo definido na empresa M e, principalmente, a mudança cultural ocorrida dentro da empresa, onde todos os interessados passaram a se envolver para criar o *software* final de qualidade.

4.3 Trabalhos futuros

O processo foi aplicado em uma empresa de médio porte, o que facilitou a aplicação. Como trabalho futuro pode-se avaliar identificar como o mesmo processo pode ser aplicado dentro de grandes empresas, com diversas equipes atuando sobre o mesmo *software*.

Outro trabalho futuro, principalmente para a empresa M, é identificar o processo de instalação automatizada, da instalação do *software* em seus devidos ambientes para evitar falhas na composição de uma versão de *software*. Ainda, muitos tópicos podem ser explorados para trazer melhorias e maturidade da entrega contínua e surgir uma continuação para esse assunto ainda pouco explorado.

REFERÊNCIAS

AVRAN, Abel & MARINESCU, Floyd. *Domain-Driven Design Quickly*. Lulu.com, 2007.

CHELIMSKY, David & ASTELS, Dave & HELMKAMP, Bryan & NORTH, D. & DENNIS, Zach & HELLESOY, Aslak. *The RSpec Book*: *Behaviour Driven Development with RSpec, Cucumber, and Friends*. Pragmatic Bookshelf, 2009. 21-52p.

DAN North & Associates. *Introducing BDD*. Disponível em: http://dannorth.net/introducing-bdd. Março, 2006. Acesso em 18 novembro de 2014.

FOWLER, Martin. *Continuous Integration*. 2000. Disponível em <http://martinfowler.com/articles/continuousIntegration.html>. Acesso em 3 junho 2015.

FOWLER, Martin. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, 1999. 63-89p.

HUMBLE, Jez & FARLEY, David**.** *Entrega contínua: Como entrega de software de forma rápida e confiável.* São Paulo, São Paulo: Bookman Companhia Ed, 2013. 11-69p ;187 – 222p

KENT, Beck. *Test Driven Development:* By Example, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2002. 20-81p.

KENT, Beck & BEEDLE, Mike & BENNEKUM, Arie Van & COCKBURN, Alistair & CUNNINGHAM , Ward & FOWLER, Martin & GRENNING, James & HIGHSMITH, Jim & HUNT, Andrew & JEFFRIES, Ron & KERN, Jon & MARICK, Brian & MARTIN, Robert C. & MELLOR, Steve & SCHWABER,Ken & THOMAS & Dave, Jeff Sutherland. *Manifesto for agile software development*. Disponível em http://agilemanifesto.org/, 02 2001. Acesso em 12 março 2015.

KNIBERG, H. *Scrum and XP from the Trenches*. InfoQ, 2007. Disponível em: <http://www.infoq.com/minibooks/scrum-xp-from-the-trenches>. Acesso em junho de 2015.

LAZAR, Loan & MOTOGNA, Simona & PÂRV, Brazil. *Behaviour-Driven Development of Foundational UML Components*. 2010.

PRESSMAN, Roger S. *Engenharia de software: Uma abordagem profissional*. Ed 7ª. São Paulo: Editora Bookman, 2011. 53 – 104p.

PUFAL, Nicholas & VIEIRA, Juraci. TRÊS FALÁCIA DO BDD. Disponível em <http://www.thoughtworks.com/pt/insights/blog/3-misconceptions-about-bdd>. Dezembro de 2013. Acesso em 8 julho de 2015.

SCHWABER, Ken & SUTHERLAND, Jeff. *Um guia definitive para o Scrum: As regras do jogo.* Disponível em:

<http://www.scrumguides.org/scrumgruides/v1/Scrum-guide-Portuguese-BR.pdf>. Junho de 2013. Acesso em julho de 2015.

SHORE, J.; WARDEN, S. *The Art of Agile Development*. O'Reilly, 2007. 255-261p.

SOLÍS, Carlos & WANG, Xiaofeng. *A study of the Characteristics of Behaviour Driven Development.* 2011 37th euromicro Conference on Software Engineering and Advanced Applications.

SMART, John Ferguson. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle.* Editora Manning, 2014. 110 - 250p.

WYNNE, Matt & HELLESøy, Aslak. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers.* Publisher: Pragmatic Bookshelf, 2012. 3-8p.

CHACON, Scott & STRAUB, Ben. Pro Git Book. Editora Apress; 2nd ed. 2014. 43 – 141p