Especificação e Automação Colaborativas de Testes utilizando a técnica BDD

Rafael Chiavegatto[1], Vanilton Pinheiro[1], Andréia Freitas Vieira[1], João Clineu[1], Erbett Hinton Oliveira[1], Ellen Barroso[1], Alexandre Amorim[1], Tayana Conte[2]

[1]FPF – Fundação Desembargador Paulo Feitoza – Manaus – Amazonas - Brasil

[2]USES – Grupo de Usabilidade e Engenharia de Software, Universidade Federal do Amazonas (UFAM), Manaus – Amazonas – Brasil

{rafael.chiavegatto, vanilton.pinheiro, avieira, joao.junior, erbett.oliveira, ellen.barroso, acruz}@fpf.br tayana@icomp.ufam.edu.br

Abstract. *Testing is an essential activity in the process of software development. However, its manual implementation has a high cost and often the customer require rapid feedback, compromising product quality. One way to work this problem is through the use of BDD (Behavior-Driven Development), together with the web test automation framework Selenium WebDriver. This paper presents an experience report on use of this technique to automate testing in a software development, enabling collaboration among team members, increasing of software quality and giving a quick response to the customer.*

Resumo. *Teste é uma atividade indispensável no processo de desenvolvimento de software. Porém, sua execução manual tem um alto custo e, muitas vezes o cliente requer um feedback rápido, assim comprometendo a qualidade do produto. Uma forma de atuar nesse problema é por meio do uso do BDD (Behavior-Driven Development), em conjunto com o framework para automatização de testes web Selenium WebDriver. Este artigo apresenta um relato de experiência da utilização dessa técnica com automatização de testes no desenvolvimento de um software, permitindo a colaboração entre os integrantes do time, aumentando a qualidade do software e dando uma resposta ágil ao cliente.*

1. Introdução

De acordo com Bartié (2002), a qualidade de *software* visa garantir a conformidade de processos e artefatos produzidos, eliminando os defeitos que são encontrados em todas as fases do processo de desenvolvimento. Metodologias de desenvolvimento ágeis como Scrum, recomendam que todas as pessoas envolvidas em um projeto trabalhem controlando a qualidade do produto todos os dias e a qualquer momento, pois se baseiam na ideia de que prevenir defeitos é mais fácil e barato que identificá-los e corrigi-los *a posteriori* [Schwaber e Beedle, 2001].

O foco da etapa de testes é garantir que o *software* atenda aos requisitos especificados [Softex, 2012]. Para a execução dessa atividade, algumas atividades se fazem necessárias, dentre as quais podem ser mencionadas: especificação de casos de testes, execução dos casos de testes especificados, registro de falhas, melhorias e reporte dos resultados obtidos. Cada uma dessas atividades necessita de um esforço grande e,

dependendo do tipo de *software* testado, requer que os mesmos testes sejam executados em diferentes configurações, aumentando ainda mais esse esforço. Muitas vezes o cliente desse *software* exige um *feedback* rápido, comprometendo a qualidade e a satisfação desse cliente.

Além disso, usualmente coloca-se toda a responsabilidade pela atividade de teste somente no analista de testes, sobrecarregando esse papel. Certas falhas em fluxos básicos são encontradas somente na fase de testes, porém poderiam ter sido previamente identificadas ainda na fase de desenvolvimento. Desse modo, o analista de testes, ao invés de focar em testes com cenários bem elaborados e alternativos, foca em garantir que os fluxos principais funcionam conforme o previsto.

Visando atuar nos problemas de alto custo para a execução de testes manuais e sobrecarga de atividades do analista de testes, a empresa do estudo em questão buscou mecanismos para agilizar a execução desses testes, bem como comprometer todo o time de um projeto com a qualidade do que é desenvolvido. O mecanismo adotado foi a utilização combinada da técnica BDD (Behavior-Driven Development), do *framework* orientado a comportamento Cucumber-JVM, da ferramenta de automatização Selenium Webdriver e do *framework* de testes JUnit, provendo benefícios tais como: otimização do tempo gasto para a execução de testes, mudança de foco da execução manual de testes pela busca por cenários mais elaborados, melhorias na comunicação dos envolvidos no projeto, colaboração e a compreensão dos envolvidos sobre o negócio do *software* que está sendo desenvolvido.

As próximas seções deste artigo estão organizadas da seguinte forma: a Seção 2 apresenta uma visão geral da técnica BDD e ferramentas de apoio à automatização de testes. A Seção 3 descreve o estudo de caso, contextualizando o problema e apresentando o processo adotado e os resultados alcançados. Por fim, a Seção 4 descreve as considerações finais e trabalhos futuros.

2. Automatização de Testes

Atualmente a automação de testes tem se tornado uma atividade de grande importância em projetos de teste de *software* [Caetano, 2007]. Segundo Maldonado *et al.* (2007), a técnica de automação é voltada principalmente para melhoria da qualidade, baseando-se fortemente na teoria de teste de *software*, para aplicar as recomendações dos testes manuais nos automatizados.

Apesar de apresentar vários benefícios, sendo o principal deles a garantia de qualidade de um *software*, com baixo esforço, ainda há uma resistência de seu uso devido ao alto custo para especificar e manter os códigos de testes. Porém, o esforço atualmente consumido com a correção de falhas pode ser investido na prevenção, através da automação de testes. Para apoiar esse processo, existem várias técnicas, *frameworks* e ferramentas, tais quais o Behavior-Driven Development (BDD) [North, 2006], Cucumber [Wynne e Hellesoy, 2012], Selenium [Gonçalves, 2011] e JUnit [Hunt e Thomas, 2003] que são descritos a seguir.

2.1. BDD

O *Behavior Driven Development* (BDD) é uma técnica de desenvolvimento ágil que foi concebida em 2003 com o intuito de envolver e incentivar a colaboração entre os envolvidos do projeto [North, 2006]. Seu foco é direcionado para o comportamento do

software, através de uma linguagem natural, proporcionando uma comunicação entendível entre os envolvidos no projeto, minimizando falhas de comunicação entre os membros do projeto. Além disso, o uso da técnica facilita a compreensão do que está sendo desenvolvido, estimula a contribuição na especificação e atualização dos requisitos do sistema.

Atualmente, existe uma grade diversidade de *frameworks* que amparam o uso do BDD em várias linguagens de programação [North, 2006]. Para a elaboração desse artigo foi utilizado o Cucumber –JVM [Wynne e Hellesoy, 2012].

2.2. Cucumber-JVM

O Cucumber-JVM é um *framework* de desenvolvimento orientado a comportamento utilizado para a especificação de cenários de testes, através de uma linguagem natural [Wynne e Hellesoy, 2012]. Seu objetivo é possibilitar uma rápida e fácil compreensão entre os envolvidos no projeto sobre as necessidades do cliente.

De acordo com Wynne e Hellesoy (2012), o Cucumber foi desenvolvido originalmente para a linguagem Ruby, porém, havia a necessidade de escrever os *steps* (passos) em linguagem Java para atender especificações de determinados projetos. Para isso, foi criado o Cucumber-JVM que é uma implementação em JRuby (compatível com a Java Virtual Machine) do Cucumber original.

A especificação dos cenários é realizada em um arquivo texto, onde são descritas as pré-condições, passos e resultados esperados através de uma sintaxe, abstendo-se de qualquer termo técnico. A partir da execução desse arquivo, será verificado e validado o comportamento do *software* de acordo com a expectativa do usuário. De modo análogo, os cenários são elaborados orientados a comportamento.

As especificações são lidas e executadas a partir dos arquivos de texto, escritos em linguagem natural. Cada cenário é uma lista de passos interpretados pelo Cucumber. Esses arquivos devem seguir algumas regras de sintaxe básica. O nome dado para este conjunto de regras é Gherkin [Wynne e Hellesoy, 2012].

A especificação dos cenários possui as seguintes notações:

- *Feature* (Narrativa): descrição da estória.
- *Scenario* (Cenário): descrição do cenário para estória.
- *Given* (Dado que): são as pré-condições para executar o cenário.
- *When* (Quando): são os testes (passos) para execução do cenário.
- *Then* (Então): é o resultado esperado da execução dos passos.

A Figura 1 exemplifica a especificação de um cenário utilizando o *framework* Cucumber-JVM.

```
Feature: Como um usuário,
         Desejo acessar minha caixa de email do Gmail no site www.gmail.com
         Então devo ter acesso a minha caixa de e-mail.

Scenario: Acessar o site e efetuar login com usuário e senha corretos

    Given Estou no site "www.gmail.com"
    When Informo o login "joao" e senha "123456"
    Then Deve ser exibido minha caixa de e-mail
```

Figura 1 - Exemplo de Especificação de cenário de testes no Cucumber

A especificação é salva em um arquivo com extensão .feature, onde é realizado um mapeamento entre o que foi especificado e onde está implementado. A implementação também deve seguir alguns conceitos para que o arquivo .feature entenda que tudo o que foi especificado — em uma classe Java — para, finalmente ser executado. O mapeamento é exemplificado conforme a Figura 2.

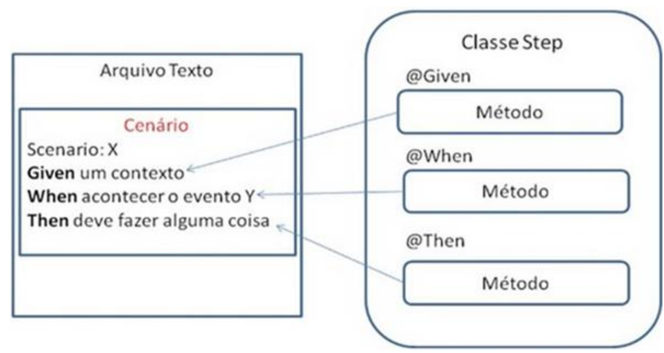

Figura 2: Mapeamento entre os cenários no Cucumber e as Classes Java

Para a execução do teste, é necessário utilizar uma classe de configuração onde se determina que os cenários especificados e implementados, devem ser executados como um teste *JUnit*. Juntamente a essa configuração, é determinada a emissão e o tipo de relatório, o diretório de onde estão especificados as estórias e o diretório onde estão implementados os métodos.

2.3. Selenium Web Driver

Conforme explicado na seção anterior, as funcionalidades do sistema são especificadas em cenários que representam o comportamento do *software* perante a expectativa do usuário, executadas de maneira automática por um framework de teste automatizado.

O Selenium WebDriver é um *framework open source*, que é utilizado para automatização de testes funcionais em aplicações web [Caetano, 2007]. Ao utilizar o Selenium WebDriver, os testes podem ser executados em diferentes navegadores e podem ser escritos em diversas linguagens de programação, tais como: Java, C#, Python, PHP, Perl, HTML e Ruby [Gonçalves, 2011]. Para executar os scripts gerados pelo Selenium WebDriver, é necessário a integração com o JUnit.

2.4. JUnit

O JUnit é um *framework open source* para utilização de testes unitários em Java, sendo possível criar classes de testes que podem ter um ou mais métodos para execução. Os testes podem ser executados sequencialmente ou de forma modularizada, dessa forma, os sistemas podem ser testados em partes ou de uma única vez.

Em 1998, Kent Beck e Erich Gamma desenvolveram o arcabouço JUnit para a linguagem Java, inspirado no SUnit [Hunt e Thomas, 2003]. Desde então a prática de testes automatizados vem sendo disseminada dentre as equipes de desenvolvimento de *software* adeptas à automatização. De forma geral seu funcionamento se dá através da verificação dos métodos de uma classe, constatando se os mesmos funcionam da maneira esperada, exibindo de forma visual o resultado da execução positiva ou negativa dos testes (*Passed or Fail*).

3. Estudo de Caso

Esta seção discute como foi feita a aplicação do *framework* de BDD (Cucumber –JVM) e o de automatização (Selenium WebDriver) em um estudo de caso de um *software* web que possui a finalidade de gerar conteúdo educacional à distância. A equipe que trabalhou nesse projeto contava com 07 desenvolvedores e 01 testador. Esse *software* tem como principais características ser executado a partir de vários navegadores, possibilitando, também, a realização de *upload* de áudio, imagem e vídeo em vários formatos. Existe uma funcionalidade principal que é comum ao sistema, ou seja, é reutilizada em vários módulos do *software*. Dessa forma, mudanças nesta funcionalidade impactavam em várias outras funcionalidades, o que fazia com que o esforço com as atividades de testes fosse alto. Adicionalmente, o cliente sempre requeria entregas rápidas.

Em função das características desse *software*, tornou-se necessária a tomada de ações que minimizassem o risco de não realizar as entregas ao cliente dentro dos prazos estabelecidos, e com a qualidade esperada. Dessa forma, foi feito um estudo de abordagens de automatização de testes utilizadas no mercado, e decidiu-se, após esse estudo, adotar a técnica BDD, juntamente com a automatização de testes.

Essa escolha ocorreu por BDD possibilitar a especificação de casos de testes em linguagem natural, os envolvidos no projeto podem contribuir para a especificação e com isso gerar um produto de qualidade. Adicionalmente, a manutenção nos *scripts* gerados é mais fácil, pois existe uma rastreabilidade entre a linguagem natural e a linguagem de programação utilizada para a automatização dos testes, reutilização da automação, minimizando o custo de automações futuras em outas partes do *software*. Foi utilizado o *framework* de BDD Cucumber –JVM para a especificação dos cenários de testes, e para a automação dos *steps* (passos), o Selenium WebDriver. O processo utilizado para adoção dessa técnica está representado na Figura 3.

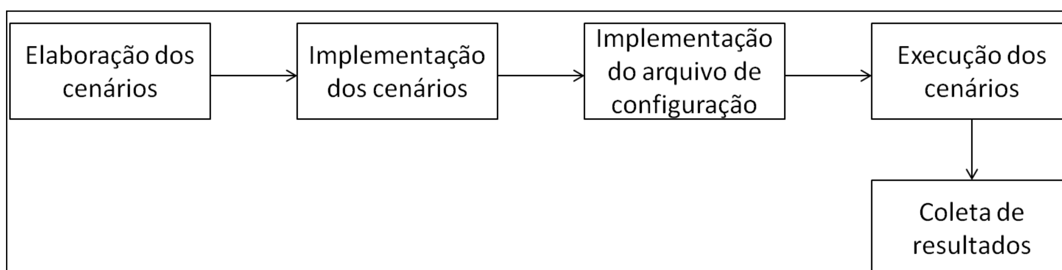

Figura 3 - Processo utilizado para aplicação do BDD

A primeira atividade realizada na técnica de BDD com automatização de testes foi o levantamento e especificação dos cenários (*.feature*), que foram agrupados em

estórias do usuário. Cada estória foi composta por um ou mais cenários elaborados baseados em comportamento utilizando o *framework* Cucumber-JVM.

A Figura 4 apresenta um dos cenários especificados para a funcionalidade principal do *software*. Nela, é possível visualizar a facilidade da escrita e leitura das regras de negócio desse *software*, uma vez que se utiliza uma linguagem natural voltada ao comportamento do sistema.

```
#Encoding: Cp1252
Feature: Como usuário do sistema x
        Desejo criar uma engine de infográfico Incluir e exlcuir salvando-as com um componente de cada vez
        E em cada formato caso exista.
        Então deve ser retornado a tela de listagem dos documentos.

Background:
    Given Estou no navegador Firefox
    And  Como usuário do sistema eConteudo desejo acessa-lo no endereço "http://win2008r2"
    And Estou na tela de listagem de documentos
    When Clico no menu "novo" e seleciono a opção "Infográfico"

Scenario: Criar uma engine de infográfico incluir e excluir um componente de caixa de texto
    Given Estou na tela correspondente a engine de "Infográfico"
    When Clico no elemento "caixaTexto"
    And Clico no botão alterar nome
    And Informo o nome "Engine_de_teste_1"
    And Clico no botão "Aplicar"
    And  Clico no link "Salvar"
    Then Apresenta a mensagem "Conteúdo salvo com sucesso."
    When Clico no botão "Fechar"
    And Verifico e seleciono a caixa de texto exibida no palco
    And Clico na imagem "excluir"
    Then Caixa de texto excluida do palco
    When Clico no elemento "caixaTexto"
    And  Preencho a caixa de texto com "Teste Caixa de Texto"
    And  Clico no link "Salvar"
    Then Apresenta a mensagem "Conteúdo salvo com sucesso."
    And  Clico no botão "Fechar"
    And  Clico no link "Listagem de documentos"
    And Clico no botão "Sim"
    And Estou na tela de listagem de documentos
```

Figura 4 - Cenário especificado no Cucumber

Após a elaboração dos cenários, foi realizada a implementação dos *steps* (passos) dos cenários especificados, utilizando o *framework* Selenium WebDriver. A Figura 5 apresenta um exemplo do mapeamento da linguagem natural para linguagem de programação, visando a implementação dos métodos.

```
@Given("^Estou no navegador Firefox$")
public void Estou_no_navegador_Firefox() throws Throwable {
    driver = new FirefoxDriver();
    driver.manage().window().maximize();
    driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
    navegador = "firefox";
}

@Given("^Como usuário do sistema x desejo acessa-lo no endereço \"([^\"]*)\"$")
public void Eu_como_usuário_do_sistema_eConteudo_desejo_acessa_lo_no_endereço(
        String txUrl) throws Throwable {
    driver.get(txUrl);
}

@Given("^Estou na tela de listagem de documentos$")
public void Eu_estou_na_tela_de_listagem_de_documentos() throws Throwable {
    assertTrue(driver.findElement(By.id("tabela-conteudo")).isDisplayed());
}

@When("^Clico no menu \"([^\"]*)\" e seleciono a opção \"([^\"]*)\"$")
public void Clico_no_menu_e_seleciono_a_opção(String txMenu, String txOpcao)
        throws Throwable {
    driver.findElement(
            By.xpath("//div[@id='btn-" + txMenu + "-conteudo']/a/span"))
            .click();
    driver.findElement(By.linkText(txOpcao)).click();
}
```

Figura 5 - Mapeamento da linguagem natural para a de programação

Após a implementação dos métodos dos cenários, foi implementada a classe de configuração que o próprio *framework* Cucumber -JVM fornece. O custo para essa implementação é de apenas alterar quais arquivos .feature devem ser executados, o

caminho onde estão os *steps* implementados e o formato que será gerado o relatório, como mostra a Figura 6.

```
package br.fpf.runner;

import org.junit.runner.RunWith;
@RunWith(Cucumber.class)
@Cucumber.Options(format = { "html:target/cucumber" },
features = {"test//resources//Infografico//Firefox_Alterar_Inclusao_Exclusao_Componentes.feature"},
glue="br.fpf.glue")


public class TestRunnereConteudo{

}
```

Figura 6 - Arquivo de configuração no Cucumber

Elaborada a classe de configuração, foi utilizada a ferramenta JUnit, que permite executar a classe como um teste *JUnit* e gerar os relatórios que o próprio *framework* JUnit e Cucumber-JVM disponibilizam. No relatório, são indicados os métodos que passaram, os que falharam e os que não foram executados devido a falhas.

Após a realização desse estudo, conseguiu-se visualizar os seguintes benefícios no processo: redução do esforço de testes e maior interação entre os integrantes do time.Com relação à redução do esforço de testes, comparado aos testes manuais, a execução automatizada proporcionou redução em até 87% do esforço necessário para realizar todos os cenários previstos para o sistema. Um conjunto de 81 cenários de testes, com 2.734 passos, executados em 4 navegadores diferentes, demandava um esforço manual de 8 horas. Esses mesmos cenários foram automatizados utilizando a técnica mencionada nesse artigo, demandando um esforço de 1 hora. Com isso, os analistas voltaram a sua atenção para testes mais elaborados, fora do fluxo principal coberto pela automatização, diminuindo a incidência de defeitos.

No que diz respeito à interação entre os integrantes do time, a responsabilidade pela qualidade do *software* foi assumida por todos, uma vez que o analista de testes foi o responsável por especificar os cenários e o desenvolvedor por implementar os métodos. Além disso, antes de liberar uma versão para o analista de testes, os desenvolvedores executavam os testes automáticos, visando garantir que falhas de regras de negócios contempladas nos testes automatizados não seriam identificadas na fase de testes.

Os benefícios também se estendem em longo prazo, pois os *steps* são reaproveitáveis. É possível, para um novo cenário de testes, reutilizar *steps* de cenários anteriormente especificados.

4. Conclusões e Trabalhos Futuros

Este trabalho teve como objetivo principal relatar a experiência na aplicação da técnica de *Behaviour-Driven Development* (BDD) com o uso do *framework* orientado a comportamento Cucumber-JVM, da ferramenta de automatização Selenium Webdriver e do *framework* de testes JUnit. Por meio da integração dessas ferramentas, da escrita em linguagem natural e do reaproveitamento de passos, a tarefa de manutenção dos testes foi facilitada. Além disso, a responsabilidade pela qualidade do *software* foi assumida por todo o time, uma vez que o analista de testes foi o responsável por especificar os cenários e o desenvolvedor por implementar os métodos.

Com os resultados desse estudo, espera-se apoiar a indústria de *software* a adotar cada vez mais estratégias para automatização de teste, contribuindo desta forma para a redução do custo de teste, apoiando a qualidade do produto e melhorando a comunicação dentro dos projetos de *software*.

Trabalhos futuros incluem a implantação dessa metodologia nos demais sistemas desenvolvidos pela empresa. Para que tal mudança de paradigma seja efetiva, há a necessidade de alocação de uma infraestrutura dedicada aos testes com o BDD, liberando recursos para outras atividades. Outro ponto importante, tendo em foco as metodologias ágeis, é a adoção da prática de Integração Contínua, isto é, a execução automática de testes a cada nova mudança, dando uma resposta instantânea do resultado à equipe. Por fim, os conceitos podem ser aplicados às plataformas móveis, em conformidade com as tendências do mercado, aumentando a gama de produtos atendidos e agregando valor ao *software* desenvolvido, com consequente aumento do grau de satisfação dos clientes.

Referências

Bartié, A. (2002). "Garantia da qualidade de software: adquirindo maturidade organizacional". Rio de Janeiro: Campus, 291p.

Caetano, C. (2007). Automação e Gerenciamento de Testes: Aumentando a Produtividade com as Principais Soluções Open Source e Gratuitas. Rio de Janeiro, 185p.

Gonçalves, H.N. (2011). "Geração de Testes Automatizados utilizando o Selenium". Trabalho de Conclusão de Universidade de Pernambuco. Escola Politécnica de Pernambuco. Graduação em Engenharia da Computação.

Hunt, A., Thomas, D. (2003) "Pragmatic Unit Testing in Java with JUnit." The Pragmatic Programmers v.2. Disponível em: < http://books.google.com.br/books/about/The_Pragmatic_Programmer.html?id=5wBQEp6ruIAC&redir_esc=y> Acessado em 07.04.2013.

Maldonado, J., C., Delamaro, M., E., e Jino, M. (2007) "Introdução ao Teste de Software." Editora: Elsevier, Campus.

North, D. (2006). "Introducing BDD." Disponível em: <http:// http://dannorth.net/introducing-bdd/> Acessado em 04.04.2013.

Schwaber, K., Beedle, M. (2001). "Agile Software Development with SCRUM". Prentice Hall.

Softex (2012). "Guia Geral MPS.BR de Software". Disponível em: <http://www.softex.br/mpsbr/_guias/guias/MPS.BR_Guia_Geral_Software_2012.pdf>.

Wynne, M., Hellesoy, A. (2012) "The Cucumber Book: Behaviour-Driven Development for Testers and Developers." Editora: The Pragmatic Programmers.