UNIP

Wérllen Guimarães

UTILIZAÇÃO DA LINGUAGEM NATURAL: Melhoria no Processo de Automação de Testes em Softwares.

Limeira

2016

UNIP

Wérllen Guimarães

UTILIZAÇÃO DA LINGUAGEM NATURAL: Melhoria no Processo de Automação de Testes em Softwares.

Trabalho de conclusão de curso apresentado à banca examinadora da Faculdade de Ciências da Unip, como requisito parcial à obtenção do grau de Bacharel em Ciências da Computação sob a orientação do professor Mestre Sergio Eduardo Nunes.

Limeira

2016

Wérllen Guimarães

UTILIZAÇÃO DA LINGUAGEM NATURAL: Melhoria no Processo de Automação de Testes em Softwares

Trabalho de conclusão de curso apresentado à banca examinadora da Faculdade de Ciências da Computação da UNIP, como requisito parcial à obtenção do grau de Bacharel em Ciências da Computação sob a orientação do professor Mestre Marcos Gialdi.

Aprovada em __ de ____ de 201__.

BANCA EXAMINADORA

_____

Prof. Dr. Nome completo

_____

Prof. Me. Nome completo

_____

Prof. Esp. Nome completo

DEDICATÓRIA

Dedico este trabalho a toda minha família pelo suporte e apoio incondicional; A todos os professores do curso de Ciência da Computação pelos ensinamentos e orientações.

“A motivação é como alimento para o cérebro, você não pode ter o suficiente em uma refeição. A motivação precisa de recargas contínuas e regulares para nutrir a fundo e transformar um ser humano comum em um verdadeiro vencedor”.

(Peter Davies)

RESUMO

A utilização da linguagem natural em processos de automação de testes visando à melhoria na qualidade de softwares assume um papel fundamental para promover o ganho em qualidade do produto x tempo hábil para realização das tarefas. Com ela é possível que os ruídos de informações entre o profissional que possui o conhecimento das regras de negócios do software a ser automatizado e o profissional que fará todo o desenvolvimento da automação de testes diminuam de maneira considerável, deixando a comunicação mais limpa e clara. Onde também, a facilidade para reconhecimento de falhas em testes e/ou bugs capturados nestes testes de softwares venham a ser de maneira muito mais simples e eficiente.

Palavra-Chave: Automação de testes, linguagem natural, qualidade de softwares.

ABSTRACT

The use of natural language in test automation processes aimed at improving the quality of software assumes a fundamental role to promote the gain in product quality x time to perform the tasks. With it, it is possible that the information noises between the professional who has the knowledge of the business rules of the software being automated and the professional that will make the whole development of the test automation decrease considerably, leaving the communication cleaner and clearer. Where, too, the facility for recognizing failures in tests and/or bugs caught in these software tests will be much simpler and more efficient.

Key words: test automation, natural language, quality of software.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

BDD	Behavior Driven Development
TDD	Test Driven Development
CMM	Capability Maturity Model

SUMÁRIO

1. INTRODUÇÃO

A qualidade de software visa o desenvolvimento e evolução de uma aplicação onde a mesma deve abranger de maneira satisfatória todos os serviços esperados pelo usuário daquele software em si. Mas como saber se o que esta sendo entregue ao cliente é um produto de qualidade satisfatória ou não? Para isso foram desenvolvidas ao longo dos anos, várias normas e modelos para que essa tal qualidade de software pudesse ser medida e assim podendo ser atingida ou não dentro dos padrões daquelas regras a serem seguidas.

O foco principal para que seja implantado um processo de qualidade de software em uma empresa de desenvolvimento de sistemas, é que esse processo sirva para estabelecer, garantir e gerenciar o nível de qualidade dos produtos que a organização fornece.

Segundo Alexandre Bartié (2002, pag. 8) software com deficiência de qualidade pode acarretar as empresas:

> "As empresas já entenderam que fabricar softwares "não adequados", além de prejudicar a imagem da organização, aumenta significativamente os custos totais de desenvolvimento. Isso consome a rentabilidade dos projetos de software, além de ampliar os riscos de insucesso dos projetos existentes."
>
> "Na verdade softwares "não adequados" são sintomas da falta de controle do processo de desenvolvimento."

Com isso as grandes empresas de software buscam sempre estar no patamar mais atualizado a que diz respeito sobre metodologias, técnicas e ferramentas que auxiliem no processo de aprimoramento do nível de maturidade de seu processo de garantia de qualidade de software.

Bem como estar em um nível alto de maturidade faz com que as indústrias de desenvolvimento de sistemas possam atingir um grande ganho no requisito custo x benefício, pois com isso assume-se que a empresa possua um senso de gerenciamento do seu processo muito forte, tendo toda as fases do projeto muito bem documentadas, padrões identificáveis nestas documentações que ajudam na criação de novos documentos. Gerenciamento

de versões liberadas, contendo informações de evoluções realizadas em cada versão e bugs corrigidos. Documentos exemplificando a funcionalidade da aplicação bem como diagramas de banco de dados. Com isso a quantidade de retrabalho em defeitos e carências do software possa ser nulo ou quase nulo, não demandando assim mão de obra para esforços desnecessários.

Dentro desta visão adentra o conceito de testes realizados em sistemas. Imagine que sua empresa já possui uma aplicação no mercado e que esta aplicação já contenha vários clientes que a utilize. E por um destes clientes seja identificado um bug, falha ou defeito em seu sistema, acarretando na parada total de uma funcionalidade do software e que a partir daquele ponto, o usuário não consiga dar seqüência em seus processos realizados diariamente neste software. Sua empresa como responsável desenvolvedora do produto deve-se então implementar uma correção para aquele bug reportado. Com isso a correção é feita e automaticamente liberada para o cliente, na qual a mesma contém sucesso na correção do problema. Porém mais a frente esta correção faz com que haja um impacto negativo em outra funcionalidade dependente dela, acarretando assim em mais um defeito.

Para que isso não ocorra deve-se, antes da liberação de um pacote de correção e/ou evolução de uma funcionalidade do sistema, realizar testes para que seja assegurado que aquela correção não impacte negativamente em outros processos da aplicação. Testes são fundamentais e indispensáveis, pois só assim pode ser observados problemas na integridade da correção que não foram observados na fase de desenvolvimento.

Quando um desenvolvimento é mal sucedido onde o mesmo é perceptível na fase de testes, o responsável pelo teste deve elaborar uma documentação evidenciado o defeito encontrado para que assim a correção volte para a fase de desenvolvimento.

1.1 Objetivo

Esse trabalho tem como objetivo principal o intuito de mostrar os benefícios da utilização da linguagem natural no processo de automação de testes funcionais em softwares. E juntamente a esta perspectiva, propor a criação de uma linguagem natural padronizada, utilizando-se dela como base para auxílio e ajuda para que outros profissionais que utilizem esta linguagem criada pelo autor do trabalho consigam desenvolver de maneira rápida e eficiente a automação de sua aplicação.

Com isso a linguagem foi desenvolvida pensando em descrever de forma geral o funcionamento básico e intermediário de uma aplicação, como por exemplo a descrição de operações como incluir, consultar, alterar e excluir. Operações estas que são realizadas pela maioria dos softwares desenvolvidos. Logicamente para que ela se adéqüe aos requisitos de sua aplicação alguns ajustes deverão ser feitos, porém é garantido que ela abrangerá a maior parte de seu processo.

E dentre seus benefícios, é de alvo principal exibir o ganho em eficiência utilizando da linguagem natural, já que ela melhora a comunicação entre o arquiteto funcional do sistema e o profissional de automação de testes. Melhora também a eficiência em identificar em que pontos do teste a falha ou defeito da aplicação ocorreu, ou até mesmo a falha no desenvolvimento da automatização. Deixando de forma muita clara e evidente em qual funcionalidade do sistema isto se sucedeu. Facilitando a manutenção e correção do desenvolvimento da automação, deixando que esta fase venha a ser de forma muito mais intuitiva do que uma análise técnica do código fonte em que a automação fora desenvolvida.

1.2 Justificativa

Este trabalho visa auxiliar as pessoas que exercem a função de automatização de testes dentro das empresas.

No setor de realização de testes funcionais das companhias desenvolvedoras, em alguns casos é necessário que quando seja realizada a execução de um teste automatizado, um funcionário fique responsável por acompanhar este teste enquanto o mesmo se encontra rodando na aplicação, para que assim caso o teste venha a “quebrar”, aquele funcionário que esta acompanhando o teste possa identificar o exato momento em que o problema ocorreu e assim partindo deste princípio para que seja feita a manutenção do código fonte. Quando utilizo a expressão código fonte me refiro unicamente ao código fonte da linguagem de automação de teste e não a lógica de programação do software.

Com o emprego da linguagem natural isso não mais se fará necessário, já que a identificação do problema no teste realizado pode ser identificado por um funcionário analisando o retorno de informações que é gerada toda vez que é finalizada a execução de um teste automatizado.

Essas informações das quais se é gerado relatórios e gráficos com estatísticas de sucesso ou falha do teste. Podendo ser muito utilizado pelos gestores da área como base para dados significativos em resposta a documentação e catálogo da gestão do setor. Adquirindo um nível de maturidade apreciável na garantia de qualidade de software e gerando um ganho em custo x benefício.

1.3 Metodologia

O trabalho será desenvolvido se preocupando apenas com os testes funcionais, pois é com eles que garantimos que as funcionalidades da aplicação se encontrem em equivalência com as especificações de seu funcionamento correto. Uma das formas de realizar este tipo de teste é através do BDD (Behavior Driven Development), onde sua característica principal é que as funcionalidades do sistema possam ser descritas em linguagem natural.

Para que o BDD possa ser usado, existem no mercado várias ferramentas e aplicações que dispõem dessa usabilidade, porém neste trabalho iremos utilizar o Cucumber, pois nele as funcionalidades do software podem ser descritas em arquivos de texto.

Etapa 1: Instalação do plugin do Cucumber em sua ferramenta de desenvolvimento, sugiro que utilize o Eclipse, porém também funciona com o NetBeans (deve-se utilizar a linguagem JAVA).

Etapa 2: A configuração do Jenkins, que trata-se de uma ferramenta de integração contínua de projeto Java. Nele configuraremos a extensão Cucumber para que ao ser executado os testes automatizados o mesmo nos retorne em informações provenientes relatórios e estatísticas provenientes da linguagem natural.

Etapa 3: Desenvolvimento da linguagem natural, com os processos e funcionalidades adequados ao seu software em conjunto com a proposta de linguagem natural concebida pelo autor

2. PROCESSO DE GARANTIA DE QUALIDADE DE SOFTWARE

2.1 Definindo qualidade de software

Toda e qualquer atitude que se tome na fase de desenvolvimento de software será de extremo impacto ao adquirir seu produto final. Decisões tomadas erroneamente podem acarretar na entrega de um sistema que não atenda a expectativa criada pelo cliente, onde o mau funcionamento faça com que o sistema se torne dispensável e/ou não recomendado para posteriores clientes, tornando todo o processo falho de qualidade de software um aspecto relevante no futuro financeiro da empresa em questão.

Conforme é dada a definição de qualidade de software por Alexandre Bartié, (2002, pag. 16):

> “Qualidade de Software é um processo sistemático que focaliza todas as etapas e artefatos produzidos com o objetivo de garantir a conformidade de processos e produtos, prevenindo e eliminando defeitos.”

2.1.1 Dimensão da qualidade de software

Segundo Alexandre Bartié (2002, p.16):

> "Softwares mal testados provocam prejuízos enormes as organizações. Um simples erro interno do projeto poderá encadear requisições de compras desnecessárias, solicitar manutenções de equipamentos antes do período ideal, produzir estatísticas falsas de produtividade, distribuir rendimentos de forma desproporcional, entre outros. Os problemas podem afetar até a tomada de decisões de gerentes, diretores e acionistas da organização. São profissionais que estão apoiando-se nas informações de sistemas informatizados para minimizar riscos, direcionar esforços, promover investimentos, sempre com o objetivo de tornar a organização mais eficiente e rentável. A qualidade das decisões está intimamente ligada á qualidade da informações disponibilizadas pelos diversos sistemas organizacionais."

Com isso fica evidente que é impossível obter um aplicação que atenda aos requisitos de qualidade de software, se a empresa possui processos de desenvolvimento frágeis e deficientes. Podendo ser estabelecido duas dimensões fundamentais para que seja atingida a qualidade de software: a dimensão da qualidade do processo e da qualidade do produto (BIRTIÉ, 2002).

2.1.2 Dimensão da qualidade do processo

Quando nos situamos diante do desafio de garantir a qualidade de um software, estamos estabelecendo a cultura de não tolerância a erros, definindo a capacidade de inibição e impedimento de falhas, possibilitando meios que durante o ciclo de desenvolvimento possuam procedimentos que avalim sua qualidade, possibilitando assim e facilitando a identificação precoce de defeitos estes meios. (BIRTIÉ, 2002).

Um processo com padrões bem evidentes e definidos é de extrema importância para que a qualidade do produto final seja alcançada. Organizações que contenham uma documentação padronizada do processo saem na frente para atingir seus objetivos. Para isso é imprescindível que ela controle todas as modificações e alterações que são feitas quem cada nova versão liberada ou não do seu sistema, que podemos atribuir o nome de Builds. Identificar o que foi alterado e corrigido facilita a manutenção e identificação de

falhas e também ajuda para que falhas já corrigidas não voltem a acontecer com a subscrição de versões por cima de correções já realizadas.

Documentações padrões também são importantes para que possa ser requisitados através dela a identificação de bugs e erros sistêmicos. Onde um analista funcional capacitado de conhecimento realizará a análise se o caso identificado se enquadra nas especificações para que seja desenvolvida uma correção ou evolução da aplicação. E então quando os dados chegarem para o desenvolvedor não haverá ruído entre o que foi identificado e o que foi realmente solicitado.

Também deve haver harmonia entre os processos de testes das funcionalidades corrigidas e desenvolvidas para que a identificação de erros de desenvolvimento possa ser identificada ainda em processos internos da empresa, para que não chegue a “estourar” o erro de uma correção má desenvolvida e má testada nas mãos de um usuário final.

Tudo isso pode ser evitado havendo a comunicação e padronização de processos entre os setores da companhia, as ilhas de responsabilidades não devem agir como se trabalhassem separadas, mas sim possuírem uma visão geral de ligação entre todas elas. Isso faz a diferença entre uma empresa que possua um processo com alta maturidade para aquelas que vão realizando as tarefas sem nenhuma forma de gerenciamento.

2.1.3 Testes que garantem a qualidade do produto

A qualidade dos produtos de softwares que a empresa desenvolve pode ser adquirida por meio de metódicas aplicações de testes nas várias fases do desenvolvimento do sistema. Testes estes, que são chamados de testes de validação (BIRTIÉ, 2002).

Definição utilizada por Alexandre Birtié (2002, p.19):

> “Estes testes são conhecidos como testes de validação porque, quando construímos uma unidade de software, validamos sua estrutura interna e sua aderência aos requisitos estabelecidos. Avaliamos sua integração com as demais unidades de softwares existentes, validando as

> interfaces de comunicação existentes entre os componentes de software. Quando determinado subsistema ou mesmo a solução esta finalizada, validamos a solução tecnológica como um todo, submetendo a testes de todas as categorias possíveis.”

Testes que são aplicados em softwares são denominados de testes de software ou chamados de testes dinâmicos.

Em testes de softwares, é possível que boa parte destes testes possam vir a ser automatizados, simulando várias regras de negócios e diversos cenários que aquela aplicação abrange.

“Quanto mais cenários simulados, maior o nível de validação que obtemos do produto, caracterizando maior nível de qualidade d software desenvolvido.” (BIRTIÉ, 2002, p. 20)

2.1.4 Dimensão da qualidade do produto

Conforme exemplificado e teorizado anteriormente, os passos relativos a qualidade do processo, posteriormente os testes feitos em função de validação do produto, podemos chegar enfim a qualidade do produto entregado.

A dimensão da qualidade do produto fica completamente evidente dentro de um processo de desenvolvimento de software. È normal que dentro do cronograma de desenvolvimento de software existam fases específicas voltadas somente para testes na aplicação, acabando por serem substituídas em grande parte das vezes por atividades de correção e manutenção de software. (BIRTIÉ, 2002).

Um produto que atinja um nível de qualidade dentro do mercado, pode se assim dizer, que seja um sistema que tenha um alto grau de confiabilidade, disponibilidade,segurança e proteção.

- Disponibilidade: É a característica ou o ato do sistema estar pronto para uso a todo o momento.
- Confiabilidade: É a expectativa do software em corresponder e oferecer os serviços esperados pelo usuário
- Segurança: Integridade das informações que hospedam e trafegam por aquele sistema.

- Proteção: Característica do software de não permitir que ocorram ataques ou invasões externas.

Vale também citar a capacidade do software em continuar atuando ou respondendo ao usuário mesmo que o mesmo se encontre sob ataque ou sob alguma falha, isto é denominado de Sobrevivência do Sistema.

3. CONHECENDO O MODELO CMM

3.1 Modelo CMM

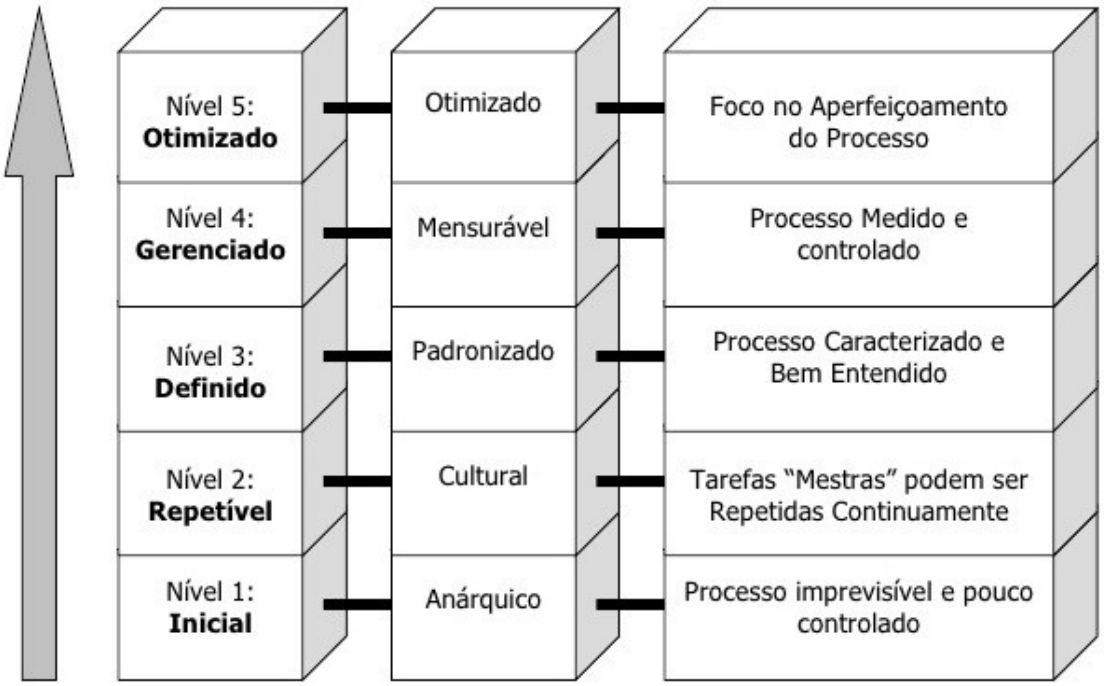
O modelo CMM (Capability Maturity Model) definido pelo Software Engineering Institute (SEI) trata-se da descrição de maneira sistemática, todos os fundamentos necessários para que a empresa de desenvolvimento de software possa tornar seu processo de desenvolvimento o mais eficiente e controlado possível. (BIRTIÉ, 2002).

> "O CMM baseia-se em um modelo evolutivo de maturidade, no qual as organizações partem de uma total falta de controle e gerenciamento dos processos (imaturidade organizacional) para gradativamente adquirir novas competências, incrementando seu nível de eficiência e maturidade em relação aos diversos processos críticos envolvidos em um desenvolvimento de software." (BIRTIÉ, 2002, p. 9)

O CMM utiliza-se de cinco estágios ou níveis de maturidade organizacional. Cada estágio ou nível equivale ao quanto o processo de desenvolvimento de software se encontra maduro ou não. Cada empresa se adéqua dentro de determinado nível ou estágio, para isso e realizada medições do controle organizacional da companhia baseadas em métricas universais.

O CMM se comporta como um procedimento em cascata ou podemos dizer nesse caso como uma escada, onde não podemos atingir o nível 3 sem antes passarmos pelo nível anterior a ele, nesse caso o nível 2.

Figura 01 – Possíveis níveis de maturidade previstos no modelo CMM

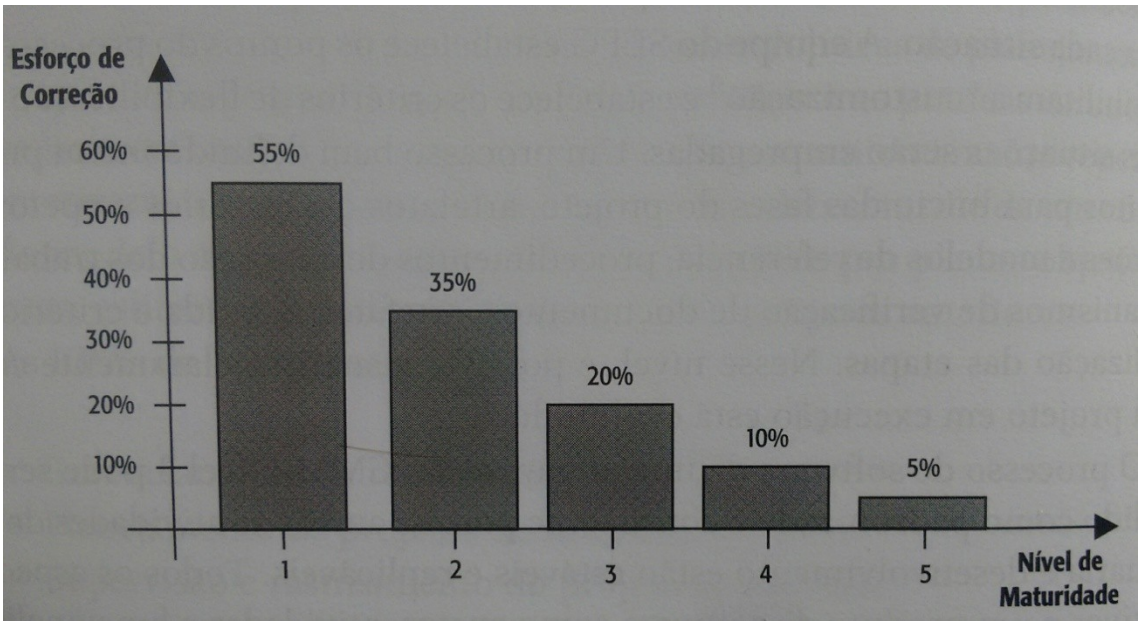


Fonte: Birtié (2002, p.9)

Quanto maior o nível que a empresa se encaixa nos elementos CMM maior é a qualidade em seu produto. Isso toma-se como base, a influência na automatização de testes, pois trata-se de um processo dentro os muitos observados pelo CMM que visa a melhoria e eficiência na sua maturidade. A automatização pode ajudar a empresa a subir de nível ou se estabelecer como um nível de maturidade alto, pois para que haja a automação de testes é necessário também todo o controle e gerência do processo, fato esse que é o qual buscamos ao utilizarmos a linguagem natural como objeto de automação.

Para exemplificar ainda melhor a maturidade que buscamos ao utilizar a automação, isto se relaciona com a imagem a seguir que demonstra o esforço de correção de acordo com o nível de maturidade que a empresa se encontra no momento.

Figura 02 – Esforço dedicado a correção de defeitos no desenvolvimento de software



Fonte: Birtié (2002, p. 14)

4. TESTES DE SOFTWARE

Testar um software significa realizar simulações sistemáticas para determinar se o produto cumpriu com as especificações que lhe foi atribuído e também verificar se funcionou adequadamente no ambiente em qual o mesmo foi projetado.

Identificando assim defeitos que antes não eram visíveis no processo de desenvolvimento, mas que posteriormente com a execução dos testes realizada, o defeito, erro ou falha pode ser identificado e eventualmente corrigido pela própria equipe de desenvolvimento.

Seu foco essencial é identificar o maior número possível de falhas dispondo do menor esforço para que isso seja realizado.

4.1 Conceitos de falha, erro e defeito

Existem diferenças entre os termos falha, erro e defeito, e entendê-los é uma parte importante para a realização de testes em softwares.

- Defeito: Característica do sistema que resulta na incapacidade de cumprir sua função.
- Erro: Provocado pelo defeito, é o estado que se encontra o sistema.
- Falha: Resultado de um defeito que gerou um erro. Pode se dizer que a falha é causada por um ou diversos erros, mas que um número de erros podem nunca talvez causar uma falha.

4.2 Conceitos de casos e critérios de Teste

Caso de Teste: descreve uma condição particular a ser testada e é composto por valores de entrada, restrições para a sua execução e um resultado ou comportamento esperado (CRAIG e JASKIEL, 2002, apud Revista Engenharia do Software, 2010).

Procedimento de Teste: é uma descrição dos passos necessários para executar um caso (ou um grupo de casos) de teste (CRAIG e JASKIEL, 2002, apud Revista Engenharia do Software, 2010).

Critério de Teste: serve para selecionar e avaliar casos de teste de forma a aumentar as possibilidades de provocar falhas ou, quando isso não ocorre, estabelecer um nível elevado de confiança na correção do produto (ROCHA et al., 2001, apud Revista Engenharia do Software, 2010).

A utilização dos critérios de testes pode ser:

Critério de Cobertura dos Testes: permitem a identificação de partes do programa que devem ser executadas para garantir a qualidade do software e indicar quando o mesmo foi suficientemente testado (RAPPS e WEYUKER, 1982, apud Revista Engenharia do Software, 2010).

Critério de Adequação de Casos de Teste: Quando, a partir de um conjunto de casos de teste T qualquer, ele é utilizado para verificar se T satisfaz os requisitos de teste estabelecidos pelo critério. Ou seja, este critério avalia se os casos de teste definidos são suficientes ou não para avaliação de um produto ou uma função (ROCHA et al., 2001, apud Revista Engenharia do Software, 2010).

Critério de Geração de Casos de Teste: quando o critério é utilizado para gerar um conjunto de casos de teste T adequado para um produto ou função, ou seja, este critério define as regras e diretrizes para geração dos casos de teste de um produto que esteja de acordo com o critério de adequação definido anteriormente (ROCHA et al., 2001, apud Revista Engenharia do Software, 2010).

4.3 Tipos de testes de software

- Teste de instalação: Teste utilizado para a verificação se o software instala corretamente em diferentes hardwares, variando as condições, como por exemplo, pouco espaço em memória, forçando a interrupção de instalação e etc.
- Teste de configuração: Averigua se o software em questão funciona corretamente no hardware instalado.
- Teste de Segurança: Verifica a integridade das informações contidas no sistema, realização de autenticações para que os dados sejam acessados somente por usuários que possuam tal permissão.
- Teste de Integridade: Teste voltado para a resistência do sistema quanto a tolerância a falhas.
- Teste Funcional: Comprova por meio de simulações se o sistema atende aos requisitos funcionais estabelecidos, funcionalidades, regras de negócios e casos de uso.
- Teste de Unidade: Realiza os testes de componentes isolados. Como por exemplo, um software que emite uma declaração de exportação que depende de n fatores e cadastros anteriores até chegar este a ponto do sistema, porém com o teste de unidade, realizo o teste abrangendo somente este momento da minha aplicação.

- Teste de Integração: Testa a integração, ou seja a combinação de componentes, um exemplo pode ser dado como o teste de uma determinada interface que transfere informações de um sistema para o outro, podendo citar um sistema de contas a pagar da empresa gera informações que são carregadas através desta interface para outro sistema de controle de exportação e importação.

- Teste de Volume: São executadas operações do sistema que geram um tráfego de dados envolvendo o banco de dados durante um grande período de tempo.

- Teste de Performance de Carga: Testa o software sob condições padrões de uso, analisando o tempo que o software leva para responder tais operações, a quantidade de usuários simultâneos que o software opera.

- Teste de Performance de Stress: Diferentemente do teste de carga, esta situação de teste faz com que o sistema atue sob condições extremas e exageradas de uso.

- Teste de Performance de Estabilidade: Analisa se após um determinado período de uso o sistema ainda se encontra respondendo e funcionando da maneira esperada e eficiente.

- Teste de Usabilidade: Trata-se de um teste focado na resposta do usuário, verifica se o layout, disponibilidade do conteúdo em tela, o acesso as funcionalidades da aplicação, interfaces visuais se adéquam satisfatoriamente, trazendo uma boa experiência para o usuário.

- Testes de Caixa Preta e Caixa Branca: O teste de caixa branca envolve principalmente a análise do código fonte do sistema, trata-se muito mais de um teste no back end da aplicação, já o teste de caixa preta trata-se de um teste onde é apurado o funcionamento correto dos requisitos funcionais do sistema,

processos de uso e funcionalidades, pode se dizer que abrange o front end da aplicação.

- Teste de Regressão: É realizado o reteste de vários componentes que já foram aprovados, porém deve verificar se alguma alteração feita recentemente causos impactos indesejados.
- Teste de Manutenção: Analisa se a alteração da ambientação interfere no bom funcionamento do software.

4.4 Automação de Testes

O trabalho realizado manualmente de testes em softwares demanda que a empresa possua profissionais altamente capacitados e com boa experiência nas funcionalidades do sistema para casos de testes mais complexos. Porém toda ação humana é suscetível a falhas, e como o processo de teste se torna um processo muitas vezes maçante e repetitivo o acontecimento dessas falhas humanas torna-se muito mais propício e comum de acontecer.

Em alguns casos o profissional analista de testes pode refazer o mesmo caso de testes inúmeras vezes, fazendo com que seu olhar para detectar e encontrar possíveis falhas esteja de certo modo viciado, não estando mais apurado como antes, deixando passar detalhes importantes. O vício é um caso comum e um dos principais sujeitos de teste mal sucedido. Pois com isso o profissional deixa de imaginar novos cenários fazendo com que aquela funcionalidade que esta sendo testada seja sempre posta a prova da mesma forma.

Nesta situação adentramos a automatização de testes, pois com o uso de uma ferramenta e uma linguagem de programação a automação de testes faz com que os problemas evidenciados no parágrafo acima referentes a falha humana, não ocorram mais, já que o sujeito responsável a realizar o teste, as validações, informações, estatísticas e relatórios é a máquina.

Vale frisar que não são todos os casos que podem usufruir da automação de testes, deve-se analisar cada projeto e identificar quais

realmente valem a pena estar automatizando. Com isso o medo inicial do funcionário que executa os testes manuais de ficar sem emprego é em vão, pois ainda precisamos dele, porém de uma forma mais capacitada e mais técnica que antes.

Definição utilizada por Alexandre Birtié para testes automatizados (2002, p.196):

> "Trata-se da utilização de ferramentas de testes que possibilitam simular usuários ou atividades humanas de forma a não requerer procedimentos manuais no processo de execução de testes. Requer profissionais especializados e tempo no desenvolvimento da automação dos testes"

Segundo Alexandre Birtié existem ganhos com a automação de testes (2002, p.197):

> "A automação dos testes é realmente desejada por diversos fatores, inclusive em termos de custos finais. Como esse processo requer um investimento inicial, a automação passa a ser encarada como mais um trabalho a ser realizado. Á medida que reexecutamos os testes, o ganho de tempo, controle, confiabilidade e as diversas possibilidades existentes com essa tecnologia ficam claros como vantagem inerente a esse processo."

4.5 Benefícios da automação de testes

A automação de testes esta ligada diretamente a qualidade do produto desenvolvido, pois através dela podemos obter um número considerável na redução da entrega do produto com defeitos. Pois como uma máquina, através de uma ferramenta de automação, implementada por uma lógica de programação é quem executa os testes, a porcentagem de assertividade na identificação de falhas é muito maior que por uma humano executando testes manuais.

Além de um produto de melhor qualidade, as automações de testes também podem beneficiar gerentes na tomada de decisão já que conta como auxílio na geração de relatórios e resultados, fazer com que os profissionais

atuantes da área de testes transformem-se em profissionais mais capacitados, deixar profissionais que antes continham a sobrecarga de tarefas para agora atuarem somente em testes onde a complexidade do teste é muito mais relevante já que a automação é mais bem empregada naqueles testes maçantes e repetitivos onde os resultados podem ser verificados por uma máquina sem a necessidade de um julgamento humano.

5.1 Utilizando o BDD para automação

Antes de citar o BDD (Behavior Driven Development), é valido informar que por volta de 1996 foi desenvolvido o TDD (Test Driven Development), como base para desenvolver os testes de unidade antes do desenvolvimento do código fonte em si. Tem como objetivo que o desenvolvedor escreva os testes para que então isto auxilie para que ele desenvolva o mínimo possível e depois assim passar um pente fino em seu código melhorando a sua implementação. Mas este modelo de teste não era satisfatório já que este sistema de TDD não englobava os chamados critérios de aceite, ou seja, não acompanhavam uma abordagem baseada nos casos de uso do usuário dificultando assim identificar se a aplicação estava se comportando como deveria e atendendo as regras de negócios do cliente já que ela baseada em casos de testes de unidade. Sem falar que o entendimento da sua escrita era muito complexo e quando mostrado ao cliente se era este o caso de uso que ele necessitava o entendimento por parte do cliente era nulo

Com isso em meados de 2006 surgiu o BDD:

> “Envolve o negócio, teste de software, desenvolvimento e planejamento. Olhando de cima, o BDD é uma abordagem inovadora de teste que junta os dois mundos: Em apenas um repositório, você mantém os testes de aceite automatizados de uma forma simples de ler para o cliente, desenvolvedores, QAs e demais membros da equipe e ao mesmo tempo esses testes são automatizados antes do desenvolvimento, formando um conjunto de testes de aceite que guiam o desenvolvimento das histórias e em seguida são usados como testes de regressão. Nessa abordagem, a documentação e o código de desenvolvimento evoluem sempre juntos.” (RIBEIRO, 2012)

O BDD possui três princípios:

- O suficiente é suficiente: Não se deve realizar a automação de tudo, mas sim daquilo que descreve o comportamento esperado do produto pelo cliente. (RIBEIRO, 2012)
- Entregar valor para os stakeholders: Deve ser entregue somente aquilo que agrega valor ao cliente. (RIBEIRO, 2012)
- Tudo é comportamento: Todas as operações que uma aplicação realiza podem ser descrita como um comportamento e o mais importante que isso pode ser explicado para qualquer pessoa que tenha certo domínio do negócio. Não importando seu nível de teste ou sua complexidade, sempre poderá ser descrito como comportamento. (RIBEIRO, 2012)

A diferença gritante entre o TDD e o BDD é que enquanto o primeiro se mantém voltado no design do código, o BDD possui como objetivo o entendimento do negócio.

> "Em uma analogia, podemos dizer que usamos BDD para descrever que um carro deve acelerar e para isso descrevemos que quando acionamos o pedal do acelerador, o carro deve acelerar a tal velocidade, o velocímetro deve aumentar e que o indicador de giro do motor deve exibir o giro, mas quando vamos montar o motor devemos descrever com TDD tudo que está dentro do motor.." (RIBEIRO, 2012)

5.2. Implementando o BDD com o conceito de Cucumber

O Cucumber é uma linguagem natural que pode ser utilizada para automação de testes, ela é baseada na descrição dos critérios de aceite e na camada história do usuário, ele serve para descrever o comportamento que o software deve obedecer para o funcionamento correto de uma determinada funcionalidade.

Para isso ele utiliza um modelo de escrita:

Scenario: <description of the test>

Given <a known state>

When <an event occurs>

And <Other event Occurs>

Then <then this should happen>

As palavras em negrito são de uso obrigatório quando irei descrever o comportamento do software em minha linguagem natural “cucumber”, e as palavras seguidas delas são a descrição do que se espera informar em cada linha.

Em uma tradução literal para o português para um melhor entendimento, as cláusulas ficariam desta maneira:

Cenário: <descrição do teste>

Dado <um estado conhecido>

Quando <um determinado evento ocorre>

E <outro evento também ocorre>

Então <isso deve ocorrer>

Exemplificando o uso do BDD em linguagem Cucumber para melhor entendimento em português:

Cenario: Cadastro básico de usuário

Dado que estou na tela de cadastro de usuário

Quando eu clico no botão de incluir cadastro

E preencho o campo “nome_usuario” com o valor “José”

E clico no botão salvar

Então o sistema deve salvar o registro com sucesso

Note que toda a informação que não é uma constante como por exemplo o nome do campo a ser preenchido e o valor que será preenchido neste campo deve ser posto sempre entre aspas.

Existe algumas variações da utilização do cucumber, como por exemplo quando um mesmo cenário de teste pode abranger vários cadastros, ao invés de ficarmos fazendo um cenário para cada teste apenas mudando as informações de cadastro, fazemos tudo em um cenário só utilizando a clausula exemplo, veja a seguir:

Cenario: Cadastro de País

Dado que estou na tela de cadastro de país

Quando eu clico no botão incluir cadastro

E preencho o campo “nome_país” com o valor “<tipo_país>”

E clico no botão salvar

Então o sistema deve salvar o registro com sucesso

Exemplos:

| tipo_país |

| Brasil |

| Inglaterra |

| Alemanha |

Note que ao invés de criar um passo a passo repetitivo para cada país que devo cadastrar você simplesmente deve criar uma variável posta em <> que mais tarde ao final do meu cenário, a identifico como Exemplos e para o

nome da variável criada então informa a variedade de informações que terei que cadastrar neste mesmo campo.

5.3 Desenvolvimento de uma linguagem natural padronizada

Lembrando que será proposto neste trabalho uma linguagem natural desenvolvida pelo autor para ser tomada como base e/ou padrão para o auxílio na criação de comportamentos de softwares. A linguagem proposta abrange uma grande parte de todas as operações que devem ser realizadas por uma aplicação fazendo com que poucas modificações nela sejam necessárias para atender o cenário do produto do desenvolvedor que preferir utilizá-la. Seu enfoque principal é a ajuda na fácil identificação dos pontos os foram detectadas falhas e auxílio na geração de relatórios de forma simples e que pode ser compreendida facilmente por pessoas que tenham um conhecimento prévio das regras de negócios do software em questão. Como já dito o BDD foca na parte de comportamento do sistema, ou seja, esta linguagem proposta foi totalmente desenvolvida com enfoque apenas na parte funcional do software.

Linguagem desenvolvida para Login em um sistema:

Trecho da Linguagem será sempre identificado como 1:

1) Given I open system and login using "name_user" username and "pass_user" password using "type_language" language

Exemplo da utilização do Trecho citado anteriormente será sempre identificado como 2:

2) Given I open system login using "AUTOMATION" username and "AUTOMATION123" password using "English" language

1) And I use "name_module" module
2) And I use "Export" module

Linguagem desenvolvida para acesso as telas e manipulação de dados:

1) When I open "name_screen" screen in "type_mode" mode
2) When I open "Export Order" screen in "Creation" mode

1) And I fill "name_field" field with value "value"
2) And I fill "Order Number" field with value "Order_Teste"

1) And I select "name_item" option of "name_combo" combo
2) And I select "English" option of "Language" combo

1) And I click on "name_button" Button
2) And I click on "Add" Button

1) And I select "value_option" option of "name_radio_button" radio
2) And I select "Active" option of "Status" radio

1) And in the LOV "name_LOV" I select "value_select" filtering by "value_filtering"
2) And in the LOV "Currency" I select "USD" filtering by "Currency Code"

1) And open tab "name_tab" of current screen
2) And open tab "Item" of current screen

1) And system return "name_screen" Screen
2) And system return "Order Item" Screen

1) And I check line [number_line] item
2) And I check line [1] item

1) And I fill the date hour field "name_field" with current date
2) And I fill the date hour field "Invoice Date" with current date

1) And I “type_manipulation” Record
2) And I “edit” Record

Linguagem desenvolvida para manipulações de dados em tabelas:

1) And I add an item on table "name_table"
2) And I add an item on table "Organization"

1) And on table "name_table", line "number_line" and column "number_column", I select LOV item "value_item" filtering field "name_field"
2) And on table "Organization", line "1" and column "2", I select LOV item "Japan Subsidiary" filtering field "Code"

1) And on table "name_table", line "number_line" and column "number_column", I select "value_item" option of "name_combo" combo
2) And on table "Customs Profile Details", line "1" and column "3", I select "JP001 - Consume" option of "Custom Profile" combo

1) And on table "name_table", line "number_line" and column "number_column", I fill "name_field" field with value "value"
2) And on table "Customs Profile Details", line "1" and column "6", I fill "Quantity" field with value "10"

Linguagem desenvolvida para cláusulas do tipo Examples:

1) And I fill "name_field" with value "<type code>"
2) And I fill "Code" field with value "<type code>"

1) And I fill "name_field" field with value "<type description>"
2) And I fill "Description field with value "<type description>"

Examples:

type code	type description
Origin Japan	Japan Origin
Destination USA	USA Destination

Linguagem desenvolvida para validações de mensagens do sistema e inserção com sucesso de dados:

1) And I "type_operarion" record
2) And I “save” Record

1) Then the system should show "type_message"
2) Then the system should show "include successful message"

1) Then the system should "result_expected"
2) Then the system should "not return results"

Linguagem desenvolvida para alguns casos específicos:

1) And I search for "value" on field "name_field" at filter screen
2) And I search for "RDI-IN" on field "Request NUmber" at filter screen

1) And I click link on table "name_table" line "number_line" and column "number_column", opening "name_modal" modal
2) And I click link on table "Items" line "2" and column "6", opening "Duty Credit Scrip" modal

1) And on table "name_table" at line "number_line", I select "value_item" option of "name_radio" radio
2) And on table "items" at line "1", I select "Item" option of "item" radio

1) And I click on "name_button" button, opening modal
2) And I click on "Apply" button, opening modal

1) And I click on "name_button" button, closing modal
2) And I click on "Apply" button, closing modal

1) And I fill "name_field" unique key field with value "value"
2) And I fill "Revision Number" unique key field with value "2"

1) And I go back to module selection
2) And I go back to module selection

6.1. Utilização da linguagem natural desenvolvida com o Jenkins

Para a gestão e manutenção da automação dos testes foi utilizada a ferramenta web denominada Jenkins, que trata-se de uma integração contínua com projetos que sejam desenvolvidos em Java, pois por trás deste métodos comportamentais do software descrito na linguagem natural utilizando o modelo Cucumber há todo um código fonte desenvolvido em Java para a execução dos métodos descritos no BDD.

O Jenkins permite a configuração de todo seu projeto como, por exemplo, a definição do diretório da máquina em que será executado os testes, no caso deste trabalho, o diretório foi definido em uma máquina virtual com repositório do projeto JAVA em Github que nada mais é de uma maneira simples o repositório na nuvem onde é hospedado o código fonte e também BDD, pois assim toda e qualquer alteração que seja feita é salva instantaneamente caso o desenvolvedor queira, evitando transtornos como possuir o repositório em uma máquina local onde toda vez deve ser verificado o que foi alterado ou não para que dados já alterados não sejam sobre escritos e percam a modificação já feita por um desenvolvedor desavisado.

Voltando para o Jenkins, com ele os projetos de testes criados podem ser configurados para o horário que deseja que o teste seja executado, quantas vezes ele seja executado, seja diariamente, semanalmente ou mensalmente. E com a utilização da linguagem natural e a instalação de um plugin do Cucumber estes testes podem ser executados sem que seja necessário o acompanhamento de um profissional já que os pontos de quebra de testes são reportados após o término de maneira clara e objetiva por conta do uso da linguagem natural.

Figura 03 – Projeto de teste criado no Jenkins e report de informações após seu término

Project	Number	Date
gtm.Fluxo.Import.China.BDD	67	24 Jun 2016, 13:39

Features Statistics

The following graphs show passing and failing statistics for features

	Scenarios			Steps								
Feature	Total	Passed	Failed	Total	Passed	Failed	Skipped	Pending	Undefined	Missing	Duration	Status
End to end flow of China import	19	8	11	279	154	11	114	0	0	0	16m 46s 497ms	Failed
MasterData of China import	9	8	1	105	93	1	11	0	0	0	4m 51s 963ms	Failed
2	**28**	**16**	**12**	**384**	**247**	**12**	**125**	**0**	**0**	**0**	**21m 38s 461ms**	**Totals**

Fonte: Autor

Pela imagem retirada do projeto criado no Jenkins é visívelmente notado o ganho de informações inteligentes devido ao fato do uso da linguagem natural.

Note que o Jenkins informa o nome do Projeto seguido de uma esfera podendo ser da coloração azul, cinza ou vermelha, onde azul significa que o teste foi eficaz passando com sucesso em todos os pontos. Cinza demonstra que o teste não foi realizado. Vermelho conota-se a testes que tiveram fracasso em algum ponto do BDD.

O símbolo de chuva e raios seguidos na figura denota-se que dentre as últimas execuções o teste obteve um número maior de falhas do que de sucesso. Quando isto é ao contrário o Jenkins mostra um símbolo ensolarado.

Embaixo temos um gráfico que informa os passos do BDD em que o teste passou com sucesso, os passos em que o teste apresentou erro e os passos em que o teste foi obrigado a pular.

Temos toda a informação relevante e necessária para a gestão da automação. Isto é de suma importância para a área de responsabilidade em gerir os testes. Também com isso não se faz necessário o acompanhamento de um profissional no momento de execução do teste notando-se que ele pode muito bem identificar as falhas com o reort feito pelo Jenkis.

Lembrando que esse nível de detalhes se deve ao fato do uso da linguagem natural estruturada no modelo Cucumber e ao plugin do Cucumber instalado no Jenkins.

Figura 04 – Identificação dos passos que falharam no BDD

Fonte: Autor

Com a utilização da linguagem natural proposta pelo autor deste trabalho fica muito fácil entender qual ponto o teste apresentou falha e então após feito isso entender e buscar informações com o próprio analista funcional se houve alguma alteração na regra de negócio e/ou processos daquele ponto identificado no sistema.

No exemplo mostrado identificamos que houve um erro no passo:

And on table “items” at line “1”, I select “Item” option of “item” radio

Com esse nível de detalhamento a conclusão de que pode ser um problema no radio Button da tabela items que faz parte do comportamento no caso de teste da funcionalidade Import Declaration (definido no nome do cenário de teste), se localiza no módulo “Import” na tela de “Import declaration” é totalmente intuitiva com a leitura do BDD, gerando benefícios na rapidez e facilidade de manutenção.

Agora veja como seria gerado este report de falha no teste caso não estivéssemos utilizando a linguagem natural:

Figura 05 – Report Jenkins sem o uso da Linguagem natural

```
Mar 01, 2016 11:00:47 AM FINE hudson.plugins.scm_sync_configuration.SCMManipulator
Checking in SCM files ...
Mar 01, 2016 11:00:47 AM FINER org.apache.maven.scm.manager.ScmManager checkIn
THROW
org.apache.maven.scm.ScmException: Exception while executing SCM command.
        at org.apache.maven.scm.command.AbstractCommand.execute(AbstractCommand.java:63)
        at org.apache.maven.scm.provider.git.AbstractGitScmProvider.executeCommand(AbstractGitScmProvider.java:291)
        at org.apache.maven.scm.provider.git.AbstractGitScmProvider.checkin(AbstractGitScmProvider.java:217)
        at org.apache.maven.scm.provider.AbstractScmProvider.checkIn(AbstractScmProvider.java:415)
        at org.apache.maven.scm.provider.AbstractScmProvider.checkIn(AbstractScmProvider.java:397)
        at org.apache.maven.scm.manager.AbstractScmManager.checkIn(AbstractScmManager.java:423)
        at hudson.plugins.scm_sync_configuration.SCMManipulator.checkinFiles(SCMManipulator.java:241)
        at hudson.plugins.scm_sync_configuration.ScmSyncConfigurationBusiness.processCommitsQueue(ScmSyncConfigurationBusiness.java:223)
        at hudson.plugins.scm_sync_configuration.ScmSyncConfigurationBusiness.access$000(ScmSyncConfigurationBusiness.java:32)
        at hudson.plugins.scm_sync_configuration.ScmSyncConfigurationBusiness$1.call(ScmSyncConfigurationBusiness.java:148)
        at hudson.plugins.scm_sync_configuration.ScmSyncConfigurationBusiness$1.call(ScmSyncConfigurationBusiness.java:145)
        at java.util.concurrent.FutureTask.run(FutureTask.java:266)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)
Caused by: org.apache.maven.scm.ScmException: Error while executing command.
        at org.apache.maven.scm.provider.git.gitexe.command.GitCommandLineUtils.execute(GitCommandLineUtils.java:122)
        at org.apache.maven.scm.provider.git.gitexe.command.checkin.GitCheckInCommand.executeCheckInCommand(GitCheckInCommand.java:129)
        at org.apache.maven.scm.command.checkin.AbstractCheckInCommand.executeCommand(AbstractCheckInCommand.java:54)
        at org.apache.maven.scm.command.AbstractCommand.execute(AbstractCommand.java:59)
        ... 14 more
Caused by: org.codehaus.plexus.util.cli.CommandLineException: Error inside systemOut parser
        at org.codehaus.plexus.util.cli.CommandLineUtils$1.call(CommandLineUtils.java:188)
        at org.codehaus.plexus.util.cli.CommandLineUtils.executeCommandLine(CommandLineUtils.java:107)
        at org.codehaus.plexus.util.cli.CommandLineUtils.executeCommandLine(CommandLineUtils.java:74)
        at org.apache.maven.scm.provider.git.gitexe.command.GitCommandLineUtils.execute(GitCommandLineUtils.java:118)
        ... 17 more
Caused by: java.lang.IllegalArgumentException: Illegal character in path at index 0: "jobs/       %2(     `  %20!       /config.xml"
        at java.net.URI.create(URI.java:852)
        at org.apache.maven.scm.provider.git.gitexe.command.status.GitStatusConsumer.resolveURI(GitStatusConsumer.java:251)
        at org.apache.maven.scm.provider.git.gitexe.command.status.GitStatusConsumer.resolvePath(GitStatusConsumer.java:232)
        at org.apache.maven.scm.provider.git.gitexe.command.status.GitStatusConsumer.consumeLine(GitStatusConsumer.java:133)
        at org.codehaus.plexus.util.cli.StreamPumper.consumeLine(StreamPumper.java:190)
        at org.codehaus.plexus.util.cli.StreamPumper.run(StreamPumper.java:135)
Caused by: java.net.URISyntaxException: Illegal character in path at index 0: "jobs/(      %20       %20       /config.xml"
        at java.net.URI$Parser.fail(URI.java:2848)
        at java.net.URI$Parser.checkChars(URI.java:3021)
        at java.net.URI$Parser.parseHierarchical(URI.java:3105)
        at java.net.URI$Parser.parse(URI.java:3063)
        at java.net.URI.<init>(URI.java:588)
        at java.net.URI.create(URI.java:850)
```

Fonte: Autor

CONCLUSÃO:

A automação de teste quando feita de maneira correta e inteligente, principalmente com a intenção de substituir testes massivos e repetitivos que abrangem muito em casos de uso e processos realizados pelo cliente para

verificação do funcionamento correto de várias funcionalidades traz consigo um efeito muito benéfico para a empresa.

E junto da automação de teste podemos usufruir de maneira mais simples e eficaz utilizando a implementação em linguagem natural. Este método apesar de possuir uma vida recente no mercado vem trazendo vários adeptos com o passar do tempo, devido a sua competência em facilitar a manutenção e o desenvolvimento de uma automação de teste.

Com o uso do BDD descrevendo o comportamento do software o entendimento do desenvolvedor da automação sobre as regras de negócio do software se tornou muito mais fácil, sendo possível que o BDD seja apresentado aos analistas funcionais do sistema ou até mesmo para o cliente para validação se os comportamentos descritos neles abrangem as funcionalidades que o usuário final utiliza. Assim se evita um transtorno constante na hora de automatizar softwares, o “ruído” de informações.

Além de todos esses benefícios a linguagem natural faz com que os profissionais possuam maior capacitação no mercado de testes, melhora e simplifica a forma de gestão dos testes, faz com que funcionários possam focar em outras tarefas mais complexas do que testes de funcionalidades maçantes e repetitivas, aumenta a assertividade dos testes já que a máquina não possui um olhar viciado, facilidade na detecção das falhas já que a linguagem natural quando bem descrita faz com que esse processo seja intuitivo pela análise dos relatórios e reports feitos pela ferramenta, fazendo com que tudo isso acarrete numa melhora significativa na qualidade final do produto.

REFERÊNCIAS BIBLIOGRÁFICAS

Livro

BERTIÉ, Alexandre. **GARANTIA DA QUALIDADE DE SOFTWARE**. 1ª edição. Rio de Janeiro: Elsevier Editora Ltda, 2002. 291 p.

KOSCIANSKI, André e SOARES, Michel dos Santos. **QUALIDADE DE SOFTWARE**. 2ª edição. São Paulo: Novatec Editora Ltda, 2006. 395 p.

Web

RIBEIRO, Camilo. **Entendendo BDD com Cucumber – Parte I.** 2012.Disponível em: http://www.bugbang.com.br/entendendo-bdd-com-cucumber-parte-i/. Acesso em: 12 out. 2016.

ARILO, Claudio. **Artigo Engenharia de Software – Introdução a teste de Software.** 2016.Disponível em: http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035. Acesso em: 16 out. 2016.

KOSMALSKI, Erick Scudero. **OS 13 PRINCIPAIS TIPOS DE TESTES DE SOFTWARE.** 2015.Disponível em: http://www.targettrust.com.br/blog/desenvolvimento/testes/os-13-principais-tipos-de-testes-de-software/. Acesso em: 16 out. 2016.

NEGRINI, Celia. **Desafios e Benefícios da automação de testes – Engenharia de Software Magazine 58.** 2015.Disponível em: http://www.devmedia.com.br/desafios-e-beneficios-da-automacao-de-testes-engenharia-de-software-magazine-58/28051. Acesso em: 22 nov. 2016.

RODRIGUES, Fernando. **Testes de Software – Entendendo Defeitos, Erros e falhas.** 2015.Disponível em: http://www.devmedia.com.br/testes-de-software-entendendo-defeitos-erros-e-falhas/22280. Acesso em: 22 nov. 2016.

MENDES, Antonio. **Plano de Teste – Um mapa essencial para teste de software.** 2015.Disponível em: http://www.devmedia.com.br/plano-de-teste-um-mapa-essencial-para-teste-de-software/13824. Acesso em: 22 nov. 2016.