PONTIFÍCIA UNIVERSIDADE CATÓLICA DE SÃO PAULO

CURSO DE ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

ELISANGELA CARVALHO DO NASCIMENTO

ANÁLISE DA TÉCNICA ESPECIFICAÇÃO POR EXEMPLO (SPECIFICATION BY EXAMPLE) COM A FERRAMENTA CUCUMBER

São Paulo, Dezembro de 2012.

ELISANGELA CARVALHO DO NASCIMENTO

ANÁLISE DA TÉCNICA ESPECIFICAÇÃO POR EXEMPLO (SPECIFICATION BY EXAMPLE) COM A FERRAMENTA CUCUMBER

Monografia apresentada ao Curso de Especialização em Engenharia de Software da Pontifícia Universidade Católica de São Paulo, como requisito parcial para obtenção do título de Especialista em Engenharia de Software, orientado pelo Prof. Dr. Daniel Couto Gatti.

São Paulo, Dezembro de 2012

ANÁLISE DA TÉCNICA ESPECIFICAÇÃO POR EXEMPLO (SPECIFICATION BY EXAMPLE) COM A FERRAMENTA CUCUMBER

ELISANGELA CARVALHO DO NASCIMENTO

Monografia aprovada em ___/___/___ para obtenção do certificado do curso de Especialização em Engenharia de Software.

Professor Daniel Couto Gatti

Dedico este trabalho à memória de minha mãe.
Que com suas palavras, tudo podia expressar.
Saudade que dói.
Que os anos amenizam, mas não curam.
Que o tempo tenta, mas não apaga.
Saudades de você, mãe.
De ter mãe.
De ser filha.

Agradeço ao meu Prof. Orientador Daniel Gatti, pelas orientações que colaboraram para a conclusão deste trabalho. E a outros que de alguma forma contribuíram para que este trabalho fosse realizado.

"Desconheço outra forma de julgar o futuro
que não seja pelo passado".
Patrick Henry

"Você jamais poderá planejar o futuro pelo
passado".
Edmund Burke

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

BDD: *Behavior Driven Development*

QA: *Quality Assurance*

SWEBOK: *Software Engineering Body of Knowledge*

LISTA DE SÍMBOLOS

 # language: Comentário utilizado pela linguagem Gherkin para introduzir a língua nativa de quem a utilizará.

RESUMO

Este trabalho aborda a Técnica de Especificação por Exemplo, que torna possível executar histórias em texto puro, descritos organizada e declarativamente, onde o cliente poderá detalhar suas necessidades de tal forma que todos os envolvidos no desenvolvimento do projeto possam entender de maneira clara e sem ambigüidade o requisito descrito.

Com esta técnica produzimos um documento de requisito executável de forma compreensível e consistente, embora esta não seja uma tarefa fácil de conseguir, pois os *Stakeholders* interpretam os requisitos de maneiras diferentes, e, muitas vezes, notam-se conflitos e inconsistências inerentes aos requisitos.

Conclui-se que o maior ganho do uso da Técnica de Especificação por Exemplo é a melhoria na comunicação aprimorando a colaboração com o cliente.

Palavras-chave: Engenharia de Requisitos, Especificação por Exemplo e *Stakeholders*.

ABSTRACT

This work is about the Technique of Specification by Example, which makes possible to execute histories in pure text, described in an organized and declared method, where the client can detail his needs in such way that all the people involved in the development of the project can understand in a clear and no ambiguous mode the requisite described.

Through this technique we produce a document of executable requisite in a comprehensible and consistent way, although this is not an easy task to accomplish, because the *Stakeholders* interpret the requisites in different points of view and, in many times, conflicts and inconsistencies attached to the requisites could be noticed.

Finally, can be concluded that the most important gain of the use of the Technique of Specification by Example is the upgrade on the communication, improving the contribution with the client.

Keywords: Requirements Engineering, Specification by Example, *Stakeholders*.

SUMÁRIO

INTRODUÇÃO

Bokhari e Siddiqui (2011, tradução nossa) apontam que há diversas falhas no desenvolvimento de software e elas estão relacionadas com a questão da engenharia de software, tais como, requisitos mal documentados, requisitos que eram impossíveis de cumprir e requisitos que não conseguem atender as necessidades do usuário, e que boas práticas ajudam a melhorar a satisfação do cliente, reduzir os custos do desenvolvimento do sistema e aumenta a chance de ter sucesso no projeto.

Conforme Brooks (2009, p. 192) a parte mais difícil da construção de um sistema de software é decidir o que construir. Nenhuma outra parte do trabalho prejudica tão seriamente o sistema resultante se for feita de forma errada. Nenhuma outra parte é mais difícil de consertar depois.

Portanto, para não tornar essa tarefa tão penosa é fundamental entender primeiramente o que são requisitos.

Pois, segundo Sommerville (2011, p. 57) requisitos são descrições do que o sistema deve fazer, os serviços que oferecem e as restrições a seu funcionamento. Esses requisitos refletem as necessidades dos clientes para um sistema que serve a uma finalidade determinada.

De acordo com o **SWEBOK**, Software Engineering Body of Knowledge, (2004), um requisito de software é uma propriedade que deve ser exibido, a fim de resolver algum problema no mundo real.

Enquanto que, para Pressman (2006, p.127), requisitos refletem os objetivos e metas estabelecidas para um produto ou sistema.

Uma definição bem simples é dada por Macaulay, (1996, tradução nossa) que descreve requisito como simplesmente algo de que o cliente necessita.

Entender os requisitos de um problema está entre as tarefas mais difíceis, afinal, o cliente não sabe o que é necessário e o usuário final muitas vezes não tem um bom entendimento das características e funções que vão oferecer benefícios.

Ouvir um cliente dizer: "Eu sei que você pensa que entende o que eu disse, mas o que você não entende é que, o que eu disse, não é o que eu queria dizer". É um dos motivos que a engenharia de requisitos nos fornece uma sólida abordagem para enfrentar esses desafios.

É sabido que requisitos são os alicerces de um projeto e que sem um documento de requisito que atenda as características de qualidade, não há como dizer que um projeto atingiu seus objetivos. Mas como documentá-los com clareza, como desenvolver um vocabulário consistente para analistas, testadores e desenvolvedores, como eliminar a ambiguidade das informações e dos problemas de comunicação que ocorrem quando pessoas técnicas falam com pessoas de negócio, como contar com a participação efetiva dos clientes e usuários na fase constante que abrange a verificação e validação e não somente na fase final do processo de requisitos. E como garantir que um requisito tenha sofrido uma alteração e que a sua documentação tenha acompanhado as mudanças?

A ferramenta Cucumber é mais do que uma ferramenta de teste, é uma ferramenta de colaboração, pois muitos projetos de software sofrem com a comunicação de baixa qualidade entre os especialistas do domínio e a equipe de programadores (EVANS apud WYNNE e HELLESOY, 2012, p. 05, tradução nossa). A ferramenta Cucumber facilita a descoberta do uso de uma linguagem universal com a equipe onde pode-se explorar e erradicar muitos mal entendidos (WYNNE e HELLESOY, 2012, p. 05, tradução nossa).

E é a partir da técnica de Especificação por Exemplo com a ajuda da ferramenta Cucumber, que se dá a importância para que os requisitos se tornem especificações executáveis, tornando possível ilustrar requisitos, usando exemplos realistas em vez de afirmações abstratas.

Motivação do Trabalho

Tendo em vista que algumas das principais razões para que um projeto não obtenha sucesso é a necessidade de ter a participação do usuário e uma declaração clara dos requisitos (THE STANDISH GROUP, 1995), o uso da técnica Especificação por Exemplo, busca resolver estas questões na forma de cenários, expressos em linguagem natural, permitindo assim, que a documentação dos requisitos discutidos com o cliente seja o mesmo artefato que será verificado e validado ao final da especificação construída (FOWLER, 2006).

Objetivos

O objetivo desta pesquisa é minimizar os esforços gastos no decorrer da etapa de análise, com o auxílio da ferramenta Cucumber que emprega requisitos de software baseados em exemplos (testes). Estes testes ilustram os requisitos e as regras de negócio, que asseguram que as especificações estejam sempre sincronizadas.

Organização do Trabalho

Este trabalho esta dividido da seguinte maneira:

- No capítulo I, são apresentados os conceitos e os processos da Engenharia de Requisitos.
- No capítulo II, são apresentadas as técnicas da Especificação por Exemplo (*Specification By Example*) e seus benefícios.
- No capítulo III, a ferramenta Cucumber é apresentada através de suas terminologias e definições.
- No capítulo IV, outras ferramentas são mencionadas através da perspectiva de outros autores.
- No capítulo V, é apresentada através de um caso de sucesso, a melhoria da qualidade no requisito de software.

CAPITULO I

1. ENGENHARIA DE REQUISITOS

A Engenharia de Requisitos é uma atividade extremamente importante dentro do desenvolvimento de um sistema, é ela que vai determinar o sucesso ou o fracasso do projeto, pois, mesmo que o sistema tenha sido bem projetado e codificado, se ela for mal especificada, haverá grandes possibilidades dela causar grandes prejuízos e transtornos para todos os envolvidos.

> A Engenharia de Requisitos é o processo de compreensão e definição dos serviços requisitados do sistema e identificação de restrições relativas à operação e ao desenvolvimento do sistema. A engenharia de requisitos é um estágio particularmente crítico do processo de software, pois erros nessa fase inevitavelmente geram problemas no projeto e na implementação do sistema (SOMMERVILLE, 2011, p. 24).

Sommerville ainda complementa dizendo que:

> [...] os requisitos de um sistema são as descrições do que o sistema deve fazer, os serviços que oferece e as restrições a seu funcionamento. Esses requisitos refletem as necessidades dos clientes para um sistema que serve a uma finalidade determinada, como controlar um dispositivo, colocar um pedido ou encontrar informações. O processo de descobrir, analisar, documentar e verificar esses serviços e restrições é chamado engenharia de requisitos (SOMMERVILLE, 2011, p. 57).

Para Nuseibeh e Easterbrook (2000, apud BOKHARI e SIDDIQUI 2011, tradução nossa) a engenharia de requisitos é o processo de descobrir as necessidades dos *Stakeholders*

e documentá-los para a comunicação, análise e implementação, e é sugerido por LARMAN (1997) e JACOBSON et al.(1998) apud DALLAVALLE et al. (2000), os seguintes artefatos para a fase de Engenharia de Requisitos: avaliação do ambiente, conhecer quem são os clientes, as metas do negócio, as funções e atributos do sistema.

> Segundo Thayer (1997, apud BELGAMO e MARTINS 2000, p. 02), a engenharia de requisitos é a primeira etapa dentro de todo o processo da engenharia de software, a qual estuda como coletar, entender, armazenar, verificar e gerenciar os requisitos. A principal preocupação na engenharia de requisitos é entender quais são os reais requisitos do sistema, bem como a documentação dos mesmos.

(TELES, 2005) [...] deficiências na participação do cliente têm sido apontadas como um dos principais fatores a gerar falhas nos projetos de software, para Brooks (2009, p. 192), a parte mais difícil da construção de um sistema de software é a decisão precisa sobre o que construir. Nenhuma outra parte do trabalho conceitual é tão difícil quanto estabelecer os requisitos técnicos detalhados, incluindo as interfaces para as pessoas, para as máquinas e para os outros sistemas de software. Nenhuma outra parte prejudica tão seriamente o sistema resultante se for feita de maneira errada. Nenhuma outra parte é difícil de corrigir depois. Portanto, a função mais importante que os construtores de software exercem em benefício de seus clientes é a extração sucessiva e o refinamento dos requisitos do produto. Mas, a verdade seja dita, os clientes não sabem o que querem. Eles normalmente não sabem quais perguntas devem ser respondidas e quase nunca pensaram sobre o problema que deve ser especificado.

> A falta de envolvimento dos usuários tradicionalmente tem sido causa número um das falhas nos projetos. No sentido oposto, a contribuição número um para o sucesso de um projeto tem sido envolvimento dos usuários. Mesmo quando entregue no prazo e dentro do orçamento, um projeto pode falhar se não tratar das necessidades e expectativas dos usuários. THE STANDISH GROUP INTERNATIONAL, (2001, p. 4, apud TELES 2005).

De acordo com a pesquisa do Relatório do Caos (The Chaos Report)[1] THE STANDISH GROUP, (1995), foram identificados que há falhas no escopo do projeto de software e que projetos de desenvolvimento de software é um caos. Foram citadas as três principais razões para que um projeto seja bem sucedido, que é ter a necessidade da participação do usuário; ter apoio da gestão e ter uma declaração clara dos requisitos.

---

[1] The Chaos Report é um estudo que a empresa The Standish Group, (localizada em Massachusetts, EUA), publica desde 1994. Trata-se de um amplo levantamento envolvendo milhares de projetos na área de tecnologia da informação. Atualmente, seus dados estão entre os mais utilizados para quantificar a "crise do software".

Entretanto, de acordo com THE STANDISH GROUP (2001) apud TELES (2005), The Chaos Report do ano de 2000 apresenta uma coletânea de dados envolvendo 280 mil projetos de software nos Estados Unidos, os quais revelaram que:

- Em média, os atrasos apresentam 63% mais tempo do que o estimado;

- Os projetos que não cumpriram o orçamento custaram em média 45% mais e no geral, apenas 67% das funcionalidades prometidas foram efetivamente entregues.

O STANDISH GROUP (2001, apud TELES 2005) classifica o resultado final de um projeto nas categorias de mal sucedido; comprometidos e fracassados, conforme figura abaixo.

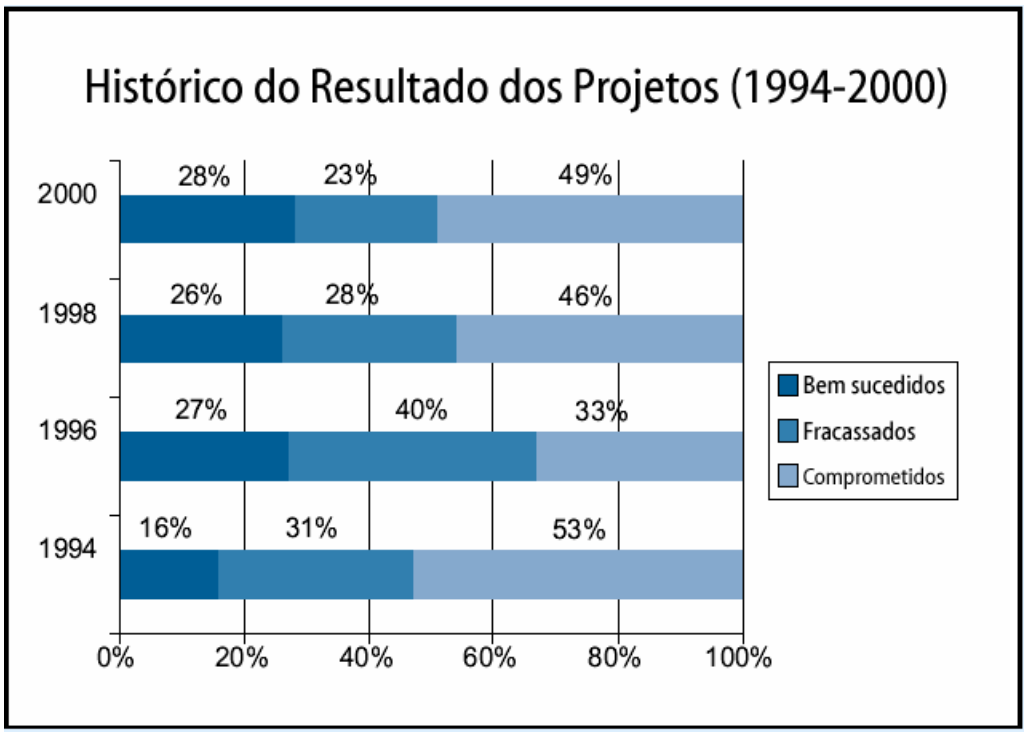

Figura 1. Evolução do resultado final de Projetos de Software ao longo dos anos.

Fonte: THE STANDISH GROUP (2001) apud TELES (2005)

Tais números, embora desastrosos, mostram um avanço em relação aos resultados do primeiro levantamento realizado em 1994.

> JACOBSON et al (1998) apud DALLAVALLE et al. (2000), afirma que atualmente muitas pessoas desenvolvem software usando os mesmos métodos de 25 anos atrás, e que sem atualizá-los não será possível atingir a meta de desenvolvimento de software complexos exigidos pelo mercado atual.

As ideias de Brooks (2010) apontam para a mesma direção, em uma entrevista dada a revista WIRED, Brooks argumenta:

> Quando eu escrevi O Mítico Homem-Mês, em 1975, eu aconselhei os programadores a descartar a primeira versão de seus sistemas e então construir uma segunda. Na edição comemorativa do 20º aniversário do livro, eu percebi que a iteração incremental constante é uma abordagem muito melhor. Você constrói um protótipo rápido e o coloca na frente dos usuários para ver o que eles farão com ele. Você sempre será surpreendido (BROOKS, 2010).

Apesar das mudanças alcançadas com a Engenharia de Software, alguns problemas continuam existindo. Brooks (2009, p. 24) acredita que "projetos de software falharam, em sua maioria, mais por falta de tempo no calendário do que em função da combinação de todas as outras causas". THE STANDISH GROUP INTERNACIONAL (2001, p. 4) apud TELES (2005, p. 74) também aponta que o tempo é o inimigo de todos os projetos. Visto que o escopo impacta na duração do projeto, ambos estão associados. Minimizando o escopo, o tempo é reduzido e, portanto, as chances de sucesso crescem.

Sommerville (2011, p. 157) ainda assim, acredita que um dos princípios gerais das boas práticas de engenharia de requisitos é que os requisitos devem ser testáveis, isto é, o requisito deve ser escrito de modo que um teste possa ser projetado para ele. Um testador pode, então, verificar se o requisito foi satisfeito. Testes baseados em requisitos é uma abordagem sistemática para projeto de casos de teste em que você considera cada requisito e deriva um conjunto de testes para eles. Testes baseados em requisitos são mais uma validação do que um teste de defeitos você está tentando demonstrar que o sistema implementou adequadamente seus requisitos.

A partir disso, você pode ver que testar um requisito não significa simplesmente escrever um único teste. Normalmente, você precisa escrever vários testes para garantir a cobertura dos requisitos. Você também deve manter registros de rastreabilidade de seus testes baseados em requisitos, que ligam os testes aos requisitos específicos que estão sendo testados.

De acordo com o nível zero de maturidade para a gestão de requisitos, Heumann (2003, tradução nossa) afirma que as organizações não investem nos requisitos, pois consideram esta tarefa pouco necessária. Algumas vezes isso funciona, porém, com uma maior frequência o produto é lançado com ausência de funcionalidades, funções são entregues desnecessariamente ou com má qualidade. [...] quando os erros são detectados na fase posterior ao desenvolvimento de software é mais caro corrigi-los que no estágio inicial, pois quando erros não são detectados na fase de requisitos leva-se ao desenvolvimento errado do produto gerando um desperdício valioso de recursos. (DAVIS, 2003 apud BOKHARI e

SIDDIQUI, 2011, tradução nossa). Teles (2004, p. 151 apud TONIAZZO, 2007, p. 54) ainda afirma que o custo de se corrigir um problema em um software cresce exponencialmente ao longo do tempo. Um problema que poderia ter custado um dólar para ser corrigido se tivesse sido encontrado durante a análise pode custar milhares de dólares para ser resolvido depois que o software já estiver em produção.

Apesar dos diversos problemas que aparecem devido ao alto grau de incerteza existente no processo de desenvolvimento de software, consequente da constante mudança de requisitos e prioridades, a Engenharia de Requisitos surge para tentar minimizar esses problemas, entretanto, obter o entendimento do processo de engenharia de requisitos é fundamental, tanto para o sucesso do desenvolvimento do projeto, quanto para entender as necessidades reais dos usuários, uma vez que para Bokhari e Siddiqui (2011, tradução nossa) a engenharia de requisitos é um problema crítico que se deve a falta de envolvimento com os *Stakeholders* nas fases dos processos de requisitos.

No entanto para Macaulay, (1996, tradução nossa) a comunicação entre as pessoas é importante em todo o processo da engenharia de requisitos a fim de produzir os documentos e outros artefatos necessários, além do bom desenvolvimento do conhecimento e do entendimento entre os participantes.

> Para Sommerville (2009, p.24.69) o processo de engenharia de requisitos tem como objetivo produzir um documento de requisitos acordados que especifica um sistema que satisfaz os requisitos dos *Stakeholders*. Esses processos incluem quatro atividades de alto nível, elas visam avaliar se o sistema é útil para a empresa (estudo de viabilidade), descobrindo os requisitos (elicitação e análise), convertendo-os em alguma forma padrão (especificação) e verificando se os requisitos realmente definem o sistema que o cliente quer (validação).

Segundo Thayer (1997, apud BELGAMO e MARTINS, 2000, p. 02), a principal preocupação na engenharia de requisitos é entender quais são os reais requisitos do sistema, bem como a documentação dos mesmos, e conforme especificado por Thayer, as fases da Engenharia de Requisitos são: elicitação, análise, especificação, verificação e gerenciamento.

Kotonya e Sommerville (1998, apud CHICHINELLI, 2002, p. 21), acredita que os processos de Engenharia de Requisitos são muito variáveis. Partindo de processos pouco estruturados até processos sistemáticos, baseados na aplicação de alguma metodologia de análise e para eles o modelo de processo mais usado na Engenharia de Requisitos envolve quatro atividades: elicitação de requisitos, análise de requisitos e negociação, documentação de requisitos e validação de requisitos.

[...] “embora os métodos estruturados tenham um papel a desempenhar no processo de engenharia de requisitos, existe muito mais para a engenharia de requisitos do que o que é coberto por esses métodos” (SOMMERVILLE, 2009, p. 69).

Pode-se notar que independente do processo de Engenharia de Requisitos utilizado, o resultado é um documento de requisitos de software e que todavia, não existe um processo ideal para a Engenharia de Requisitos.

1.1. Processos da Engenharia de Requisitos

O processo da engenharia de requisitos é composto basicamente pelas seguintes atividades:

- Elicitação de Requisitos;
- Análise de Requisitos;
- Especificação de Requisitos e
- Validação de Requisitos.

No entanto, na prática, essas atividades não são feitas em apenas uma sequencia e sim [...] simultaneamente e incrementalmente, num processo evolutivo que dura todo o processo de desenvolvimento do software (JACOBSON et al. 1999).

[...] Várias são as formas de se caracterizar o processo da Engenharia de Requisitos, variando segundo fases e os autores; daí encontra-se muitas inconsistências na terminologia utilizada na área (DAVIS, 2003 apud ALENCAR 1999).

Alguns problemas com requisitos encontrados durante o processo de engenharia de requisitos são relatados por Kotonya et al. (1997):

- Os requisitos não refletem as reais necessidades do cliente em relação ao sistema a ser desenvolvido;
- Os requisitos são inconsistentes e/ou incompletos;
- É dispendioso fazer mudanças após os requisitos terem sido acordados entre as partes (cliente e equipe de desenvolvimento);
- É comum ocorrer interpretação errônea entre clientes e equipe de desenvolvimento.

A Figura 2, conforme mostrada abaixo, contém várias atividades que se repetem até que o documento de requisito seja aceite. Quando a versão provisória do documento de requisito tem problema volta-se a entrar na espiral repetindo-se todas as atividades.

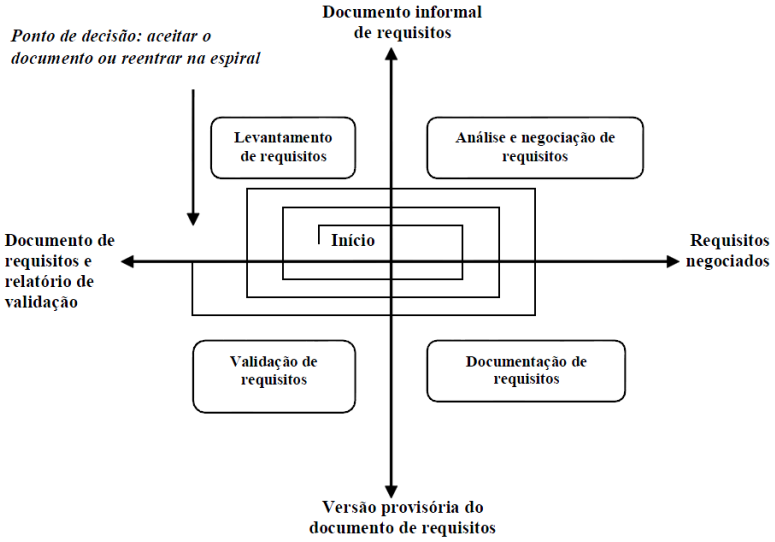

Figura 2. Modelo em espiral do processo de Engenharia de Requisitos.

Fonte: Kotonya and Sommerville, 1998 apud Silveira (2006).

1.2. Elicitação de Requisitos

A Elicitação de Requisitos é o início para toda a atividade de desenvolvimento de software, e não acontece somente uma vez, seu processo é iterativo.

"Ao iniciar a Elicitação de Requisitos, os usuários podem ter um forte papel no processo de desenvolvimento, através de seus conhecimentos e experiências" (GOGUEN, 1997 apud BELGAMO e MARTINS, 2000, p. 02).

A escolha dos usuários no projeto é de extrema importância, pois a escolha de uma pessoa errada acarretará em uma perda de tempo e dinheiro para ambas as partes envolvidas no desenvolvimento do software. McConnel (1998 apud BELGAMO e MARTINS, 2000, p. 02) cita que é na iteração com o usuário é que nascem os requisitos para a construção do software.

Goguen (1997 apud BELGAMO e MARTINS, 2000, p. 02) ainda menciona que a natureza dos requisitos é mutante ao iniciarmos a fase de Elicitação de Requisitos, pois, os clientes podem mudar seus pensamentos, uma vez que eles veem várias possibilidades mais claramente em momentos posteriores.

Já para Sommerville (2009, p. 69.72) nessa atividade de elicitar requisitos "os engenheiros de software trabalham com os clientes e usuários finais do sistema para obter

informações sobre o domínio da aplicação, os serviços que o sistema deve oferecer, o desempenho do sistema, restrições de hardware e assim por diante". Sommerville, também argumenta que a descoberta de requisitos (ás vezes, chamada de elicitação de requisitos) é o processo de reunir informações sobre o sistema requerido e os sistemas existentes e separar dessas informações os requisitos de usuários e de sistema.

"Algumas técnicas são utilizadas para descobrir (elicitar) os requisitos do sistema a ser desenvolvido, como entrevista, reuniões e questionários" (GOGUEN, 1994). Nessas entrevistas conforme apontado por Sommerville (2009, p. 72.73), a equipe de engenharia de requisitos questiona os *Stakeholders* sobre o sistema que usam no momento e sobre o sistema que será desenvolvido. Requisitos surgem a partir das respostas a essas perguntas, porém, Sommerville também argumenta que entrevistas não é uma técnica eficaz para a elicitação do conhecimento sobre os requisitos e restrições organizacionais, porque, entre as diferentes pessoas da organização, existem sutis relações de poder. [...] os entrevistados podem preferir não revelar a estrutura real, e sim a estrutura teórica, para um estranho.

Outra técnica para elicitar requisitos, são os cenários, (SOMMERVILLE, 2009, p. 73) que podem ser particularmente úteis para adicionar detalhes a uma descrição geral de requisitos. A Elicitação baseada em cenários envolve o trabalho com os *Stakeholders* para identificar cenários e capturar detalhes que serão incluídos nesses cenários. Os cenários podem ser escritos como texto, suplementados por diagramas e telas. As pessoas geralmente acham mais fácil se relacionar com exemplos da vida real do que com descrições abstratas. Elas podem compreender e criticar um cenário de como elas podem interagir com um sistema de software. As histórias são um tipo de cenário de requisitos.

De acordo ainda com Sommerville, o processo de elicitar e compreender os requisitos dos *Stakeholders* do sistema são difíceis por várias razões:

- Exceto em termos gerais, os *Stakeholders* costumam não saber o que querem de um sistema computacional; eles podem achar difícil articular o que querem que o sistema faça, e, como não sabem o que é viável e o que não é, podem fazer exigências inviáveis.
- Naturalmente, os *Stakeholders* expressam requisitos em seus próprios termos e com o conhecimento implícito de seu próprio trabalho. Engenheiros de requisitos, sem experiência no domínio do cliente, podem não entender esses requisitos.
- Diferentes *Stakeholders* têm requisitos diferentes e podem expressar essas diferenças de várias maneiras. Engenheiros de requisitos precisam descobrir todas as potenciais fontes de requisitos e descobrir as semelhanças e conflitos.

- Fatores políticos podem influenciar os requisitos de um sistema. Os gerentes podem exigir requisitos específicos, porque estes lhes permitirão aumentar sua influência na organização.
- O ambiente econômico e empresarial no qual a análise ocorre é dinâmico. É inevitável que ocorram mudanças durante o processo de análise. A importância dos requisitos específicos pode mudar. Novos requisitos podem surgir a partir de novos *Stakeholders* que não foram inicialmente consultados (SOMMERVILLE, 2009, p. 71).

Para McConnel (1998 apud BELGAMO e MARTINS 2000, p. 02), porém, a parte mais difícil da elicitação de requisitos não é o ato de registrar o que os usuários querem; pois a atividade de elicitação é exploratória e desenvolvimental, na verdade estamos ajudando os usuários a compreenderem melhor o que eles querem.

Sommerville (2009, p.69) acredita que a elicitação de requisitos, em particular, é uma atividade centrada em pessoas, e as pessoas não gostam de restrições impostas por modelos rígidos de sistema.

1.3. Análise de Requisitos

Após o início da atividade de elicitação, a análise também pode ser iniciada, uma vez que, os problemas são descobertos quando os requisitos estão sendo elicitados. Os requisitos também precisam ser analisados, pois nesta ocasião são identificadas inconsistências de informações fornecidas por diferentes fontes e diferentes perspectivas de cada um dos grupos de usuários, inadequações e ambiguidades que possam existir segundo a lista dos requisitos elicitados.

Segundo Kotonya (1998, apud CHICHINELLI, 2002), o processo de análise de requisitos é o momento em que diferentes *Stakeholders* negociam para decidir que requisitos serão aceitos. Este processo é necessário porque existem inevitáveis conflitos entre os requisitos de diferentes origens, ou a informação pode estar incompleta ou a maneira que os requisitos foram descritos pode estar incompatível com o orçamento disponível para desenvolver o sistema. Há sempre alguma flexibilidade nos requisitos e a negociação é necessária para decidir o acordado conjunto de requisitos para o sistema.

De acordo com Pressman (1995 apud CHICHINELLI, 2002), a tarefa de análise de requisitos é um processo de descobertas, refinamento, modelagem e especificação. Esta tarefa efetua a ligação entre o projeto de software e o sistema que está sendo estudado em nível de

software. Proporciona ao engenheiro de software a representação da informação e funções a serem traduzidas em um projeto procedimental, arquitetônico e de dados. Permite assim, que desenvolvedor e cliente possuam critérios de avaliação de qualidade assim que o software estiver concluído.

Enquanto que para Sommerville, (2009, p. 25) [...] essa parte do processo pode envolver o desenvolvimento de um ou mais modelos de sistemas e protótipos, os quais nos ajudam a entender o sistema a ser especificado.

Anos mais tarde Pressman (2006, p.144) argumenta que:

> [...] a análise de requisitos resulta na especificação das características operacionais do software; indica interface do software com outros elementos do sistema e estabelece restrições a que o software deve satisfazer e ainda afirma que a análise de requisitos fornece ao projetista de software uma representação da informação, função e comportamento, que podem ser traduzidos para os projetos arquiteturais, de interface e em nível de componentes.

Contudo, Pressman ainda define que a análise de requisitos deve atingir os três objetivos principais que são: descrever o que o cliente exige; estabelecer a base para a criação de um projeto de software e definir um conjunto de requisitos que possam ser validados quando o software for construído.

Por fim, o modelo de análise de requisitos proporcionam ao desenvolvedor e ao cliente os meios de avaliar a qualidade quando o software é construído (PRESSMAN 2006, p.145).

1.4. Especificação de Requisitos

O documento de requisitos de software, chamado de especificação de requisitos é uma declaração oficial de que os desenvolvedores do sistema devem implementar (SOMMERVILLE, 2009, p. 63). E em geral, há a necessidade de um documento que seja compreendido por todos os *Stakeholders* do sistema (KOTONYA, 1998).

“Acredito que a parte mais difícil na construção de um software é a especificação” Brooks (2009, p. 176).

> De acordo com Sommerville (2009, p. 60) a imprecisão na especificação de requisitos é a causa de muitos problemas da engenharia de software. É compreensível que um desenvolvedor de sistemas interprete um requisito ambíguo de uma maneira que simplifique sua implementação. Muitas vezes, porém, essa não é a preferência do cliente, sendo necessário, então, estabelecer novos requisitos e

> fazer alterações no sistema. Naturalmente, esse procedimento gera atrasos de entrega e aumenta os custos.

Segundo Thayer (1997 apud BELGAMO e MARTINS) especificação é o processo de criação de um documento no qual estão definidos todos os requisitos analisados. Enquanto que, para Sommerville (2009, p. 42) o documento de requisitos do sistema, é o documento-chave que informa ao engenheiro de software o que o sistema deve fazer. Sem esse conhecimento, é difícil avaliar o impacto das mudanças propostas.

Para o Guia da Engenharia de Software (**SWEBOK**, 2004, tradução nossa) a especificação de requisito de software se refere normalmente à produção de um documento, que pode ser sistematicamente analisado, avaliado e aprovado e ainda estabelece uma base para um acordo entre os clientes, prestadores de serviços ou fornecedores do que o produto faz, bem como o que não é esperado fazer.

A especificação de requisitos [...] é a base para as demais atividades de desenvolvimento e sua qualidade é fundamental para o sucesso do projeto (STÁBILE; CAZARINI, 2003), e se bem elaborada é pré-requisito para um software de qualidade, embora não seja garantia disso.

> Pressman (2006, p.120) argumenta que a especificação é o produto de trabalho final produzido pelo engenheiro de requisitos. Ela serve como fundamento das atividades de engenharia de software subsequentes. Ela descreve a função e o desempenho de um sistema baseado em computador e as restrições que governarão o seu desenvolvimento.

Para Sommerville (2011, p. 58, 65) a especificação de requisitos é o processo de escrever os requisitos do usuário (declarações de quais serviços o sistema deverá fornecer a seus usuários e as restrições com as quais este deve operar) e os requisitos do sistema (descrições mais detalhadas das funções, serviços e restrições operacionais do sistema de software, [...] não apenas especificam os serviços ou as características necessárias ao sistema, mas também a funcionalidade necessária para garantir que esses serviços/características sejam entregues corretamente) em um documento de requisitos. Os requisitos de usuário e de sistema devem ser claros, inequívocos, de fácil compreensão, completos e consistentes. Na prática, isso é difícil de conseguir, pois os *Stakeholders* interpretam os requisitos de maneiras diferentes, e, muitas vezes notam-se conflitos e inconsistências inerentes aos requisitos.

McEwen (2004, tradução nossa) apresenta algumas qualidades que deveriam caracterizar um documento de especificação de requisito de software:

- Falta de ambiguidade: a equipe de desenvolvimento de software será incapaz de produzir um produto que satisfaça as necessidades dos usuários se um ou mais requisitos forem interpretados de várias maneiras. Sommerville (2011) também argumenta que ter ambiguidade é um problema, já que uma das formas de documentar requisitos é através de documentos escritos muitas vezes em linguagem natural. A sua compreensão depende de uso das mesmas palavras para os mesmos conceitos, pelos leitores e por quem escreve as especificações e, além disso, um determinado requisito pode ser expresso de modos completamente diferentes;
- Completo: no começo do projeto, você não deve esperar conhecer todos os requisitos do sistema com detalhe, a equipe de desenvolvimento não deve gastar tempo tentando especificar coisas que vão evoluir. [...], no entanto, você deve manter seu documento de especificação de requisito de software atualizado, pois quanto mais conhecimento tiver sobre o sistema, mais o documento de especificação crescerá completo;
- Consistência: você não pode construir um sistema que satisfaça todos os requisitos, se há dois requisitos conflitantes ou se os requisitos não refletirem as mudanças que foram feitas para o sistema durante o desenvolvimento iterativo e testes de funcionalidade;
- Rastreabilidade: a equipe deve rastrear a origem de cada requisito, se ele evoluiu a partir de um requisito mais abstrato, ou se foi de um encontro específico com o usuário. Sommerville (2000, apud PUC-RIO) define rastreabilidade como uma propriedade de uma especificação de requisitos que reflete a habilidade de encontrar requisitos relacionados;
- Nenhuma informação do projeto: enquanto um requisito abordar um comportamento externo, como visto por usuários ou por outras interfaces de sistemas, eles ainda serão requisitos, indiferente dos seus níveis de detalhes. No entanto, se um requisito tenta especificar subcomponentes particulares ou seus algoritmos, isto já não é um requisito, o mesmo tornou-se uma informação do projeto.

Conforme mencionado, McEwen apresenta algumas qualidades que um documento de especificação de requisito deveria ter. Macaulay (1996, tradução nossa) complementa, enfatizando que o documento de requisito deveria conter declarações que são inequívocas,

completa, verificável, consistente, modificável, rastreável e utilizável durante as fases de operação e manutenção, assim como, a prática recomendada para especificações de exigências de software (IEEE Std 830), nela são descritas as características de um bom documento de especificação de requisitos, conforme descrito a seguir:

- Correto: se cumpriu todas as exigências correspondidas pelo software;
- Não ambíguo: se têm apenas uma única interpretação;
- Completo: para Sommerville (2011, p. 60) significa que todos os serviços requeridos pelo usuário devem ser definidos;
- Consistente: se, nenhum subconjunto individual de exigências descrito nele entra em conflito. Para Sommerville (2011) significa que os requisitos não devem ter definições contraditórias;

Porém, na prática, para Sommerville (2011, p. 60):

> [...] sistemas grandes e complexos, é praticamente impossível de se alcançar a completude e consistência dos requisitos. Uma razão para isso é que ao elaborar especificações para sistemas complexos é fácil cometer erros e omissões. Outra razão para isso é que em um sistema de grande porte existem muitos *Stakeholders*. Os *Stakeholders* têm necessidades diferentes e, muitas vezes, inconsistentes. Essas inconsistências podem não ser evidentes em um primeiro momento, quando os requisitos são especificados, e, assim, são incluídas na especificação. Os problemas podem surgir após uma análise mais profunda ou depois de o sistema ter sido entregue ao cliente.

- Classificável: por importância e/ou estabilidade; se cada exigência nele contido tem associado um identificador de estabilidade e/ou importância;
- Verificável: se, cada exigência especificada é verificável. Uma exigência é verificável, se e só se, existe um processo finito e de custo aceitável através do qual uma pessoa ou uma máquina pode verificar que o produto de software cumpre essa exigência. Em geral uma exigência ambígua não é verificável. Exigências não verificáveis incluem por norma frases tais como "trabalha bem", "boa interface" ou "irá acontecer normalmente". Estas exigências não podem ser verificadas, pois, não é possível definir os termos "bom", "bem" ou "normalmente". A exigência "O programa nunca entrará num ciclo infinito" embora esteja definida objetivamente, não é verificável, pois testar esta qualidade é teoricamente impossível;

- Modificável: se, a sua estrutura e estilo são tais que, mudanças a exigências podem ser efetuadas de forma fácil, completa e consistente, preservando simultaneamente estrutura e estilo.
- Rastreável: se cada uma das suas exigências é clara e facilitadora da identificação da mesma exigência em versões futuras do desenvolvimento ou da documentação.

Sommerville (2011, p. 176) na sua concepção, sintetiza da seguinte maneira as propriedades de um sistema de software através da complexidade, conformidade, flexibilidade e invisibilidade.

- Complexidade: Entidades de software são mais complexas que, qualquer outra coisa produzida pelo homem. Da complexidade vêm a dificuldade de comunicação entre os membros da equipe, que leva a deficiência do produto, aumento dos custos, atrasos de cronograma. Da complexidade também vem à dificuldade de enumerar, e muito menos entender, todos os estados possíveis de um programa, e daí vem à falta de confiabilidade. Da complexidade das funções vem à dificuldade de utilizá-las, o que torna os programas difíceis de serem utilizados. Da complexidade da estrutura vem à dificuldade de ampliar os programas com novas funções se, com isso, criar efeitos colaterais. Da complexidade da estrutura vêm também os estados não percebidos, que acabam por criar vulnerabilidade de segurança.
- Conformidade: A maior parte da complexidade que o engenheiro de software deve dominar é arbitrária e sem rima ou razão por muitos sistemas e instituições humanas às quais suas interfaces devem estar em conformidade.
- Mutabilidade: A entidade de software é constantemente sujeita a pressões em prol de uma mudança. O software pode ser modificado com mais facilidade, pois, é pura matéria de pensamento, infinitamente maleável.
- Invisibilidade: Software é invisível e é impossível criar uma boa representação visual dele. Não há representação geométrica pronta para representá-lo, da mesma forma que um terreno tem mapas, circuitos integrados têm seus diagramas.

Cosgrove (apud BROOKS, p. 114) cita que a relutância em documentar projetos não é por simples preguiça ou falta de tempo. Ao contrário, ela decorre da relutância do projetista em comprometer-se com a defesa de decisões que não passam de tentativas, "ao documentar

um projeto, o projetista expõe-se à críticas de todos e ele deve ser capaz de defender tudo o que escreve. Se, de qualquer maneira, a estrutura organizacional representa uma ameaça, nada será documentado até que seja inteiramente defensável".

Pois, conforme o nível 1 do Modelo de Maturidade da Gestão de Requisitos relatado por Heumann (2003, tradução nossa) [...] uma vez escrito os requisitos, vários benefícios se tornam evidente, pois, além de você ter um contrato real com o cliente, se você escreve bem, os requisitos podem indicar claramente o seu entendimento do que o cliente quer que você construa.

Faulk (2007) ainda aponta, que dentre os benefícios obtidos pelos documentos gerados na fase de especificação, pode-se citar que o documento de especificação:

- É o veículo básico de comunicação entre desenvolvedores e usuários sobre o que deve ser construído;
- Registra os resultados da análise do problema (obtido através da elicitação e análise dos requisitos);
- Definem quais propriedades o sistema deve ter e quais são as restrições impostas em seu projeto e implementação;
- É a base para estimativas de custo e cronograma;
- É a base para o desenvolvimento do plano de teste do sistema;
- Oferece uma definição padrão de comportamento esperado pelos profissionais envolvidos na manutenção do sistema e
- É utilizado para registrar mudanças na engenharia do sistema.

Uma vez identificados e analisados, os requisitos dever ser documentados para que possam servir de base para o restante do processo de desenvolvimento.

Para Adzic (2011, p. 125, tradução nossa) as especificações devem ser autoexplicativas para evitar qualquer desentendimento que os desenvolvedores tenham ao especificar a funcionalidade pela primeira vez. E complementa dizendo que, para garantir que a especificação seja autoexplicativa, pergunte para outra pessoa o que ela entendeu e veja se corresponde com a sua intenção. [...] assim ao planejar qualquer atividade de software, é necessário permitir uma interação extensa entre o cliente e o projetista como parte da definição do sistema.

Brooks (2001, p. 128) resume dizendo: “é bem melhor o excesso do que a grave falta de documentação que caracteriza a maioria dos sistemas de programação”.

1.5. Validação de Requisitos

Brooks (2009, p. 137) argumenta que: muito antes de qualquer código existir, a especificação deve ser entregue a um grupo externo de testes para verificar se ela esta completa e clara. Como assinalada Vyssotsky (apud BROOKS, 2009), os desenvolvedores não podem fazer isso sozinhos: "Eles não dirão a você que eles não entenderam a especificação. Eles inventarão, com muita alegria, meios para resolver as lacunas e partes obscuras".

A atividade de testes de software é uma das atividades mais onerosas do processo de produção de um software. Estudos indicam que testes consomem mais de 50% (cinquenta) do custo de desenvolvimento de software (HARROLD, 2000, apud BUHLER, 2007).

Teles (2004, apud TONIAZZO, 2007) cita que testes é uma parte do desenvolvimento que todos têm consciência de sua importância, mas ninguém quer fazer. É uma tarefa considerada maçante, repetitiva e que atrasa o desenvolvimento. Dessa forma o ato de testar torna-se mera formalidade, executado de forma superficial e muitas vezes errônea. Esse fator resulta em erros consideráveis em fases adiantadas do projeto.

Na tentativa de reduzir estes custos, têm sido propostos técnicas, critérios e ferramentas que auxiliam na condução e avaliação do teste de software. E uma dessas técnicas é chamada de Técnica de Especificação por Exemplo e é apresentada com mais detalhe na capítulo 2.

Antes de iniciarmos o entendimento sobre a validação de requisitos, é fundamental, deixar claro que verificação de requisitos e validação de requisitos não são as mesmas coisas, embora sejam frequentemente confundidas, pois, conforme Sommerville (2011, p. 145) ambas objetivam verificar se o software em desenvolvimento satisfaz suas especificações e oferece a funcionalidade esperada pelas pessoas que estão pagando pelo software.

Boehm (1979, apud SOMMERVILLE, 2011) expressou sucintamente a diferença entre validação e verificação:

- 'Verificação: estamos construindo o produto da maneira certa?'.
- 'Validação: estamos construindo o produto certo?'.

Para Sommerville, o objetivo da verificação é checar se o software atende a seus requisitos funcionais e não funcionais, enquanto que, na validação o objetivo é garantir que o software atenda às expectativas do cliente. Ele vai além da simples verificação de conformidade com as especificações, pois tenta demonstrar que o software faz o que o cliente espera que ele faça. A validação é essencial porque, especificações de requisitos nem sempre refletem os desejos ou necessidades dos clientes e usuários do sistema.

Pressman (2006, p. 289) possui uma ideia semelhante para ambos os processos. O mesmo declara que a verificação se refere ao conjunto de atividades que garante que o software implementa corretamente uma função específica e a validação se refere a um conjunto de atividades diferentes que garante que o software construído corresponde aos requisitos do cliente.

Para Pfleeger (2004 apud ANDRADE, 2010), a validação é todo procedimento que busca assegurar que o sistema de fato implementou os requisitos que se espera que estejam implementados, enquanto que a verificação é o que visa garantir que cada função do software está operando conforme o esperado.

Thayer (1997 apud BELGAMO e MARTINS, 2000) complementa dizendo que, a verificação de requisitos é o processo que busca assegurar que a especificação de requisitos de software está em concordância com os requisitos elicitados e analisados nas fases anteriores, ou seja, estão de acordo com o desejo do cliente.

Entretanto, para Brooks (2009, p. 189) [...] mesmo a perfeita verificação de requisito ([...] processo este, iniciado assim que os requisitos estão disponíveis e continuam em todas as fases do processo de desenvolvimento (SOMMERVILLE, 2011, p. 145) pode apenas estabelecer que um requisito atenda a sua especificação. A parte mais difícil da tarefa de desenvolvimento de software é chegar a uma especificação completa e consistente, e muito da essência da construção de um programa é, de fato, a depuração da especificação.

McConnell (2004 apud PINTO SILVEIRA, 2006) define que a validação de requisitos corresponde ao procedimento de verificação dos requisitos quanto à validade, consistência, integridade, realismo e certeza. Pretende-se descobrir os erros no documento de requisitos, ou seja, os problemas, as omissões e as ambiguidades.

Pressman, (2006, p. 120) também possui uma definição sólida sobre a validação de requisitos, o mesmo afirma que, o processo de validação examina a especificação para garantir que todos os requisitos do software tenham sido declarados de modo não ambíguo; e que as inconsistências, omissões e erros tenham sido detectados e corrigidos e que os produtos de trabalho estejam de acordo com as normas estabelecidas para o processo, o projeto e o produto. A equipe que valida os requisitos inclui engenheiros de software, clientes, usuários e outros interessados que examinam a especificação procurando por erros de conteúdo ou de interpretação, áreas em que esclarecimentos podem ser necessários, informações omissas, inconsistências (um problema importante quando produtos ou sistemas de grande porte passam por engenharia), requisitos conflitantes ou requisitos irrealísticos (inatingíveis).

Diversas definições são apresentadas para a validação de requisitos, no entanto, a definição defendida por Sommerville nos dá uma ampla compreensão do processo de validação de requisitos.

Para Sommerville (2011, p. 76) a validação de requisitos é o processo pelo qual se verifica se os requisitos definem o sistema que o cliente realmente quer. Ela se sobrepõe à análise, uma vez que está preocupada em encontrar problemas com os requisitos. A validação de requisitos é importante porque erros em documento de requisitos podem gerar altos custos de retrabalho quando descobertos durante o desenvolvimento ou após o sistema já estar em serviço. O custo para consertar um problema de requisitos por meio de uma mudança no sistema é geralmente muito maior do que o custo de consertar erros de projeto ou de codificação. A razão para isso é que a ocorrência de mudança dos requisitos normalmente significa que o projeto e a implementação do sistema também devem ser alterados. Além disso, o sistema deve, posteriormente, ser retestado.

McConnell (2004 apud PINTO SILVEIRA, 2006) também acredita que a fase de validação de requisitos é importante, porque os erros no documento de requisitos podem conduzir a custos excessivos. Estes erros, quando detectados em fases avançadas do processo de desenvolvimento de software ou depois do sistema estar em funcionamento, são muito mais difíceis de corrigir do que quando são detectados logo na fase de análise de requisitos. Por exemplo, um erro nos requisitos detectado na codificação é cerca de 5-10 vezes mais caro de corrigir do que se tivesse sido detectado durante a fase de requisitos.

Sommerville ainda afirma que, durante o processo de validação de requisitos, diferentes tipos de verificação devem ser efetuados com os requisitos no documento de requisitos. Essas verificações incluem:

- Verificações de consistência: Requisitos no documento não devem entrar em conflito, ou seja, não deve haver restrições contraditórias ou descrições diferentes da mesma função do sistema.
- Verificações de completude: o documento de requisitos deve incluir requisitos que definam todas as funções e as restrições pretendidas pelo usuário do sistema.
- Verificações de realismo: Usando o conhecimento das tecnologias existentes, os requisitos devem ser verificados para assegurar que realmente podem ser implementados. Essas verificações devem considerar o orçamento e o cronograma para o desenvolvimento do sistema.

- Verificabilidade: Para reduzir o potencial de conflito entre o cliente e o contratante, os requisitos do sistema devem ser passíveis de verificação. Isso significa que você deve ser capaz de escrever um conjunto de testes que demonstrem que o sistema entregue atende a cada requisito especificado.

Existe uma série de técnicas de validação de requisitos que podem ser usadas individualmente ou em conjunto:

1. Revisões de requisitos. Os requisitos são analisados sistematicamente por uma equipe de revisores que verifica erros e inconsistências.
2. Prototipação. Nessa abordagem para validação, um modelo executável do sistema em questão é demonstrado para os usuários finais e clientes. Estes podem experimentar o modelo para verificar se ele atende a suas reais necessidades.

Brooks (2009, p. 192) afirma que é realmente impossível para os clientes, mesmo aqueles que trabalham como engenheiros de software, especificar completamente, precisamente e corretamente os requisitos exatos de um produto moderno de software, sem que tenham versões do produto que eles estão especificando. Portanto, um dos atuais esforços tecnológicos mais promissores, do problema de software é o desenvolvimento de abordagens e ferramentas para a prototipagem rápida de sistemas como parte da especificação iterativa de requisitos.

Para McConnell (2004 apud PINTO SILVEIRA, 2006) a prototipação é uma das principais técnicas para a validação, pois quando dispomos de uma Especificação de Requisitos Executável, a validação fica facilitada, pois aquela já pode ser diretamente utilizada como teste pelos próprios usuários. Com isto pode-se ter redução dos custos e dos prazos de desenvolvimento dos sistemas

3. Geração de casos de testes. Os requisitos devem ser testáveis. Se os testes forem concebidos como parte do processo de validação, isso frequentemente revela problemas de requisitos. Se é difícil ou impossível projetar um teste, isso normalmente significa que os requisitos serão difíceis de serem implementados e devem ser reconsiderados.

Você não deve subestimar os problemas envolvidos na validação de requisitos. Afinal, é difícil mostrar que um conjunto de requisitos atende de fato às necessidades de alguns usuários; os usuários precisam imaginar o sistema em operação e como esse sistema se encaixaria em seu trabalho. Até para profissionais qualificados de informática é difícil fazer esse tipo de análise abstrata, e é mais difícil ainda para os usuários do sistema. Como

resultado, durante o processo de validação dos requisitos você raramente encontrará todos os problemas de requisitos. Após o ajuste do documento de requisitos, é inevitável a necessidade de mudanças nos requisitos para corrigir omissões e equívocos.

Pressman (2006, p. 303) de uma forma mais simples, argumenta que a validação só se torna bem sucedida quando o software funciona de um modo que pode ser razoavelmente esperado pelo cliente.

Expectativas razoáveis são definidas nas Especificações dos Requisitos de Software, um documento que descreve todos os atributos do software visíveis ao usuário. A especificação contém uma seção chamada de Critérios de Validação. A informação contida nessa seção forma a base para abordagem do teste de validação

1.5.1. Critérios do Teste de Validação.

A validação do software é conseguida por intermédio de uma série de testes que demonstram conformidade com os requisitos. Tanto o plano de teste quanto o procedimento são projetados para garantir que todos os requisitos funcionais sejam satisfeitos, todas as características comportamentais sejam conseguidas, todos os requisitos de desempenho sejam alcançadas, e a documentação esteja correta, usabilidade e outros requisitos sejam satisfeitos (por exemplo, transportabilidade, compatibilidade, recuperação de erro e manutenibilidade).

Um dos grandes problemas referente a teste de software é o fato de a equipe considerar essa atividade como secundária. O responsável pelo teste acaba executando este de má vontade, de forma errônea e ineficiente. Ao longo do projeto não é dada muita importância aos testes, estipulando prazos reduzidos, e métodos pré-estabelecidos que não se aplicam a todos os casos. Devido a isso, é importante buscar uma metodologia de testes adequada, através de uma equipe treinada e consciente da importância dos testes para o sucesso do projeto (RIOS, 2006 apud TONIAZZO, 2007).

Depois que cada caso de teste de validação tenha sido conduzido, uma de duas possíveis condições se realiza.

2. A característica de função ou desempenho satisfaz à especificação e é aceita ou;
3. É descoberto um desvio de especificação e uma lista de deficiências é criada. Desvios ou erros descobertos neste estágio, em um projeto, raramente podem ser corrigidos antes da entrega programada. É frequentemente necessário negociar com o cliente para estabelecer um método de resolução de deficiências.

"Qualquer programador pode escrever códigos que
um computador consegue ler. Bons programadores
escrevem códigos que humanos conseguem ler!".
LOCAWEB

CAPITULO II

2. ESPECIFICAÇÃO POR EXEMPLO

Documentar processos é tão difícil de escrever e tão caro para manter, como qualquer outro tipo de documento de sistema. Mas, por que precisamos oficialmente de uma documentação? Muitas vezes, a única maneira de descobrir o que o sistema faz é olhar o código fonte da linguagem de programação ou utilizar a engenharia reversa[1] na funcionalidade do negócio. Mas mesmo quando o código está correto, a engenharia reversa é a uma tarefa impossível para os usuários de negócios, testadores e até mesmo para os desenvolvedores. (ADZIC, 2011, p. 31, tradução nossa).

Para Crispin e Gregory (2009, p. 257, tradução nossa) a pergunta é, por que queremos automatizar os testes e o que nos impede?

- Testes manuais levam muito tempo;
- Processos manuais são propensos a erros;
- Automatizar os testes liberta pessoas para fazer seu trabalho melhor;
- Testes fornecem documentação, apesar das pessoas não familiarizadas com o desenvolvimento ágil, frequentemente tem o equívoco de achar que não há documentação. O fato é que, projetos ágeis produz documentação utilizável que contêm testes executáveis, portanto, é sempre atualizado. E uma das vantagens

[1] Pressman (2006, p. 687) A engenharia reversa de software consiste na análise de um programa de computador, em um esforço para representá-lo em uma abstração mais alta do que o código fonte. A engenharia reversa é um processo de recuperação de projeto. As ferramentas de engenharia reversa extraem informação do projeto de dados arquitetural e procedimental, para um programa existente.

dos testes executáveis como parte de seu documento de requisitos é que é difícil argumentar com os seus resultados;

- A automatização pode ser um bom retorno de investimento, principalmente quando os testes que ilustram exemplos de comportamento desejado são automatizados, tornando "viva" a documentação de como o sistema realmente funciona.

Cohn (2004, p. 32, tradução nossa) também questiona. Por que mudar? Por que não apenas continuar escrevendo documentos de requisitos da forma tradicional? Para tirar essas dúvidas, Cohn responde dizendo que as histórias de usuário, oferecem inúmeras vantagens, entre elas:

- Histórias de usuário são compreendidas por ambos envolvidos;
- Histórias de usuário trabalham com desenvolvimento iterativo, e;
- Através das histórias de usuário o requisito pode ser adiado até que se tenha um entendimento do que realmente precisa ser feito.

Osterwalder e Pigneur (2010, p. 172) afirmam que contar histórias serve como uma poderosa ferramenta para fazer novos modelos de negócios mais tangíveis.

Uma observação citada por Cohn (2004, p. 188, tradução nossa) argumenta que histórias de usuário não são casos de uso, e a diferença mais óbvia é o seu escopo, apesar de ambas serem dimensionados para agregar valor ao negócio.

Casos de usos é uma descrição generalizada de um conjunto de iterações entre o sistema e um ou mais atores. Um caso de uso quase sempre abrange um âmbito muito maior do que uma história, focam muito cedo na implementação de software ao invés dos objetivos do negócio. E casos de usos são artefatos permanentes que continuam a existir enquanto o produto encontra-se em desenvolvimento ativo ou de manutenção. E finalmente, casos de usos são escritos como resultados de uma atividade de análise.

Histórias de usuários, por outro lado, são escritas como notas que podem ser usadas para iniciar conversar de análise e são mantidas em menor escopo porque são colocadas restrições sobre seu tamanho. E não sobrevive a iteração em que são adicionadas ao software.

Adzic argumenta que, o problema de qualquer outro tipo de documentação são os custos das manutenções, alterar partes do que está desatualizado não contribui significativamente para os custos. Muitas vezes, custos é o resultado do tempo gasto em encontrar o que precisa ser mudado (Adzic, 2011, p. 31, tradução nossa).

Ribeiro (2010) cita que testes bem escritos atuam como um tipo de requisitos executáveis que ajudam a manter o entendimento compartilhado da equipe de desenvolvimento, sobre como o sistema de software representa os problemas do mundo real.

Alencar (1999) complementa, dizendo que, quando dispomos de uma especificação de requisitos executável, a validação fica facilitada, pois aquela já pode ser diretamente utilizada como teste pelos próprios usuários.

Para Sommerville (2001, p. 63) [...] requisitos mudam tão rapidamente que um documento de requisito já está ultrapassado assim que termina de ser escrito. Portanto, o esforço é em grande parte desperdiçado. Sommerville acredita que uma boa abordagem para os sistemas de negócio em que os requisitos são estáveis é coletar os requisitos de usuário de forma incremental e escrevê-los como estórias de usuário, onde então, é priorizado os requisitos para a implementação no próximo incremento do sistema.

E é a partir dessas histórias que a Especificação por Exemplo se baseia, ajudando as equipes a estabelecer um processo de especificação colaborativa, reduzindo problemas no meio da iteração[1], pois ela se encaixa dentro de pequenas iterações não necessitando de longos meses escrevendo uma documentação (Adzic, 2001, p. 13, tradução nossa).

Adzic complementa, dizendo que, uma solução ideal poderia ser uma documentação de sistema que fosse fácil e barata de manter, de tal modo que ele possa ser mantido de acordo com a funcionalidade do sistema, mesmo se o código da linguagem de programação fosse alterado frequentemente.

2.1. Terminologia e Definição

O termo Especificação por Exemplos (Specification by Example) surgiu segundo seu criador Gojko Adzic (2011) com o intuito de unificar em um só termo as seguintes técnicas, metodologias e práticas ágeis:

[1] Uma iteração abrange as atividades de desenvolvimento que conduzem à liberação de um produto com uma versão do produto estável e executável. Portanto, uma iteração de desenvolvimento é de certa forma uma passagem completa por todas as disciplinas: Requisitos, Análise e Design, Implementação e Teste. É como um pequeno projeto.

- Agile acceptance testing - Teste de Aceitação Ágil.
- Acceptance Test-Driven Development - Desenvolvimento Orientado a Testes de Aceitação.
- Example-Driven Development - Desenvolvimento Orientado por Exemplos.
- Story testing - Teste de História.
- Behavior-Driven Development - Desenvolvimento Orientado a Comportamento.
- Specification by Example - Especificação por Exemplo.
- Executable requirements - Requisitos Executáveis

Segundo Adzic o fato de as mesmas práticas possuírem tantos nomes diferentes reflete o grande número de inovações nesse campo no cenário atual da indústria de software. E sobre essas diferentes nomenclaturas, Adzic afirma:

"Se queremos os usuários finais (*Stakeholders*) mais envolvidos, o que é o principal objetivo dessas práticas, precisamos usar os nomes corretos para as coisas corretas e parar de confundir as pessoas".

Adzic decidiu por tanto simplificar essa terminologia na comunidade de software para que as pessoas possam usufruir de seus benefícios sem entrarem em discursões improdutivas sobre terminologias. Especificação por Exemplos é o resultado de um conjunto de estudos de casos realizados por Adzic com mais de 50 projetos de desenvolvimento de software do mundo real. Projetos que variam em tecnologia (de websites a softwares para desktop) e em metodologias (Extreme Programing, Scrum, Kanban e outras metodologias normalmente agrupadas sobre os termos agile e lean). Especificação por Exemplos pode ser definida como um suplemento de práticas que podem ser incorporadas a qualquer metodologia de desenvolvimento atual.

Outra definição é citada por Adzic em seu artigo (Lições aprendidas a partir dos 50 projetos de sucesso, 2011) onde o mesmo afirma que a Especificação por Exemplos é uma abordagem colaborativa para a definição de testes funcionais de software, aplicada por líderes de equipe para reduzir significativamente o tempo de comercialização, reduzindo o desperdício causado por requisitos mal entendidos, alinhando as atividades de vários papéis em iterações curtas e reduzindo ainda mais o retrabalho. (Adzic, 2011, tradução nossa)

Para Martin, (2012, p. 105) escrever testes é simplesmente o trabalho de especificar o sistema. Especificar nesse nível de detalhe é a única forma que os programadores sabem que "acabou" significa. É a única forma na qual os *Stakeholders* podem garantir que o sistema pelo qual estão pagando realmente fará o que desejam.

Adzic cita que na última década, a comunidade de desenvolvimento de software tem se esforçado para construir o software do jeito "certo", com foco nas técnicas e ideias para garantir a alta qualidade dos resultados. Mas construir o produto certo e construir certo o produto são duas coisas diferentes e que precisam andar juntas, a fim de obter o sucesso. (Adzic, 2001, p. 4, tradução nossa).

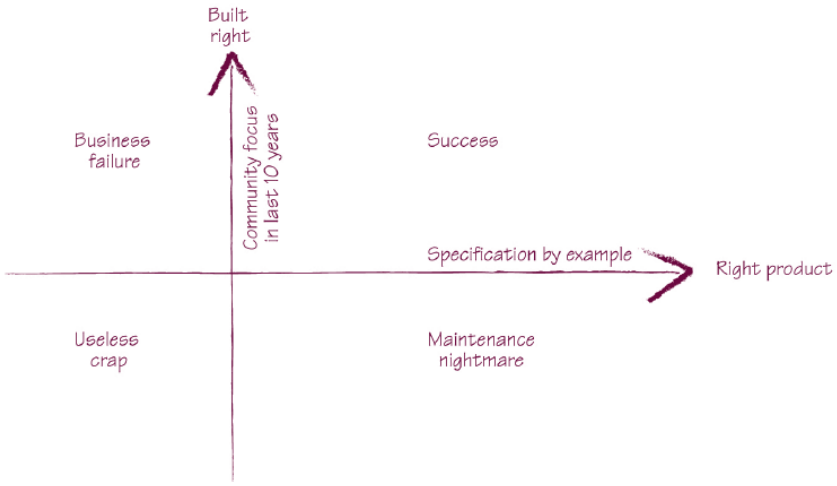

Figure 3. Especificação por Exemplo ajuda as equipes a construírem o produto de software correto, complementando as técnicas que asseguram que o produto será construído corretamente.

Fonte: Adzic, 2011, p. 4, tradução nossa.

E para construir o produto efetivamente certo, práticas de desenvolvimento de software tem que fornecer as seguintes características a seguir, conforme apontado por Adzic (2011, p.4, tradução nossa):

- Garantia que todas as partes interessadas e os membros das equipes entendam o que precisa ser entregue, e da mesma forma;
- Especificações precisas são entregues as equipes evitando o desperdício do retrabalho causado pela ambiguidade;
- Facilidade para alterar a documentação, tanto nas características do software quanto na estrutura da equipe.

2.2. Necessidades

Construir o produto certo e construir certo o produto são dois objetivos distintos. Porém os dois são necessários para se obter sucesso. Segundo Adzic tradicionalmente, construir o produto certo requeria enormes especificações funcionais, muitas documentações e longos ciclos de teste, atualmente em um mundo de releases[1] semanais isso não funciona. Ainda segundo Adzic é necessário uma solução para:

- Evitar o acúmulo de especificações inúteis;
- Evitar gastar tempo em detalhes que mudarão antes do primeiro fragmento de trabalho ser desenvolvido;
- Ter documentos legíveis que explicam o que o sistema faz para que se possa modificá-lo facilmente;
- Checar de maneira eficiente que o sistema faz o que as especificações dizem;
- Manter a documentação relevante e confiável com o menor custo de manutenção, e;
- Colocar tudo isso em processos baseados em iterações, de forma que a informação do trabalho a ser feito seja produzida sob demanda.

2.3. Padrões de Processo

A Especificação por Exemplos unificou padrões e processos que as equipes do mundo real utilizaram para atingir esses objetivos.

[1] Um release pode ser interno ou externo. Um release interno é usado apenas pela organização de desenvolvimento, como parte de um marco, ou para fazer uma demonstração para usuários ou clientes. Um release externo é liberado para os usuários finais. Um release não é necessariamente um produto completo, mas pode ser apenas uma etapa ao longo do caminho, com sua utilidade avaliada apenas do ponto de vista da engenharia. Uma das funções dos releases é forçar a equipe de desenvolvimento a fazer fechamentos em intervalos regulares. Releases regulares permitem que você fragmente os problemas de integração e teste, e os distribua pelo ciclo de desenvolvimento.

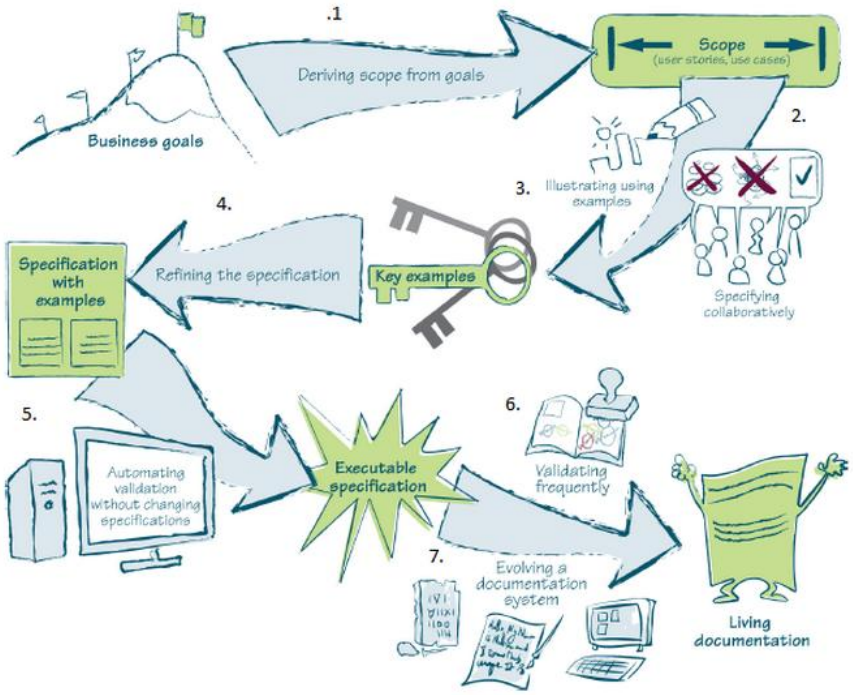

Figura 4. Os principais padrões de processo da Specification by Example.

Fonte: Adzic, 2011, p. 18, tradução nossa.

A Especificação por Exemplos consiste em vários padrões de processos que são elementos de amplo desenvolvimento no ciclo de vida de um software. Os padrões estabelecidos pela Especificação por Exemplos são assim denominados no sentido em que eles são elementos recorrentes usados por diferentes equipes; e são resultados de uma série de experimentos empíricos com projetos e equipes reais e não de um modelo teórico na elaboração de processos.

A seguir serão apresentadas desafios e ideias para implementar cada um desses padrões em diferentes contextos.

1. Deriving scope from goals (Derivando o escopo dos objetivos): Nessa etapa a equipe que criará o software deve derivar os escopos de desenvolvimento a partir dos objetivos estabelecidos pelos *Stakeholders*. Segundo Adzic é um erro esperar que os usuários envolvidos digam qual o escopo de desenvolvimento pois os mesmos não são designers de software. O mais apropriado nessa etapa é elencar os objetivos que os *Stakeholders* querem atingir com o software em

questão. Em outras palavras, qual ou quais os problemas que a "solução" deve ser capaz de resolver. Começa-se com um objetivo de negócios a equipe de desenvolvimento colabora definindo o escopo que vai atingir este objetivo.

2. Specifying collaboratively (Especificando colaborativamente): A equipe nessa etapa deverá juntamente com os *Stakeholders* especificar colaborativamente o comportamento esperado para cada funcionalidade do escopo. Desenvolvedores, testadores e analista de negócios devem participar dessa etapa, pois cada um tem um contexto único e habilidades que podem ajudar a resolver questões que de outra forma poderiam passar despercebidas gerando retrabalho no futuro.

3. Illustrating using examples (Ilustrando utilizando exemplos): Ao invés de esperar que as especificações sejam expressas precisamente em uma linguagem de programação durante a implementação, deve-se ilustrar o comportamento esperado do sistema utilizando exemplos antes do desenvolvimento ser iniciado. A equipe de desenvolvimento trabalha com os *Stakeholders* para identificar exemplos-chave que descrevem a funcionalidade esperada. Desenvolvedores e testadores dão exemplos adicionais que revelam áreas antes ignoradas da funcionalidade evitando assim gaps funcionais e faz com que todos tenham um conhecimento distribuído sobre aquilo que precisam entregar.

4. Refining the specification (Refinando a especificação): Nessa etapa a equipe de desenvolvimento deve remover informações irrelevantes dos exemplos tais como detalhes que tornam os exemplos extensos sem adicionar informações úteis. Segundo Adzic deve-se identificar o que o software deve fazer e não detalhar o como fazer. Os resultados dessa etapa podem ser considerados critérios de aceitação, que ao mesmo tempo são especificações para o desenvolvimento e futuramente quando automatizados farão parte dos testes de regressão.

5. Automating validation without changing specifications (Automatizando as validações sem modificar as especificações): Com os exemplos especificados e refinados a equipe pode usá-los como meio de validar o produto. O sistema será validado diversas vezes durante o desenvolvimento para assegurar que o mesmo

atinge os objetivos. Rodar essas validações manualmente introduziria um delay[1] desnecessário e o feedback[2] seria lento. A solução para isto é automatizar as validações, porém essa automação deve também ser acessível aos *Stakeholders* e não deve modificar as especificações originais. A equipe de desenvolvimento literalmente deve escrever a automação com as mesmas palavras utilizadas para criarem os exemplos, pois scripts e código fonte não têm a capacidade de contar adequadamente o contexto de negócio em que estão inseridos e a documentação puramente textual fica desatualizada rapidamente. Portanto, nessa etapa a validação automática do sistema, suas especificações por exemplos e seus critérios de aceitação tornam-se um só. O resultado dessa etapa gera especificações executáveis.

6. Validating Frequently (Validando frequentemente): Adzic afirma que a documentação tradicional de software fica desatualizada antes mesmo de o sistema ser lançado. A validação frequente é um produto das especificações executáveis ela sincroniza o que foi especificado com o que o sistema realmente está fazendo.

7. Evolving a documentation system (Evolução de um sistema de documentação): Nessa etapa a equipe de desenvolvimento mantém a documentação viva (especificações executáveis - testes) atualizada com as modificações requisitadas pelos *Stakeholders* para incrementar o software. As especificações e testes estando em um mesmo lugar facilitam essa mudança, pois concentram os esforços em apenas uma fonte de informações. Desenvolvedores utilizam essa documentação como especificações, testadores utilizam como testes e os analistas de negócio utilizam para avaliar o impacto que uma mudança ou especificação futura pode ter no sistema.

(ADZIC, 2011, p.18-24, tradução nossa)

---

[1] Delay: Tempo de espera de uma ação

[2] Feedback: Quando é dado um parecer sobre uma pessoa ou grupo de pessoas na realização de um trabalho com o intuito de avaliar o seu desempenho. É uma ação que revela os pontos positivos e negativos do trabalho executado tendo em vista a melhoria do mesmo.

2.4. Desafios

Alguns dos desafios apresentados por Adzic (2011, p. 37, tradução nossa) citam que antes de começar alterando o processo, é imprescindível começar a mudar a cultura da equipe, iniciando com a abolição do termo ágil, principalmente quando trabalhar em um ambiente que é resistente a mudanças, e ainda:

- Verificar se já existe um processo de mudanças acontecendo, e se não há, utilizar as ideias importantes da Especificação por Exemplo;
- Apresentar a automatização das especificações executáveis para as equipes que possuam testes automatizados separados do desenvolvimento;
- Concentrar-se sempre em primeiro momento em melhorar a qualidade do requisito;
- Introduzir uma ferramenta para as especificações executáveis: quando os próprios analistas de testes automatizam os testes; entretanto, não gaste o seu tempo concentrando-se demais em uma ferramenta ao invés de focar no alto nível de colaboração e mudanças de processos que as especificações executáveis nos fornecem.

Crispin e Gregory (2009, p. 265, tradução nossa) listam as barreiras para o insucesso das especificações executáveis.

- Atitude dos programadores: Programadores que estão acostumados a trabalhar em um ambiente tradicional, onde a equipe de QA (Quality Assurance) faz todo o teste.
- Curva de aprendizagem: É difícil aprender automação de teste, especialmente fazendo isso de uma maneira que produza um bom retorno sobre os recursos investidos nele;
- Investimentos iniciais: Mesmo com toda a equipe trabalhando no problema, a automatização requer um grande investimento, que pode não compensar de imediato;
- Sistemas legados: É difícil de automatizar, pois não foi projetado para isso;
- Medo: Automatizar teste é assustador para quem nunca o fez. Programadores podem ser bons pra escreverem códigos, mas eles não possuem muita experiência para escrever testes automatizados.

- Velhos hábitos: Quando a equipe encontra-se em pânico, devido aos atrasos nas entregas, eles fatalmente voltam aos velhos hábitos, mesmo que esses hábitos não produzam bons resultados.

Crispin e Gregory argumentam que cada equipe, cada projeto e cada organização tem uma situação única com os desafios de automação. Pois, cada um tem sua própria cultura, história, recursos, produtos e experiências.

2.5. Benefícios

De acordo com Rob Park (Líder Ágil, da LeanDog, apud Adzic, 2011, pg. 13, tradução nossa) o aspecto mais recompensador da Especificação por Exemplo é receber o sentido da história e saber exatamente até que extensão você chegará ao começar a construir. [...] permite que as equipes definam a funcionalidade esperada em um objetivo claro e de forma mensurável. Também acelera o feedback, melhorando o fluxo de desenvolvimento e evitando interrupções no trabalho planejado.

As descrições da Especificação por Exemplos permite que as equipes se envolvam melhor com os usuários de negócios, garantindo assim um entendimento comum dos resultados.

Adzic (2011, p. 14, tradução nossa) relata através das suas experiências que muitas equipes tem reduzido significativamente ou eliminado completamente o retrabalho que ocorreu quando um resultado foi mal entendido ou a expectativa do cliente foi negligenciado e ainda complementa que muitas equipes mudaram gradativamente o seu processo de trabalho de acordo com o uso da Especificação por Exemplo, muitas pessoas ficam confusas quando os papéis começam a mudar. Testadores tiveram que ficar mais envolvidos na análise, desenvolvedores tiveram que se envolver mais nos testes e analistas tiveram que mudar a forma como obtinha e transmita os requisitos. E os usuários de negócio tiveram que assumir um papel muito mais ativo na preparação das especificações.

(Adzic, 2011, p. 55, tradução nossa) A Especificação por Exemplo também fornece artefatos em torno dos requisitos como forma de uma documentação viva que pode ser usada para a rastreabilidade, mantenho as especificações executáveis em um sistema de controle de versões, pois ela está diretamente ligada ao código da linguagem de programação, (através da camada de automatização) o que significa que será relativamente simples provar a rastreabilidade do código.

Para Shore (2010), o verdadeiro objetivo da técnica de Especificação por Exemplo é a melhoria da comunicação, aprimorando a colaboração com o cliente.

2.6. Fatores Negativos

Segundo o artigo publicado por Martin Fowler (2002, republicado, 2011, tradução nossa), o mesmo declara que Especificação por Exemplo não é a maneira como a maioria de nós foram educados para pensar sobre especificações. Especificações devem ser gerais, para todos os casos. Exemplos apenas destacam alguns pontos, temos que deduzir as generalizações por si só. Isto significa que a Especificação por Exemplo, não pode ser a técnica de requisitos que você usa, mas isso não significa que ele não possa assumir um papel de liderança. Pois, ela sozinha, não é o suficiente, você precisa fazer mais para garantir que tudo está devidamente se comunicando.

A especificação formal constantemente tinham problemas ao verificar se um projeto satisfazia uma especificação, especialmente porque os seres humanos são propensos a erros. Para Especificação Por exemplo, isto é fácil. Outro caso da especificação por Exemplo é ser menos valiosa na teoria, e mais valiosa na prática.

Uma das coisas mais perigosas sobre uma especificação de requisitos tradicional é quando as pessoas pensam que uma vez já escrito, eles já estão se comunicando.

E finaliza dizendo que, a Especificação por Exemplo só funciona no contexto de uma relação de trabalho onde ambos os lados estão colaborando e não lutando, além do mais, ela é uma ferramenta poderosa, talvez a minha ferramenta mais utilizada, mas nunca minha única ferramenta.

Cohn aborda alguns inconvenientes relacionados às histórias de usuários. Cohn (2004, p. 197, tradução nossa) cita que um projeto grande com muitas histórias, pode ser difícil de entender as relações entre as mesmas e ainda segue dizendo que é necessário aumentar a documentação caso haja a obrigatoriedade de rastrear os requisitos no seu processo de desenvolvimento.

Há uma ressalva citada por Ribeiro (2010), dizendo que, o fato de se ter um grande número de testes passando com sucesso, pode passar de uma falsa sensação de segurança, resultando na implementação de menos atividades de QA, como testes de integração e testes de conformidade.

A automatização não é necessária em tudo, pois, o custo de manutenções ao longo prazo nas especificações executáveis é um dos maiores problemas que as equipes enfrentam hoje ao implementar a Especificação por Exemplos. (Adzic, 2011, p. 137, tradução nossa).

2.7. Considerações Importantes

A Especificação por Exemplo requer uma participação ativa das partes interessadas, e a total entrega dos membros da equipe, incluindo desenvolvedores, testadores e analistas. A Especificação por Exemplo fornece apenas uma documentação suficiente no momento certo, ajudando a construir o produto certo com iterações curtas.

No início, muitas equipes não entendiam que a documentação reflete o modelo de domínio que o sistema descreve, no entanto, esse documento deve possuir exemplos precisos que nos ajudam a evitar a ambiguidade, definindo claramente o contexto e a forma como o sistema deve funcionar em um determinado caso. Devem ser completos, realistas, fáceis para entender, e bem organizadas, afim de que as especificações sejam de fácil acesso, e não deve haver respostas sim/não em seus exemplos, enquanto o conceito não estiver bem definido, pode dar as pessoas uma falsa sensação de que elas compartilham do mesmo entendimento que você, quando não o fazem.

Use um formato padrão nas linguagens de especificações a fim de fazer os testes mais fáceis de entender, como: Dado - Quando - Então.

- Dado: uma condição
- Quando: uma ação acontece
- Então: as seguintes pós-condições devem ser satisfeitas.

A junção de todas essas informações são condições indispensáveis para tornar um documento de requisito executável construindo assim, a confiança entre as partes interessadas e os membros da equipe, reduzindo significativamente o feedback das iterações no desenvolvimento de software, levando a menos retrabalho, alta qualidade no produto, tempo de resposta mais rápido para as mudanças de software e melhor alinhamento das atividades de vários papéis envolvidos no desenvolvimento, tais como testadores, analistas e desenvolvedores. (ADZIC, 2001, tradução nossa)

Conforme Glass (apud Brooks 2009, p. 216) desenvolvimento de software é algo conceitualmente difícil. Que soluções mágicas não se encontram nas esquinas. Que é tempo para examinar as melhorias evolutivas em vez de esperar ou desejar melhorias

revolucionárias. E que agora, talvez, possamos prosseguir com os avanços incrementais possíveis para a produtividade no desenvolvimento de software em vez de ficar esperando avanços revolucionários que, provavelmente, não aparecerão.

Entretanto, para Papo (2011) não importa se chamarmos de Desenvolvimento Orientado a Comportamento ou Desenvolvimento Orientado a Teste de Aceitação ou Especificação por Exemplos. O que queremos é o mesmo: um entendimento compartilhado do que deve ser construído, para criarmos um produto certo.

CAPITULO III

3. FERRAMENTA CUCUMBER

Quando construímos software para os *Stakeholders*, não é segredo que há uma dificuldade para descobrir exatamente o que eles querem que a gente construa. (WYNNE e HELLESOY, 2012, pg. 25, tradução nossa).

"A parte mais difícil de construir um software é decidir precisamente o que construir". (WYNNE e HELLESOY apud BROOKS, 2012, p. 25, tradução nossa).

Baseado na mesma ideia Brooks apud Einstein (2009, p. 67) onde ele diz que: "A formulação de um problema é muitas vezes mais importante do que a sua solução".

Todos nós já trabalhamos em projetos onde, por causa do mal entendimento, códigos que tínhamos trabalho arduamente durante vários dias, haviam sido jogados fora.

Para evitar o desperdício de tempo é essencial que haja uma melhoria na comunicação entre os desenvolvedores e os *Stakeholders*.

Uma técnica que facilita essa comunicação é o uso de exemplos concretos para ilustrar o que queremos que o software faça. (WYNNE e HELLESOY, 2012, pg. 25, tradução nossa)

Cucumber é uma ferramenta de automação de teste funcional[1] para as equipes ágeis.

---

[1] Teste funcional: são aqueles que procuram testar as funcionalidades de sua aplicação, verificando a integração entre as diversas partes que a compõe.

Você pode utilizá-lo para automatizar sua validação funcional de forma legível e compreensível para os usuários de negócios, desenvolvedores e testadores. (FLORINIER e ADZIC, 2010, tradução nossa).

Isso ajuda as equipes a criar especificações executáveis, critérios de aceitação e verificações funcionais para futuras mudanças que são os objetivos para o desenvolvimento.

Cucumber permite que as equipes criem uma documentação viva, tornando uma única fonte de informação o local onde as funcionalidades dos sistemas estarão sempre atualizadas. (WYNNE e HELLESOY, 2012, p. 7, tradução nossa). Ter um único lugar para obter informações, economiza muito tempo que às vezes é desperdiçado tentando manter os documentos de requisitos, testes e códigos todos em sincronia.

3.1. Por que usar o Cucumber?

Florinier e Adzic (2010, p.2, tradução nossa) citaram diversas razões para se utilizar a ferramenta Cucumber:

- Relativamente fácil de configurar;
- Suporta múltiplos formatos de relatórios, incluindo HTML e PDF;
- Foi traduzido para mais de quarenta idiomas, tornando ainda mais fácil o processo e a integração com o cliente e facilitando a sua utilização para as equipes que estão fora do território da língua inglesa, podendo assim criar seus testes na sua língua nativa;
- Suporta diferentes formatos ao descrever as especificações executáveis, incluindo histórias em formato de lista, prosa e dados de tabela;
- Embora o Cucumber seja uma ferramenta escrita em Ruby[1], pessoas que trabalham com outras plataformas não precisa aprender Ruby para utilizar.

Além das razões citadas, o Cucumber serve de apoio ao BDD (Behavior Driven Development – Desenvolvimento Orientado a Comportamento) que por definição, possui três princípios básicos:

[1] Ruby: Uma linguagem dinâmica, open source com foco na simplicidade e na produtividade. Tem uma sintaxe elegante de leitura natural e de fácil escrita.

- O suficiente é suficiente: Não devemos automatizar tudo, mas sim tudo o que descreve o comportamento esperado do produto pelo cliente. O suficiente para desenvolver a solução. Mais do que isso é desperdício de esforço.
- Entregar valor para os *Stakeholders*: Entregue somente o que tem valor para o cliente, nada mais. Se o que estiver fazendo não agrega valor para o cliente ou não potencializar o valor entregue, pare de fazer isso.
- Tudo é comportamento: Tudo que um software faz pode ser descrito como comportamento e explicado para qualquer pessoa que tenha o domínio do negócio. Não importa o nível de teste, o tipo de funcionalidade, sempre será descrito como comportamento. (RIBEIRO, C., 2012).

3.2. Como o Cucumber Funciona?

O Cucumber é uma ferramenta de linha de comando.

Seu modo de funcionamento e suas camadas operam basicamente da seguinte forma: (WYNNE e HELLESOY, 2012, pg. 7, tradução nossa).

3.3. Camada de Negócios

O Cucumber lê as especificações escritas em uma linguagem simples a partir de um arquivo de texto, chamados features (funcionalidade); que examina esse tipo de arquivo a procura de cenários (scenarios), cada cenário é uma lista de passos (steps) que o Cucumber irá tentar executar, cada caso de teste do Cucumber é chamado de cenário, cada cenário é agrupado dentro de funcionalidade e cada cenário contêm vários passos.

Cada passo é escrito em linguagem natural de domínio da aplicação. Para que sejam facilmente lidos e interpretados por pessoas sem a necessidade de qualquer conhecimento técnico.

Para que o Cucumber possa entender a funcionalidade desses arquivos, ele deve seguir algumas regras básicas de sintaxe, chamada Gherkin.

3.4. Camada de Tecnologia

Para realizar quaisquer ações os passos (steps) precisam estar implementados em um arquivo separado chamado de definição de passos (step-definition).

A implementação dos passos (steps) dentro uma definição de passos (step-definition) é realizada através do código de suporte (support code), que pode ser escrito em uma variada gama de linguagens.

O código de suporte (support code) é o responsável por se encarregar das tarefas de automação, normalmente isso envolve uma biblioteca de automação (automation library) que então entrará em contato direto com o sistema alvo do teste.

Essa hierarquia de camadas pode ser melhor observada na figura abaixo.

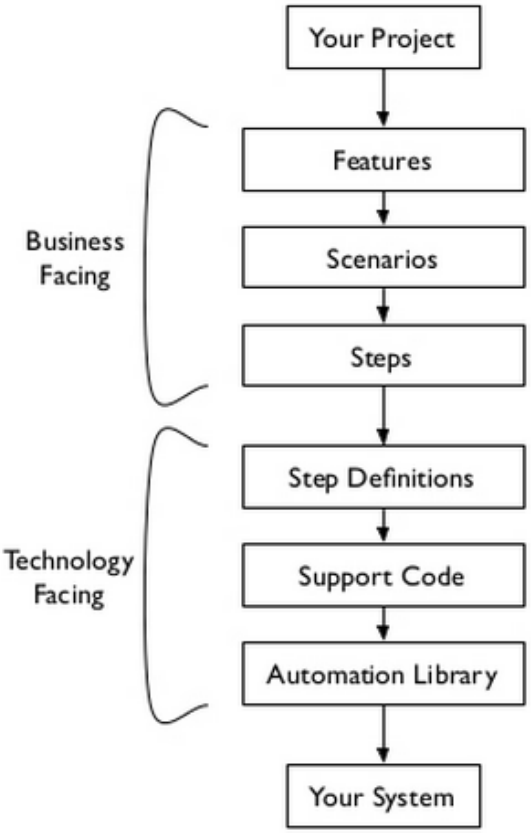

Figura 5. Pilha de funcionamento do Cucumber.

Fonte: Wynne e Hellesoy, 2012, pg. 08, tradução nossa.

3.5. Gherkin

Gherkin é uma linguagem utilizada para escrever as funcionalidades (features) do Cucumber.

A linguagem Gherkin nasceu da necessidade de expressar de forma clara o comportamento esperado das funcionalidades que os *Stakeholders* esperam do sistema utilizando exemplos.

"O Gherkin fornece uma estrutura leve para documentar os exemplos de comportamento que os *Stakeholders* querem". (WYNNE e HELLESOY, 2012, pg. 96, tradução nossa).

O melhor cenário para o Gherkin são criados quando três amigos andam juntos, em três diferentes perspectivas:

- O primeiro é o testador, que pensa em como quebrar as coisas.
- O segundo é o programador, que pensa em como fazer as coisas.
- O terceiro é o dono do produto (Product Owner[1]), que se preocupa com o escopo.

Quando o testador pensa nos cenários que atingiram casos extremos, o dono do produto, não pensa sobre, ele pode dizer a equipe para que deixe esse cenário, pois esta fora de escopo.

Quando o programador explica que a implementação de um cenário em particular será complicada, o dono do produto tem a autoridade para ajudar a decidir sobre as alternativas ou simplesmente tirá-lo. (WYNNE e HELLESOY, 2012, pg. 96, tradução nossa).

Mesmo o Gherkin podendo ser similar a uma linguagem de programação, o seu principal objetivo é ser legível para os humanos, isto significa que, é possível escrever testes automatizados que podem ser lidos como documentação, tais como no exemplo a seguir:

Funcionalidade: Leitor de tipos de Triângulo

Para conhecer o tipo de um triângulo

Como um aluno da matemática

Eu quero informar os tamanhos do lado de um triângulo e saber qual o tipo do triângulo

[1] Product Owner: representa os interesses de todos os envolvidos (*Stakeholders*), define as funcionalidades do produto e prioriza os requisitos.

NARRATIVA

Um triângulo com todos os lados iguais é chamado Equilátero.

Um triângulo com dois lados iguais é chamado Isósceles.

Um triângulo com todos os lados diferentes é chamado Escaleno.

FORA DE ESCOPO

- Validar triângulos inválidos.
- Exibir o triângulo graficamente.
- Validação de entrada de dados do usuário.

3.5.1. Exemplo de cenário escrito com Gherkin.

O Gherkin também foi traduzido para mais de quarenta idiomas bastando para isso colocar # language: pt, na primeira linha de um arquivo da funcionalidade (feature), no mais é continuar utilizando as palavras-chave (em negrito como no exemplo abaixo) equivalentes no idioma desejado.

#language: pt
Funcionalidade: Leitor de tipos de Triângulo

Para conhecer o tipo de um triângulo
Como um aluno da matemática
Eu quero informar os tamanhos do lado de um triângulo e saber qual o tipo do triângulo

NARRATIVA

Um triângulo com todos os lados iguais é chamado Equilátero
Um triângulo com dois lados iguais é chamado Isósceles
Um triângulo com todos os lados diferentes é chamado Escaleno

FORA DE ESCOPO

- Validar triângulos inválidos
- Exibir o triângulo graficamente
- Validação de entrada de dados do usuário

Cenario: Consultando um triângulo escaleno
Dado que estou na página de consulta de triângulos
Quando eu informo os lados de um triangulo:
| lado_a | lado_b | lado_c |

```
        | 3     | 4     | 5     |
        Entao o sistema informa que o triângulo é "Escaleno"

Cenario: Consultando um triângulo equilátero
        Dado que estou na página de consulta de triângulos
        Quando eu informo os lados de um triangulo:
        | lado_a | lado_b | lado_c |
        | 5     | 5     | 5     |
        Entao o sistema informa que o triângulo é "Equilátero"

Cenario: Consultando um triângulo isósceles
        Dado que estou na página de consulta de triângulos
        Quando eu informo os lados de um triangulo:
        | lado_a | lado_b | lado_c |
        | 5     | 4     | 5     |
        Entao o sistema informa que o triângulo é "Isósceles"
```

O teste mencionado acima é autoexplicativo, ou seja, qualquer pessoa que saiba ler português sabe o que será testado. Além disso, o teste está junto com o seu requisito, ou seja, o requisito e o teste nasceram juntos e caso um seja alterado o outro será também. Na verdade, neste tipo de abordagem, o teste é o requisito. (RIBEIRO, C., 2012).

3.5.2. Formato e Sintaxe.

Os arquivos Gherkin usam a extensão .feature e são salvos em arquivos texto, isto significa que eles podem ser lidos e editados por ferramentas simples. A estrutura de um arquivo Gherkin é estabelecida pelas seguintes palavras-chave:

- Feature
- Scenario
- Given
- And
- When
- Then
- Background
- And
- But
- Scenario Outline
- Examples

Estas mesmas palavras podem ser traduzidas para o português como:

- Funcionalidade
- Cenário
- Dado
- E
- Quando
- Então
- Contexto
- E
- Mas
- Esquema do Cenário
- Exemplos

Para outro idioma basta utilizar as palavras equivalentes na linguagem desejada.

3.5.3. Feature (Funcionalidade)

Cada arquivo Gherkin começa com a palavra-chave Funcionalidade (Feature). Essa palavra-chave não afeta o comportamento do Cucumber, ela apenas dá um lugar conveniente para escrever alguma documentação resumida a respeito do comportamento da funcionalidade que será descrita através dos testes.

Segue um exemplo da utilização dessa palavra-chave:

Funcionalidade: Esse é o título da funcionalidade. Essa é a descrição que pode ocupar diversas linhas. É possível até mesmo colocar linhas em branco.

Tudo que esteja escrito até a próxima palavra-chave do Gherkin é incluído na descrição.

Dentro de uma estrutura válida do Gherkin a palavra-chave Funcionalidade (Feature) deve obrigatoriamente ser seguida de uma das seguintes palavras-chave:

- Scenario (Cenário)
- Background (Contexto)
- Scenario Outline (Esquema do Cenário)

3.5.4. Scenario (Cenário)

Para representar o comportamento necessário, cada funcionalidade (feature) contém vários cenários (scenarios). Cada cenário (scenario) é um único e concreto exemplo de como a funcionalidade em questão deve se comportar em um caso específico. Todos os cenários (scenarios) de uma funcionalidade (feature), juntos definem o comportamento dessa funcionalidade (feature).

Todos os cenários (scenarios) seguem o seguinte padrão:

- Colocam o sistema em um estado específico;
- Realizam algum tipo de ação;
- Examinam o estado final.

3.5.5. Given, When, Then (Dado, Quando, Então)

No padrão Gherkin os estados, ações e asserções que ocorrem em um cenário (scenario) são representados pelas palavras-chave Dado (Given), Quando (When) e Então (Them), da seguinte forma:

Cenários (Scenarios): <descrição do teste>

Dado (Given): <prepara o contexto>

Quando (When): <executa uma ação ou evento>

Então (Then): <verifica o estado final>

O Dado (Given) nunca executa ação/evento e nem faz asserção explicita, ele prepara o ambiente. Não faz parte dele passos (steps) como:

- Dado que eu estou entrando na página (Evento).
- Dado que eu clico no botão “Ok” (Evento).
- Dado que a mensagem de erro é igual a “Mensagem” (Resultado).

O Dado (Given) é focado em estado. Exemplos de passos (steps) seriam:

- Dado que eu tenho um usuário logado.
- Dado que existe um produto chamado “Refrigerante”.
- Dado a home page do google.com.

Já o Quando (When) indica uma ação do usuário ou um evento sistêmico. São maus exemplos:

- Quando o usuário "João" está logado (Estado).
- Quando o processamento dos dados terminar (No sentido de conferir, não de evento).
- Quando eu estou na home page do google.com.

São bons exemplos do uso do Quando (When):

- Quando eu preencho o formulário de cadastro do cliente "Joao".
- Quando o sistema desliga o servidor.
- Quando eu envio um e-mail através da ajuda ao cliente.

O Então (Then) é o teste em si. Esse passo verifica se com o nosso estado inicial e as nossas ações nós chegamos ao nosso resultado esperado. São maus exemplos do uso do então (then):

- Então eu devo clicar no botão "Ok"
- Então eu envio um e-mail através da ajuda ao cliente
- Então eu Busco pela frase "Voto Como Vamos"

São bons exemplos do uso do Então (Then):

- Então eu devo ver o usuário cadastrado
- Então o servidor deve estar desligado
- Então eu devo receber um e-mail da central de ajuda ao cliente

Dessa forma usa-se o Dado (Given) para preparar o contexto de onde o cenário (scenario) vai acontecer. Usa-se o Quando (When) para interagir com o sistema de alguma forma e o Então (Then) para checar se o resultado dessa ação foi o esperado.

3.5.6. And, But (E, Mas)

Cada linha de um cenário (scenarios) é conhecida como passo (step). É possível adicionar mais passos (steps) para cada Dado (Given), Quando (When) ou Então (Them) utilizando-se E (And) e Mas (But). Como no exemplo a seguir:

Cenário: Retirando dinheiro de uma conta com crédito

Dado uma conta contendo R$ 100,00

Quando eu requisito $ 49,00

Então o caixa deve liberar o valor de $ 49,00

E o saldo restante deve ser R$ 51,00

Essas palavras-chave não modificam o comportamento do Cucumber e existem simplesmente para promover a legibilidade dos cenários (scenarios).

3.6. Quando não utilizar o Cucumber

Wynne e Hellesoy (2012, p. 86, tradução nossa) identificaram alguns problemas causados quando as equipes deixam de colaborar.

- Quando os *Stakeholders* não leem as funcionalidades (features): Se eles pensam que estão ocupados demais para ajudar você a entender exatamente o que eles querem, então você tem um enorme problema na equipe, que o Cucumber não pode ajudá-lo a resolver. Por outro lado, quando as equipes começam a ficar desinteressadas, eles estão desperdiçando a oportunidade de construir uma relação de colaboração.

  Quando as funcionalidades são escritas somente pelos analistas de testes e os desenvolvedores, eles inevitavelmente utilizam técnicas e termos que fazem os *Stakeholders* se sentirem sem importância enquanto leem.

Isto se torna um círculo vicioso, com a perda de interesse dos *Stakeholders*, eles gastam menos tempo ajudando a escrever as funcionalidades em uma linguagem que faça sentido para eles. E antes que eles percebam, a funcionalidade se tornou nada mais que uma ferramenta de teste.

- Se o projeto requer uma quantidade superior de qualidade, ou seja, que não seja necessário que apenas o comportamento do sistema seja testado continuamente, vale a pena apostar em outras ferramentas e técnicas para testar os cenários que o Cucumber não cobre por natureza. (RIBEIRO, C., 2012).

"Cucumber não faz sentido a menos que você tenha clientes lendo os testes." (HANSSON[1], 2011).

[1] Hansson: DAVID HEINEMEIER HANSSON, criador do Ruby on Rails. Disponível em: <http://david.heinemeierhansson.com/>. Acesso em 20 de Julho de 2012.

CAPITULO IV

4. OUTRAS FERRAMENTAS

Outras ferramentas foram pesquisadas, porém, não foram mencionadas com mais ênfase neste trabalho, pois, as mesmas não dispunham de informações suficientes que contivessem conteúdo para descrevê-los.

As informações relatadas abaixo descrevem os comentários empíricos dos autores em questão.

Concordion é capaz de gerar especificações ativas, sendo uma ferramenta adequada à aplicação de Desenvolvimento Orientado a Comportamento. Concordion oferece aos membros das equipes de implementação a capacidade de mapeamento das especificações de funcionalidade do software em código executável de teste, que podem ser escritos desde o princípio, direcionando toda a atividade de desenvolvimento, com isso, Concordion também permite a rastreabilidade desde a especificação até o código-fonte relativa à implementação correspondente, já incentivando sua cobertura por testes.

Documentos de especificação tratados pelo Concordion são especificações ativas por possibilitarem total visibilidade aos clientes e especialistas em negócio sobre o andamento das atividades, dando indicativas de forma clara e inequívoca sobre que funcionalidades especificadas se encontram implementadas. (ANDRADE, 2010).

Shore (2010, tradução nossa) declara que as experiências com a ferramenta **FIT** e outras ferramentas de testes de aceitação ágeis é que eles custam mais do que valem.

O benefício dessas ferramentas é que os clientes devem escrever os exemplos, assim melhorando a comunicação entre clientes e programadores. Na prática descobrir que os clientes não se interessam em fazer isso, muitas vezes não conseguem entender e não confiam em testes que foram escritos por outros.

Normalmente, a responsabilidade para os testes ficam entregues aos testadores.

Esses problemas, do cliente não participar, elimina o propósito do teste de aceitação, significado que o teste de aceitação não vale a pena o custo.

CAPITULO V

5. CASO DE TESTE

O caso de teste apresentado relata a história de Janet e Dave, proprietários do "Beautiful Tea" uma boutique de chá no estado de Byron Bay interior da Austrália. Eles cultivam e vendem folhas orgânicas, direto para apreciadores de chá, utilizando um aplicativo de pedidos on-line desenvolvido alguns anos atrás. O aplicativo de pedidos on-line foi a princípio desenvolvido e mantido por uma empresa externa, mas devido ao mau serviço e custos excessivos, decidiram por migrar a aplicação e fazê-la "em casa". Nessa transição, Dave e Janet decidiram montar uma pequena equipe para desenvolvedor e manter a aplicação de pedidos on-line.

Equipe de Funcionário da Beautiful Tea:

Janet e Dave: Proprietários e Especialistas no assunto

Henry: Suporte ao Cliente

Mark: Analista de Negócio

Madison: Testador

Monique: Programadora

Problemas de Velocidade e Qualidade:

Desde que foi criada, a aplicação de pedidos on-line da Beautiful Tea tem tido problemas com qualidade e velocidade.

Estes problemas tem significado não somente a perda diretamente de dinheiro (por exemplo, calcular mal o frete dos transportes e os Impostos sobre Mercadorias e Serviços - GST[1]), mas também tem prejudicado os planos de expansão dos negócios. Por exemplo, Janet e Dave tem tido uma demanda grande de pequenas varejistas especialistas em chá, para que eles possam revender seus produtos, mas a aplicação de pedidos on-line é tão vulnerável e difícil mudar, que eles não foram capazes de fazer as alterações necessárias para facilitar suas vendas.

A documentação do sistema fornecido pela empresa externa não era apenas escasso, como confuso de ler e carente de informações. A única verdade sobre o sistema é o sistema, isto quer dizer, que qualquer problema simples de ser resolvido levava uma grande quantidade de tempo, significando que poucas mudanças não poderiam ser feitas em um curto período de tempo.

Primeira Fase: Testes Automatizados

Devido a problemas de qualidade que ocorrem quando são apresentadas quaisquer mudanças, Madison, o testador da Beautiful Tea, deu início a uma missão para criar testes automatizados utilizando o Selenium.

Exemplo de um Roteiro (Script) de Teste Automatizado:

```
assertTitle "Beautiful Tea"
pause "2000"
clickAndWait "link=Teas"
pause "3000"
assertTitle "Range of Teas"
clickAndWait "link=Byron Breakfast"
assertTitle "Byron Breakfast"
click "//input[@name='buy' and @value='1']"
type "quantity", "6"
clickAndWait "//input[@value='Add to cart']"
assertTitle "Beautiful Tea Cart"
open http://beautifultea.com
```

[1] No Brasil o GST é equivalente ao ICMS (Imposto sobre Circulação de Mercadorias e Serviços).

Embora Madison, tenha rodado o roteiro do Selenium rapidamente, e em sua maioria gravando cada roteiro numa IDE do Selenium, problemas também começaram a surgir rapidamente.

- Devido ao grande número de scripts gerados, uma pequena alteração na aplicação significava muitos testes sem funcionar, sendo que estes haviam levado horas para serem executados;
- Os dados de testes foram difíceis de codificar no roteiro de teste, embora Madison tenha achado difícil executar os testes em diferentes ambientes;
- Para cada roteiro foi escrito um código, e logo os proprietários Janet e Dave tiveram dificuldade em entender o propósito de cada script, especialmente quando incluíam no script XPath e CSS, e;
- Madison era o único da equipe que entendia completamente o conjunto dos testes.

Segunda Fase: Cucumber

Madinson começou a perceber os problemas associados ao utilizar scripts em testes automatizados, e começou a ler muito sobre o Cucumber que serve de apoio ao BDD, modelo de framework que esta se tornando cada vez mais popular no mundo de teste ágil. Como não tinha nada a perder, Madison começou a converter seus roteiros existentes no Selenium para cenários do Cucumber e definição de passos.

Exemplo de uma Funcionalidade no Cucumber:

```
# language: pt
Funcionalidade: Frete Beautiful Tea
  Cenario: Frete gratis na Australia
        Dado Eu estou na home page da Beautiful Tea
        Quando Eu procuro pelo cha Byron Breakfast
        Entao Eu vejo a pagina do cha Byron Breakfast
        Quando Eu adiciono o cha Byron Breakfast para o meu carrinho
        E Eu seleciono 10 na quantidade
        Entao Eu vejo 10x  cha Byron Breakfast no meu carrinho
        Quando Eu seleciono Check Out
        E Eu entro no meu pais Australia
        Entao Eu vejo o total incluso do GST
        E Eu vejo que eu estou legivel para frete gratis
  Cenario: Sem frete gratis para fora da Australia
```

```
        Dado Eu estou na home page do Beautiful Tea
        Quando Eu procuro pelo cha Byron Breakfast
        Entao Eu vejo a pagina do cha Byron Breakfast
        Quando Eu adiciono o cha Byron Breakfast para o meu carrinho
        E Eu seleciono 10 na quantidade
        Entao Eu vejo 10x  cha Byron Breakfast no meu carrinho
        Quando Eu seleciono Check Out
        E Eu entro no meu pais New Zealand
        Entao Eu vejo o total sem GST
        E Eu vejo que eu nao estou legivel para frete gratis
Cenario: Sem frete gratis na Australia
        Dado Eu estou na home page do Beautiful Tea
        Quando Eu procuro pelo cha Byron Breakfast
        Entao Eu vejo a pagina do cha Byron Breakfast
        Quando Eu adiciono o cha Byron Breakfast para o meu carrinho
        E Eu seleciono 1 na quantidade
        Entao Eu vejo 1x  cha Byron Breakfast no meu carrinho
        Quando Eu seleciono Check Out
        E Eu entro no meu pais Australia
        Entao Eu vejo o total incluso do GST
        E Eu vejo que eu nao estou legivel para frete gratis
3 scenarios (3 undefined)
30 steps (30 undefined)
0m0.033s
```

Levou algum tempo para converter esses roteiros em cenários para o Cucumber, mas após algumas iterações Madison percebeu que estes estavam funcionando muito bem.

Foi apenas quando uma mudança considerável foi planejada para todo o processo de compra que Madison percebeu a necessidade de alterar o escopo dos seus cenários com o Cucumber. Madison organizou um workshop com toda a equipe, poucos dias antes de uma nova iteração para discutir as mudanças necessárias com as funcionalidades do Cucumber. Mark, o analista de negócio, assim como Janet e Dave, os especialistas no assunto, apontaram rapidamente como foi difícil entender as diferenças entre os diversos cenários. Eles também encontraram algumas terminologias que não correspondiam com o que foi usado em torno das

propriedades do chá. Madison também percebeu que eles precisavam constantemente explicar o que estava acontecendo, no caso de não ser algo que eles queriam então alguns desses cenários exigiam um pouco mais de trabalho.

Terceira Fase: Entendimento Comum: Rumo as Especificações Executáveis

Madison fez algumas pesquisas e encontrou alguns princípios que poderiam ajudá-los a melhorar os cenários do Cucumber que eles haviam escritos.

- Especificações, não roteiros: ela deve se basear menos, em fluxo de trabalho e mais em especificações sobre o que é necessário, pois elas são mais fáceis de entender, mais precisos e testáveis;
- Abstrata: uma especificação deve ser abstrata o suficiente para enfatizar os detalhes, remover o ruído, e não ficar vinculado a implementação de interface com o usuário;
- Linguagem onipresente: as especificações devem ser consistentes ao longo do processo de desenvolvimento para assegurar um entendimento comum;
- Casos extremos: divergências incomuns devem ser especificadas para assegurar clareza nas expectativas: se algo esta óbvio de mais, é onde está o perigo;
- Exemplos importantes: cada ponto de decisão deve ter 5 á 6 exemplos importante, e não mais. Estes podem ser criados focando nas diferenças entre os cenários existentes;
- Acessível: a publicação das especificações para Janet, Dave e outros podem facilmente acessar as últimas versões.

Com este arsenal de truques, Madison organizou um workshop para reescrever no Cucumber os testes como especificação.

Especificação por Exemplo:

Funcionalidade: Custo do Frete Beautiful Tea

- Clientes Australianos pagam GST
- Clientes no Exterior não pagam GST
- Clientes Australianos obtêm frete grátis acima de $100
- Clientes no Exterior sempre pagam o frete, independente do tamanho do pedido

Cenário: Calculo GST por Estado e Taxa de envio

Dado que o cliente e de <país do cliente>

Quando o total de ordem do cliente <total do pedido>

Então o cliente <paga GST>

E eles são cobrados <taxa de envio>

País do Cliente	Paga GST	Total do Pedido	Taxa de Envio
Australia	Deve	$ 99.99	Padrão Nacional
Australia	Deve	$ 100.00	Grátis
New Zealand	Não Deve	$ 99.99	Padrão Internacional
New Zealand	Não Deve	$ 100.00	Padrão Internacional
Zimbawbe	Não Deve	$ 100.00	Padrão Internacional

Tabela 1. Especificação por Exemplo formato tabela.

Fonte: Specification by Example: a Love Story (2011)

Imediatamente a equipe pode olhar para a especificação e ver de forma clara sobre os custos do frete e o estado que deve pagar GST, algo que antes havia sido confuso e tinha custado em produção muito dinheiro para o Beautiful Tea.

Quarta Fase: Documentação Viva

Agora que a colaboração tinha começado na criação das especificações, estes se tornaram mais e mais o ponto focal para qualquer mudança. Madison passou com frequência a usar conversas colaborativas no envolvimento dessas especificações, consistindo do analista de negócio, os especialistas no assunto, o testador e uma programadora. Todos começaram a se sentir responsáveis pela qualidade.

Madison encontrou as especificações e as associou aos testes de aceitação muito mais fácil para manter com todos compreendendo, ficando assim menos ligados a implementação dos pedidos on-line.

Henry descobriu que as especificações eram vitais em seu papel de suporte ao cliente, e Monique a programadora se viu respondendo menos perguntas sobre o que o sistema de pedidos on-line realmente faz.

E o mais importante, as especificações executáveis estavam sempre atualizadas, isto significava que era muito mais fácil atualizar o sistema para dar suporte ao negócio sem ter o risco de introduzir questões indesejáveis. Os proprietários Janet e Dave foram capazes de especificar novas funcionalidades de pedidos online para revendedores e isto foi facilmente incorporado dentro da aplicação pela equipe.

Entregando a nova funcionalidade rapidamente e sem problemas significou para a empresa, capacidade de crescimento perante o negócio.

CONCLUSÃO

Este trabalho começou apresentando os processos da Engenharia de Software e as suas dificuldades para se descobrir as necessidades dos *Stakeholders*, além da falta de uma comunicação eficaz, que é de extrema importância em todas as etapas da construção do software.

Com o intuito de obter uma melhoria contínua na qualidade do requisito, foi apresentado a Técnica de Especificação por Exemplo, que comprovou que, especificações bem escritas atuam como um tipo de Requisitos Executáveis que ajudam a manter o entendimento compartilhado por todos da equipe de desenvolvimento, tendo como objetivo a melhoria na comunicação e o aprimoramento na colaboração com o cliente sobre como o sistema de software representa os problemas do mundo real.

Com a adoção da Técnica de Especificação por Exemplo e com o auxílio da ferramenta Cucumber como uma alternativa viável para a resolução dos problemas de uma das etapas da Engenharia de Requisitos que é a Especificação de Requisitos, o Cucumber vem com o objetivo de fazer com que essas histórias sejam escritas em conjunto com o cliente final, tornando a partir dai, Especificações Executáveis em vez de um mero papel que não tem como validar automaticamente.

Embora, tenhamos que reescrever tudo em Ruby, ou em outra linguagem, muitos dizem que, se você não pretende usar isso com um cliente de verdade, parece um trabalho duplicado, porém, não importa onde e de que forma você vai coletar as histórias. Pode-se utilizar desde um software sofisticado a uma folha de papel; no final o que interessa mesmo não é se o cliente escreve ou não o arquivo do Cucumber, mas sim aproximar o cliente. Isso normalmente é feito em uma conversa, em que as necessidades são anotadas e transferidas para os testes do software de qualquer forma, o fluxo deve ser assim com o Cucumber ou com qualquer outra ferramenta.

Além disso, não importa se seu cliente escreve ou não arquivos; o importante é ter um cliente ativo durante o desenvolvimento. Poder integrar o Cucumber nesse processo é apenas uma das possibilidades, e não algo obrigatório quando se usa a ferramenta.

Sommervile (2009, p. 41) relatou alguns pontos, que o mesmo afirma que são difíceis de concretizar:

- Embora a ideia de envolvimento do cliente no processo de desenvolvimento seja atraente, seu sucesso depende de um cliente disposto e capaz de passar o tempo com a equipe de desenvolvimento, e que possa representar todos os *Stakeholders* do sistema. Frequentemente, os representantes dos clientes estão sujeitos a diversas pressões e não podem participar plenamente do desenvolvimento de software.
- Membros individuais da equipe podem não ter personalidade adequada para o intenso envolvimento que e típico dos métodos ágeis e, portanto, não interagem bem com outros membros da equipe.
- Priorizar as mudanças pode ser extremamente difícil, especialmente em sistemas nos quais existem muitos *Stakeholders*. Normalmente, cada stakeholder dá prioridades diferentes para mudanças diferentes.
- Manter a simplicidade exige um trabalho extra. Sob a pressão de cronogramas de entrega, os membros da equipe podem não ter tempo para fazer as simplificações desejáveis.
- Muitas organizações, principalmente as grandes empresas, passaram anos mudando sua cultura para que os processos fossem definidos e seguidos. É difícil para eles mudar de um modelo de trabalho em que os processos são informais e definidos pelas equipes de desenvolvimento.

Embora tal técnica tenha sua controvérsia, existem razões sólidas pelas quais podem funcionar de fato.

O estudo de caso apresentado mostrou que o projeto foi capaz de gerar um conjunto de funcionalidades que atenderam às necessidades dos usuários de forma adequada mediante a rapidez na entrega e a confiança gerada por toda equipe pela entrega do trabalho bem feito.

Em vista dos argumentos apresentados é de suma importância mencionar que a técnica apresentada, contribui para um requisito de maior qualidade, pois o foco esta na melhoria da comunicação entre os envolvidos no projeto, entretanto, conforme mencionado por Brooks

(2010, tradução nossa) em uma entrevista dada a revista WIRED, modele, modele e modele e busque por críticas sábias, em torno de toda a melhoria do software.

REFERÊNCIAS BIBLIOGRÁFICAS

ADZIC, Gojko. **Specification by Example**: How successful teams deliver the right software – Shelter Island, NY, 2011.

ADZIC, Gojko. **Winning Big with Specification by Example** – Lessons learned from 50 successful projects. London-United Kingdom, 2011.

BROOKS, Frederick P. **O mítico homem-mês**: ensaios sobre engenharia de software. Edição de 20º Aniversário – Rio de Janeiro: Elsevier, 2009.

BROOKS, Frederick P., "**No Silver Bullet: Essence and Accidents of Software Engineering**" Computer, Vol. 20, Nº 4 (Abril 1987) pp. 10-19.

CRISPIN, Lisa e GREGORY, Janet. **Agile Testing. A Practical Guide for Testers and Agile Teams**. Boston,MA: Pearson Education, 2009.

COHN, Mike. **User Stories Applied For Agile Software Development**. Boston, MA,Addison Wesley, 2004.

DALLAVALLE, Silvia Inês; CAZARINI e WALMIR, Edson, **Regras do Negócio, um fator chave de sucesso no processo de desenvolvimento de Sistemas de Informação**. Escola de Engenharia de São Carlos - USP-EESC, p. 1-8, 2000.

FAULK, S. R., Software Requirements: **A Tutorial, in Software Requirements Engineering**, 2nd. Ed., IEEE CS Press, 1997, pp 128-149.

FLORINIER, David de e Adzic, GOJKO. **The Secret Ninja Cucumber Scrolls**. Strictly Confidential. London, United Kingdom, 2010.

GAZOLA, Alexandre. Design Ágil e Evolutivo com TDD. **Benefícios do TDD medidos na prática**, Rio de Janeiro, v. 54, ano IX, p. 6-14, jul/ago. 2012.

GOGUEN, Joseph A. **Techniques for Requirements Elicitation** in Software Requirements Engineering, IEEECS Press, Second Edition, 1997, p.p.110 –122, apud BELGAMO, Anderson e MARTINS, Luiz Eduardo Galvão, **Estudo Comparativo sobre as Técnicas de Elicitação de Requisitos do Software**, Universidade Metodista de Piracicaba, p.1-8, 2000.

HARROLD, Mary Jane. **Testing: A Roadmap. In: International Conference on Software Enginnering Future of Software Engineering**, 22, 2000, Atlanta. Atlanta, 2000, p. 6172, apud BUHLER, Luciano. **Análise Comparativa de Ferramentas Gratuitas para Teste de Software Orientado a Objetos**. Universidade de Passo Fundo, 2007.

HEUMANN, Jim. **The Five Levels of Requirements Management Maturity**, Copyright Rational Software, feb 2003.

JACOBSON, I., BOOCH, G. and RUMBAUGH, J.: **Unified Software Development Process**. Rational Software Corporation. Addison-Wesley Object Technology Series. Jan., 1999, apud PUC-Rio - Certificação Digital nº 0210666/CA, **Engenharia de Requisitos**, cap.02, p. 26.

KOTONYA, G.; SOMMERVILLE, I. **Requirements Engineering – Processes and Techniques**. Chichester, England : John Wiley & Sons Inc., 1998, apud CHICHINELLI, Micheli, **Contribuição da Técnica de Modelagem Organizacional i* ao Processo de Engenharia de Requisitos, com destaque aos requisitos não Funcionais**, Escola de Engenharia de São Carlos da Universidade de São Paulo, 2002

MACAULAY, Linda. **Requirements Engineering Techniques**. Department of Computation, University of Manchester Institute of Science and Technology, Manchester, p. 157-164, 1996.

MARTIN, Robert C. **O Codificador Limpo**: um código de conduta para programadores profissionais. Rio de Janeiro/RJ: Alta Books, 2012.

McCONNEL, Steve. **Software Project Survival Guide: How to Be Sure Your First Important Project isn't Your Last**. 1998, p.p 113-124, apud BELGAMO, Anderson e MARTINS, Luiz Eduardo Galvão, **Estudo Comparativo sobre as Técnicas de Elicitação de Requisitos do Software**, Universidade Metodista de Piracicaba, p.1-8, 2000.

McEWEN, Scott. **Requirements: An introduction**, Metasys Technologies, Inc. 16 Apr 2004.

McConnell, Steve. **Code Complete; Second Edition**, Microsoft Press (2004), apud PINTO SILVEIRA, Maria Clara dos Santos. **A Reutilização de Requisitos no Desenvolvimento e Adaptação de Produtos de Software**. Tese (Doutor em Engenharia Eletrotécnica e de Computadores) Universidade do Porto Faculdade de Engenharia, 2006.

NUNES, Demetrius. Anos de Design Patterns, O Que Ficou e o Que Mudou. **Automação de Testes de Aceitação com Cucumber e JRuby**, Rio de Janeiro, v. 39, ano VII, p. 36-44, jan/fev. 2010.

OSTERWALDER, Alexander e PIGNEUR, Yves. **Business Model Generation** - A Handbook for Visionaries, Game Changers, and Challengers. Hoboken, New Jersey. John Wiley & Sons, Inc., 2010.

PFLEEGER, S. L. **Engenharia de Software: Teoria e Prática**, 2ª edição, Prentice Hall, ISBN 85-87918-31-1, São Paulo - SP – Brasil, 2004 apud ANDRADE, Marcelo de Freitas. **Um Estudo de Caso de Especificações Ativas de Requisitos de Software.** ConSerpro (Conselho Serpro de Tecnologia e Gestão aplicadas a Serviços Públicos), 2010.

PRESSMAN, Roger S. **Engenharia de Software**. 6ª ed. Rio de Janeiro: McGrawHill, 2006.

RIBEIRO, Cássio L**., A Relação Entre Desenvolvimento Orientado a Testes e Qualidade de Software**. Goiânia/GO, 2010

SOMMERVILLE, Ian. **Engenharia de Software** 9º ed. São Paulo: Pearson Prentice Hall, 2011.

TELES, Vinícius M. **Extreme Programming:** aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade. São Paulo: Novatec, 2004, apud TONIAZZO, José Carlos. **Extreme Programming: Uma Abordagem em Testes de Software Utilizando XUnit**. Chapecó (SC), 2007, 90f. - Programa de Pós-Graduação em Engenharia e Qualidade de Software, Universidade Comunitária Regional de Chapecó, Santa Catarina, 2007.

THAYER, R. H.; DORFMAN, M.; "**Introduction to Tutorial Software Requirements Enginnering**" in Software Requirements Engineering, IEEE-CS Press, Second Edition, 1997, p.p. 1-2, apud BELGAMO, Anderson e MARTINS, Luiz Eduardo Galvão, **Estudo Comparativo sobre as Técnicas de Elicitação de Requisitos do Software**, Universidade Metodista de Piracicaba, p.1-8, 2000.

THE STANDISH GROUP, **CHAOS**, Massachusetts, EUA, 1995.

WYNNE, Matt e HELLESOY, Aslak. **The Cucumber Book** - Behaviour-Driven Development for Testers and Developers. Dallas, Texas. Pragmatic Programmers, 2012.

REFERÊNCIAS WEB

BOKHARI, Mohammad Ubaidullah; SIDDIQUI, Shams Tabrez (2011). Disponível em: <http://www.academia.edu/1297947/Metrics_for_Requirements_Engineering_and_Automated_Requirements_Tools>. Acesso em 23 de Outubro de 2012.

DAVIS, Alan M. **Software Requirements - Objects, Functions, and States**. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1993, apud ALENCAR, F. M. R. **Mapeando a modelagem organizacional em especificações precisas**. Recife, 1999. 304f. Tese (Doutorado em Informática) - Centro de Informática, Universidade Federal de Pernambuco.

DAVIS, Alan M. **Software Requirements - Objects, Functions, and States**. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1993, apud BOKHARI, Mohammad Ubaidullah; SIDDIQUI, Shams Tabrez (2011). Disponível em: <http://www.academia.edu/1297947/Metrics_for_Requirements_Engineering_and_Automated_Requirements_Tools>. Acesso em 23 de Outubro de 2012.

FOWLER, Martin. **Specification by Example**. Disponível em: <http://martinfowler.com/bliki/SpecificationByExample.html>. Acesso em: 17 de Outubro 2012.

KELLY, Kevin. **Master Planner: Fred Brooks Shows How to Design Anything**. Publicado na revista WIRED em 28 de Julho de 2010. Disponível em: <http://www.wired.com/magazine/2010/07/ff_fred_brooks>. Acesso em: 20 de Julho de 2012.

NUSEIBEH, Bashar; EASTERBROOK, Steve. **Requirements engineering**: A roadmap International ConferenceSoftware Engineering, 2000 apud BOKHARI, Mohammad Ubaidullah; SIDDIQUI, Shams Tabrez (2011). Disponível em: <http://www.academia.edu/1297947/Metrics_for_Requirements_Engineering_and_Automated_Requirements_Tools>. Acesso em 23 de Outubro de 2012.

PAPO, José. **Testes em um Mundo Ágil**, 2011. Disponível em: <http://www.slideshare.net/jpapo/especificao-por-exemplos-e-testers>. Acesso em: 20 de Julho de 2012.

RIBEIRO, **Camilo. Entendendo BDD com Cucumber** (2012). Disponível em <http://www.bugbang.com.br/entendendo-bdd-com-cucumber-parte-i/>. Acesso em 23 de Julho de 2012.

SHORE, James. **The Problems with Acceptance Testing** (2010). Disponível em: <http://jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html>. Acesso em: 18 de Outubro de 2012.

SWEBOK - **Guide to the Software Engineering Body of Knowledge**, 2004. by Alain Abran e James W. Moore Disponível em <http://www.computer.org/portal/web/swebok/html/ch2#ch2-5>. Acesso em: 19 de Julho de 2012.

TELES, Vinícius Manhães. Um estudo de caso da adoção das práticas e valores do Extreme Programming. Rio de Janeiro: UFRJ, 2005. 181 f. Dissertação (Mestrado em Informática) - **Um estudo de caso da Adoção das Práticas e Valores do Extreme Programming**, Universidade Federal do Rio de Janeiro - UFRJ, Rio de Janeiro, 2005. Disponível em:

<www.improveit.com.br/xp/dissertacaoXP.pdf>. Acesso em: 08 de Agosto de 2012.

THE STANDISH GROUP INTERNATIONAL, Inc. **Extreme chaos**. The Standish Group International, Inc, 2001. Disponível em: <http://www.standishgroup.com/sample_research/PDFpages/extreme_chaos.pdf>. Acesso

em: 23/12/2004, apud TELES, Vinícius Manhães. Um estudo de caso da adoção das práticas e valores do Extreme Programming. Rio de Janeiro: UFRJ, 2005. 181 f. Dissertação (Mestrado em Informática) - **Um estudo de caso da Adoção das Práticas e Valores do Extreme Programming**, Universidade Federal do Rio de Janeiro - UFRJ, Rio de Janeiro, 2005.

GLOSSÁRIO

Elicitação

O termo Elicitação vem da palavra inglesa elicitation, que significa descobrir, desvendar algo obscuro. Embora o termo Elicitação não exista no vocabulário da língua portuguesa ele vem sendo aceito e utilizado na comunidade de Engenharia de Requisitos como tradução do termo elicitation.

Stakeholder

São pessoas ou organizações que estão envolvidos ativamente no projeto; possuem interesses que possam ser positiva ou negativamente afetados pelo desempenho ou conclusão do projeto e que possam exercer influência sobre o projeto, seus entregáveis e/ou membros da equipe do projeto.

SWEBOK

Um documento criado com a finalidade de servir como referência em assuntos considerados como essenciais na área de Engenharia de Software e foi conduzido pelo IEEE (Institute of Electrical and Electronics Engineers).

Workshop

É uma reunião de um grupo de pessoas interessados em um determinado assunto ou pode ser uma atividade para discussão sobre um tema que é de interesse para todos.