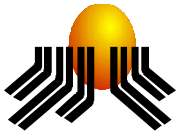


UNIVERSIDADE METODISTA DE PIRACICABA

FACULDADE DE CIÊNCIAS EXATAS E DA NATUREZA

MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

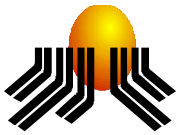
UM MODELO PARA PEQUENAS EMPRESAS INICIAREM PROCESSOS DE QUALIDADE ATRAVÉS DE ASPECTOS EXTRAÍDOS DE MÉTODOS ÁGEIS

GERSON NUNHO CARRIEL

ORIENTADOR: PROF.DR. PLÍNIO ROBERTO SOUZA VILELA

PIRACICABA, SP

2008



UNIVERSIDADE METODISTA DE PIRACICABA

FACULDADE DE CIÊNCIAS EXATAS E DA NATUREZA

MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

UM MODELO PARA PEQUENAS EMPRESAS INICIAREM PROCESSOS DE QUALIDADE ATRAVÉS DE ASPECTOS EXTRAÍDOS DE MÉTODOS ÁGEIS

GERSON NUNHO CARRIEL

ORIENTADOR: PROF.DR. PLÍNIO ROBERTO SOUZA VILELA

Dissertação apresentada ao Mestrado em Ciência da Computação, da Faculdade de Ciências Exatas e da Natureza, da Universidade Metodista de Piracicaba – UNIMEP, como requisito para obtenção do Título de Mestre em Ciência da Computação.

PIRACICABA, SP

2008

UM MODELO PARA PEQUENAS EMPRESAS INICIAREM PROCESSOS DE QUALIDADE ATRAVÉS DE ASPECTOS EXTRAÍDOS DE MÉTODOS ÁGEIS

AUTOR: GERSON NUNHO CARRIEL

ORIENTADOR: PROF. DR. PLÍNIO ROBERTO SOUZA VILELA

Dissertação de Mestrado defendida e aprovada em 28 de Agosto de 2008, pela Banca Examinadora constituída dos Professores:

Prof. Dr. Plínio Roberto Souza Vilela – UNIMEP (Orientador)
Prof. Dr. Clênio Figueiredo Salviano - CenPRA
Prof. Dr. Ana Estela Antunes da Silva - UNIMEP

À

Minha família que sem perceber sempre me deu suporte nas caminhadas mais difíceis, especialmente meus pais: Gentil e Eloísa.

AGRADECIMENTOS

Ao professor Plínio Roberto Souza Vilela a orientação e Incentivo dispensado ao desenvolvimento deste trabalho.

À instituição de fomento CAPES, Centro de Aperfeiçoamento de Pessoal Especializado, pelo apoio financeiro.

Aos amigos Washigton Tavares, Aldi Pereira e Flávio Dias Semim Jr ao permitir que o projeto ou parte dele fosse desenvolvido dentro de suas empresas, num trabalho conjunto pela melhoria de qualidade no desenvolvimento de software.

Aos docentes da Faculdade de Ciências Exatas e da Natureza, programa de Mestrado em Ciência da computação: Prof. Dr. Luiz Eduardo Galvão Martins, Prof. Dr. Marcio Merino Fernandes, Prof. Dr. Nivaldi Calonego Júnior, Prof. Dr. Marina Teresa Pires Vieira, Prof. Dr. Plinio Roberto Souza Vilela e Prof. Dr. Tereza Gonçalves Kirner, pela enorme contribuição em minha formação profissional.

“... Here we go. Focus. Speed. I am speed.

One Winner, 42 loosers. I eat Losers for breakfast.

Breakfast! Wait, may be I Should have had breakfast.

A little Breck-y could be good for me.

No, no, no, Stay focused. Speed.

I´m faster than fast. Quicker than quick.

I am Lightning!

Hey, Lightning! Are you Ready? ...”

Trecho inicial do filme Carros(original: Cars)

Disney Pixar 2006.

RESUMO

O desenvolvimento de software é realizado por muitas empresas, e muitas delas são pequenas empresas produtoras de software, onde a competitividade é sempre muito grande. Na busca da redução de custos de produção muitas dessas pequenas empresas não utilizam processos bem definidos de produção de software, alegando não poder aumentar o custo de produção. Este trabalho tem como objetivo apresentar o uso de modelos baseadas em desenvolvimento ágil, teste de software, iniciação a métricas e uso de ferramentas *open source* na busca por melhoria na qualidade de software com foco na pequena empresa, propondo pequenas mudanças em seus processos de produção de software e acompanhamento de seus resultados.

PALAVRAS-CHAVE: Desenvolvimento ágil, Teste de Software, Teste de Unidade

A SOFTWARE DEVELOPMENT MODEL BASED ON AGILE TECHNIQUES FOR QUALITY IMPROVEMENT IN SMALL SOFTWARE DEVELOPMENT COMPANIES

ABSTRACT

Software development is carried through by many companies. Several of these are small software producing companies, where competitiveness is always an issue. Searching for production costs reduction many of these small companies do not use well defined software development processes. This is motivated by a feeling that the company would not be able to afford the cost of process improvement. This work has as objective to present the use of models based on agile development, software test, introduction to metrics use and the use of open source tools in search for improvement in software quality focused in small companies. Small changes in their current software development processes are incorporated and the results evaluated.

KEYWORDS: Agile Software Development, Software Test, Unit Test

PIRACICABA, SP

2008

Sumário

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

ADQ	**Adicional de Qualidade.**
CFL	**Construir a Lista de Funcionalidades.**
CK	**Chidamber – Kemerer OO metrics.**
CMS	**Content Management System.**
CPF	**Construir por Funcionalidade.**
DMA	**Desenvolver um Modelo Abrangente.**
DPF	**Detalhar por Funcionalidade.**
FDD	**Feature-Driven Development.**
IPL	**Initial Program Load (reboot).**
KLOC[1]	**Thousand of Lines Of Code .**
LOC	**Lines Of Code**
LP	**Linguagem de Programação**
OO	**Orientado a Objetos**
PPF	**Planejar por Funcionalidade.**
TDD	**Teste-Driven Development.**
XP	**eXtreme Programming.**
XT	**eXtreme Testing.**
WMC	**Wighted Methods Per Class.**
DIT	**Depth of Inheritance Tree.**
PF	**Ponto de Função.**
NOC	**Number of Children.**
CBO	**Coupling Between Object Classes.**
RFC	**Response for a Class.**

[1] A abreviação utiliza a letra K que tem origem na palavra grega *khilioi*, cujo significado é mil.

LCOM	**Lack of Cohesion in Methods.**
PTR	**Problem Tracking Report (teste defects).**
STREW	**Software Testing and Reliability Early Warning.**
SLOC	**Source Lines of Code.**
TLOC	**Test Lines of Code.**
SM	**STREW Metric.**
GQM	**Goal Question Metric.**
CenPRA	**Centro de Pesquisas Renato Archer.**

LISTA DE QUADROS

1 INTRODUÇÃO

1.1 Contexto

No Brasil, o uso de processos definidos de qualidade em desenvolvimento de *software* vem crescendo, mas ainda há muito trabalho a ser realizado.

Tal afirmação pode ser percebida na pesquisa MCT (2006), a qual mostra que medidas para prevenção de defeitos não são muito utilizadas através do percentual que representa as empresas que utilizam medições de qualidade (métricas) no ano de 2005, que chega a apenas 23% das empresas pesquisadas, conforme apresentado no Quadro 1.

Outros valores apresentados referem-se às medidas de detecção/remoção de defeitos, que foram distribuídas entre várias técnicas, com percentuais variando de 18% a 54% no mesmo ano para empresas que as utilizam, como pode ser visto Quadro 2.

Outros números da mesma pesquisa mostram que 54% das empresas realizam testes de aceitação, testes de sistema integrado são realizados em 45% das empresas e os testes de unidade são realizados em 40%.

Quadro 1 - Métodos de prevenção de defeitos (MCT, 2006)

Técnicas de Prevenção de defeito	1995	1997	1999	2001	2005
Auditorias		17%	21%	23%	26%
Gerência de Configuração		7%	15%	23%	24%
Joint Application Design – JAD		8%	9%	8%	
Medições de Qualidade(Métricas)	10%	8%	12%	17%	23%
Prototipação	46%	44%	44%	51%	14%
Reuso(de código)	37%	19%	24%		

O resultado apresenta um percentual de 95% de confiabilidade, através de pesquisas realizadas num universo de aproximadamente 2500 empresas com margem percentual de erro calculada em 3.5%, conforme MCT(2006).

Quadro 2 - Métodos de Detecção e Remoção de defeitos (MCT, 2006)

Métodos de detecção/remoção de defeitos	1995	1997	1999	2001	2005
Inspeções Formais	10%	17%	20%	16%	18%
Revisões Estruturadas		19%	16%		
Testes de Aceitação	48%	47%	48%	57%	54%
Testes do Sistema Integrado	62%	67%	47%	52%	45%
Testes de Unidade	24%	23%	31%	35%	40%
Validação		42%	45%		

Observando os percentuais é possível visualizar a necessidade de trabalho e de soluções para melhoria de qualidade no ambiente de desenvolvimento de *software*, ou seja, existe muito trabalho a ser desenvolvido para melhoria em processos de prevenção, como de detecção e remoção de defeitos.

Em pesquisa apresentada em 2000, segundo MCT/SEPIN(2000), empresas de *software que possuem até 5 funcionários rep*resentam 26,5% das empresas pesquisadas e empresas com funcionários efetivos entre 6 e 10 representam mais 16,5%. Esses percentuais mostram que 43% das empresas pesquisadas eram microempresas, conforme denominação de MCT/SEPIN (2000).

Neste trabalho, o grupo de 43% das empresas tratadas na pesquisa por microempresas, será tratado por: Pequenas empresas produtoras de software.

1.2 Motivações

A forma padrão de desenvolvimento de software, principalmente na pequena empresa, começa pelo código (codificação feita pelo desenvolvedor), conforme afirma BEZERRA (2004). Essa afirmação vem ao encontro da realidade encontrada no mercado, onde as empresas que começam com pouquíssimos funcionários têm no código tudo relacionado ao seu produto de trabalho.

Com o aumento do grau de complexidade através do crescimento do número de funcionários e clientes, essas empresas se deparam com problemas como atrasos em seu desenvolvimento e entregas, além de terem por parte de seus clientes uma percepção de qualidade inferior ao desejado, isso percebido pela quantidade de defeitos identificados pelos clientes e pelos prazos que não são cumpridos.

Essa pequena empresa produtora de software pode recorrer a alguma consultoria para melhoria de qualidade, mas esbarram no problema financeiro de uma contratação desse porte, implementar atividades propostas em livros de forma desordenada, também não elevam significantemente a qualidade devido às "revoluções" propostas em algumas literaturas.

Além de cenários como os descritos acima, temos os números apresentados no MCT(2006) que motivam o trabalho aqui proposto, que busca organizar e definir um conjunto de modelos para melhorar a qualidade de *software*, onde pequenas empresas produtoras de software possam realizar melhoria na qualidade de *software* e acompanhar os resultados através de mudanças pequenas no processo desenvolvimento, como mudanças nos requisitos, no desenvolvimento de software e realização de testes e até buscando explorar as ferramentas gratuitas que existem ao seu alcance.

Em administração estratégica, um dos primeiros passos é identificar a empresa no seu contexto presente para depois conseguir atuar de forma estratégica em seu futuro, conforme ALDAY (2000). Nesta proposta representamos essa necessidade ao mostrar que a empresa em qualquer mudança deve buscar métricas de identificação de seu posicionamento no presente, para que se possa criar um objetivo futuro e que esse objetivo possa ser mensurável no decorrer do caminho através de métricas de produtividade e qualidade.

O enfoque será dado a microempresas[1], principalmente pela representatividade apresentada anteriormente e que pode ser visto em MCT/SEPIN (2006), empresas essas com número de funcionários menor que dez, mas o trabalho pode ser também aplicável a empresas maiores.

O conjunto de modelos tem um enfoque ágil na busca de qualidade e pode beneficiar empresas ao auxiliar na diminuição dos defeitos no código, melhoria no controle do software desenvolvido com uso de métricas, ao incentivar a coleta de dados para verificações de qualidade tanto antes da implantação de um dos modelos sugeridos, como o acompanhamento após a implantação e efetivação do uso de métricas no dia-a-dia da empresa, tudo isso tanto dentro do processo de desenvolvimento como no acompanhamento após a implantação do *software*.

Um processo ágil pode trazer qualidade através da simplicidade e *feedback* constantes com uso de ferramentas de engenharia de software.

A apresentação de números para a empresa de desenvolvimento, através do que aqui é proposto, irá motivar a empresa a continuar melhorando e acrescentando novos processos de melhoria ao

---

[1] Segundo MCT/SEPIN (2000) a microempresa é caracterizada pela quantidade de funcionários menor ou igual a dez.

aqui proposto à medida que a melhoria passa a ser percebida e verificada através de números.

1.3 Objetivo

O objetivo deste trabalho é o de apresentar modelos de cunho ágil e aplicá-los em pequenas empresas produtoras de software para melhorar o controle no processo de desenvolvimento de software, sem gerar mudanças grandes nas atividades diárias da empresa, como melhoria nos requisitos, o controle de testes no desenvolvimento, resultados de testes, assim como o acompanhamento após implantação em relação a defeitos reportados pelo cliente. Esses modelos serão utilizadas em empresas de São Paulo e São Roque, empresas estas com menos de dez funcionários. Ao final, é esperado que a quantidade de defeitos reportados pelo cliente seja inferior e com menor gravidade em relação aos valores apurados na empresa, antes da implantação do processo, e que essas mudanças motivem mais a empresa a procurar novas melhorias para o processo de desenvolvimento de software utilizado.

1.4 Organização

O trabalho está dividido em capítulos. Este Capítulo apresenta o tema e o objetivo da dissertação, assim como a estrutura na qual foi elaborado o texto.

O Capítulo 2 apresenta a Revisão de Literatura, mostrando em sua primeira sessão a necessidade do uso de metodologias no desenvolvimento de *software* e apresenta a metodologia de desenvolvimento de *software* em cascata, passando por uma descrição

de metodologia incremental chegando ao desenvolvimento ágil e sua receptividade por profissionais da área de desenvolvimento de *software*. Em suas sessões temos descrições mais específicas de algumas metodologias ágeis de desenvolvimento de *software*, expondo características do *eXtreme Programming*, Scrum, Crystal e o FDD. Além de uma abordagem generalizada de um aspecto importante de qualquer metodologia, que é a fase de testes e como os testes são controlados, apresenta as principais técnicas de testes, inclusive considerações sobre testes de *software* dentro de metodologia de desenvolvimento ágil, neste caso específico, o que é chamado de *eXtreme Testing (* XT).

Em seguida. temos mecanismos de controle através de métricas de *software*, são apresentados conceitos relacionados a essas métricas, algumas formas de medir *software* e algumas métricas relacionadas a testes em *software*, que tem a função de auxiliar no controle e gerenciamento de resultados de testes dentro de uma empresa com um processo de desenvolvimento definido.

A importância do uso de métricas de software é revista em seguida, onde métricas tradicionais, métricas orientadas a objetos e métricas *in-process* são apresentadas.

Ao final desse Capítulo, a metodologia GQM é apresentada, já que ela define o caminho que uma parte do trabalho é desenvolvido.

O Capítulo 3 apresenta a metodologia definida para ser seguida nos três projetos.

O Capítulo 4 traz os resultados das três empresas, sendo dois resultados parciais.

No Capítulo 5, temos as conclusões tiradas a partir da metodologia seguida e dos resultados alcançados nos projetos executados.

2 REVISÃO DE LITERATURA

Em 1969, na conferência "NATO Science Committee", Fritz, Friedrich Ludwig Bauer apresentou a idéia de engenharia de *software*, e a necessidade de princípios de engenharia para que os *software*s possam ser desenvolvidos de forma a serem economicamente viáveis, confiáveis e que funcionem eficientemente conforme PRESSMAN (2001). Através de princípios de engenharias, muitos processos foram, no decorrer dos anos, sendo desenvolvidos e melhorados, assim como muitas metodologias surgiram. Algumas metodologias se destacam mais, como a clássica (modelo em cascata), incremental, espiral, chegando às metodologias ágeis como o *Extreme Programming*.

2.1 Metodologia de Desenvolvimento de Software

2.1.1 Cascata

O Modelo em Cascata foi primeiro a ser publicado segundo SOMMERVILLE (2003) e é o mais antigo de todos e o mais utilizado, conforme afirma PRESSMAN (2001). Para um sistema onde os requisitos não mudam, esse paradigma atende de forma satisfatória, como apresentado em BECK (1999), porém para ambientes onde mudanças são constantes, ou mesmo, quando o usuário não tem clareza nos requisitos, esse paradigma acaba por frustrar o usuário do produto desenvolvido, que após um longo período aguarda pelo *software* que, ao final, pode não atender suas necessidades, devido à dinâmica de mudanças nos processos e procedimentos das empresas. A ilustração 1 demonstra o processo em cascata.

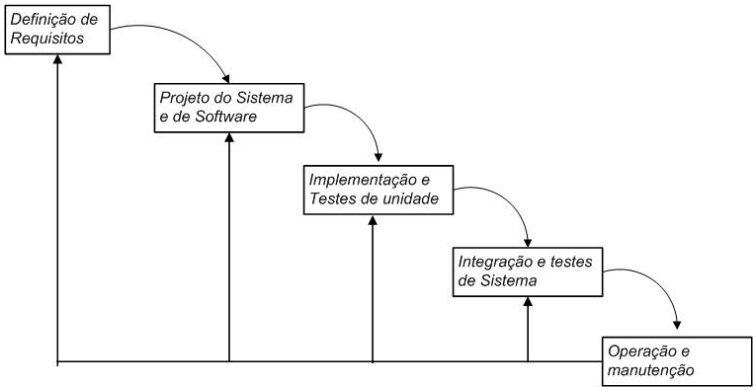


Ilustração 1 - Modelo em Cascata (SOMMERVILLE, 2003)

2.1.2 Prototipação

A prototipação auxilia um melhor alinhamento entre os requisitos recebidos e a realidade solicitada pelo usuário. Após criar o protótipo, o usuário é chamado para avaliar e corrigir falhas na interpretação, o que permite que clientes e desenvolvedores examinem alguns aspectos e decidam se é apropriado para o produto final PFLEEGER (2001).

PFLEEGER (2001) apresenta a vantagem de encontrar discrepâncias nos requisitos antes da chegada de fases mais custosas através do uso de prototipação, mas PRESSMAN (2001) apresenta algumas desvantagens:

1 - Pressão do cliente, ao ver que existe algo funcionando, sobre o desenvolvedor para colocar em produção como está;

2 - O próprio desenvolvedor acaba por esquecer o propósito do protótipo e mantém o código feito às pressas e sem boa definição de arquitetura.

Pode ser observado que, embora existam vantagens, o código feito para a prototipação acaba por influenciar os dois lados,

programadores e usuários, para que o código seja “ajustado" e acelere o processo de desenvolvimento.

2.1.3 Espiral

O modelo espiral acrescenta uma nova visão aos modelos em cascata e de prototipação, pois possui uma fase de avaliação de risco a cada volta num espiral, que é respresentado no processo, como pode ser visto na illustração 2.

Quatro partes distintas podem ser definidas no processo espiral, conforme PRESSMAN (2001):

1 - Estabelecer Comunicação com o cliente;

2 - Planejamento (define objetivo, alternativas e restrições);

2 - Análise de Risco (identifica/minimiza riscos);

3 - Engenharia (os processos de construção do software)

4 - Construção/versão (Construir, testar, instalar e suporte)

5 - Avaliação de resultado pelo Cliente

Essas atividades podem ocorrer várias vezes no decorrer do projeto, como pode ser visto na ilustração 2.

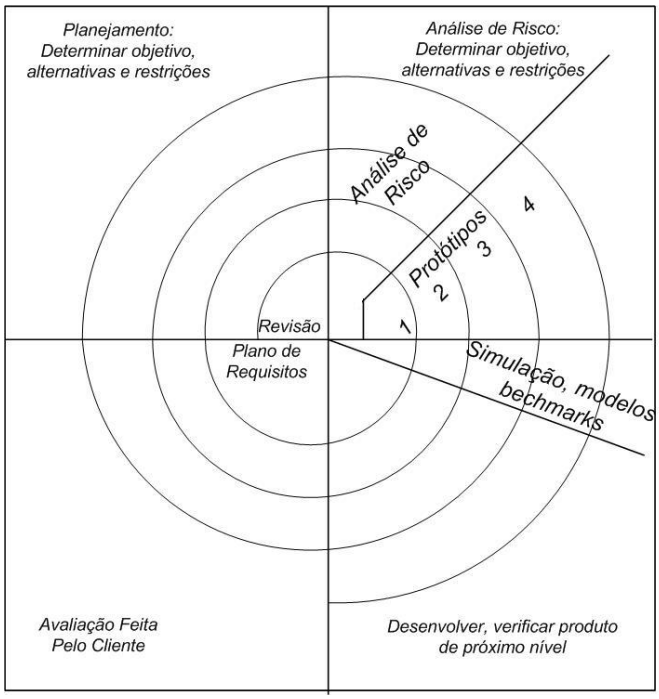


Ilustração 2 - Modelo Espiral (BOEHM, 1988)

2.1.4 Metodologia Ágil

Este modelo é marcado pelo uso de incrementos curtos, participação do cliente e uma preocupação grande em trabalhar voltado ao que o cliente necessita, não apenas acumular métodos e materiais de controle, mas sim trabalhar naquilo que é mais importante para o cliente.

COCKBURN (2002) expõe que, em fevereiro de 2001, em um Resort nas montanhas Wasatch, no estado Utah, um grupo de dezessete profissionais que trabalham com metodologias ágeis realizaram um encontro para discutir o que existia de comum entre essas metodologias existentes, e criaram um manifesto, com valores e princípios que indicam o que é ser ágil. Esse manifesto é chamado de manifesto ágil. Para AMBLER (2004) um ponto interessante nessa reunião é que todos são de áreas diferentes em suas formações e ainda assim concordam em pontos onde as metodologias normalmente não têm consenso.

2.1.4.1 Conceitos

O Manifesto Ágil é uma expressão de um grupo sobre pontos comuns em metodologias de desenvolvimento ágil. Dessa reunião alguns valores e princípios foram definidos para colocar o que existe de comum entre as metodologias ágeis. No Quadro 3, temos seus valores:

Quadro 3 - Valores XP (BECK, 2001)

Indivíduos e iterações	**valem mais**	**que processos e ferramentas**
***Software* em funcionamento**	**vale mais**	**que documentação extensa**
A colaboração do cliente	**vale mais**	**que as renegociações de contrato**
Enfrentar a mudança	**vale mais**	**que seguir o planejamento**

BECK (2001) define que o que está à esquerda(em negrito) tem mais valor para o desenvolvimento ágil de software em relação ao que está à direita.

O Manifesto ágil também apresenta princípios que devem nortear os praticantes de metodologias ágeis como apresenta COCKBURN (2002) abaixo:

1. **Nossa maior prioridade é satisfazer o cliente através da entrega contínua e antecipada de *software* de valor.**
2. **Nós acolhemos bem mudanças nos requisitos, mesmo em fases tardias no desenvolvimento. Processos ágeis aproveitam a mudança para permitir um diferencial ao cliente.**
3. **Entregar *software* funcionando freqüentemente, em períodos de poucas semanas ou meses, com preferência à menor escala de tempo possível.**
4. **O pessoal de negócios e os desenvolvedores devem trabalhar juntos diariamente durante todo o projeto.**

5. **Construir projetos com indivíduos motivados, dando-lhes o ambiente e o suporte necessário, além de confiar neles para que possam realizar o trabalho.**

6. **O método mais eficiente e efetivo de transmitir uma idéia para uma equipe de desenvolvimento é a conversa face-a-face.**

7. ***Software* em funcionamento é a principal medida de progresso.**

8. **Processos ágeis promovem o desenvolvimento sustentável. Os clientes, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente.**

9. **Atenção contínua à excelência técnica e um bom projeto intensificam a agilidade.**

10. **Simplicidade - a arte de maximizar o montante de trabalho não feito - é essencial.**

11. **As melhores arquiteturas, requisitos e projetos emergem de equipes auto-organizáveis.**

12. **Em intervalos regulares, a equipe reflete em como tornar-se mais efetiva, e então afina e ajusta o seu comportamento de modo adequado.**

2.1.4.2 XP - Extreme Programming

Em meados dos anos 90, Kent Beck iniciou uma nova abordagem ao desenvolvimento de *software* num projeto chamado C3. Essa nova metodologia visava deixar o projeto mais leve, garantir que

testes fossem feitos e refeitos, melhorar a comunicação entre os membros da equipe e entre os desenvolvedores e o cliente. Desse projeto, surgiu o Extreme Programming (Programação Extrema) de acordo com MYERS (2004).

O XP baseia-se em valores e práticas e é, de longe, a metodologia ágil mais utilizada, segundo MYERS (2004). O XP possui quatro valores, que norteiam os praticantes dessa metodologia, como apresentado em seguida (BECK, 2004):

Comunicação: mostra a necessidade dos participantes do projeto terem melhores formas de comunicação e argumenta sobre os problemas de projetos, nos quais programador, cliente e gerente de projetos não conseguem estar alinhados a respeito do projeto e que as práticas acabam por reforçar essa comunicação.

Simplicidade: como não programar pensando em possíveis necessidades futuras? É aí que a simplicidade entra: programar para o que é necessário apenas, não tentar fazer algo para o futuro, pois no futuro o requisito pode mudar e o esforço ser inútil.

***Feedback*: o *feedback* é muito importante para o projeto XP. Aparece nos testes de unidade em que o *feedback* é imediato, os programadores sabem ou não que uma determinada classe funciona e já estão a postos para realizar as alterações necessárias. Outro fator importante é que *software* ser disponibilizado cedo, pois assim o *feedback* de seu funcionamento irá auxiliar a equipe no tratamento de eventuais problemas, como afirma BECK (2004): o otimismo é uma doença profissional da programação e o Feedback é o seu tratamento.**

Coragem: este princípio é percebido no momento de verificar que algo tem que ser abandonado ou refatorado, quando a equipe decide jogar o código fora e começar do zero. Coragem é necessária para refatorar e deixar mais simples um código que esteja difícil de entender.

Além desses valores, existem princípios básicos definidos por BECK (2004), como o *feedback* rápido (auxilia na ação), a simplicidade presumida (pensar que sempre existe uma solução simples para resolver o problema), mudanças incrementais - realizar mudanças incrementais auxilia no controle, mudanças grandes geram dificuldade para encontrar onde ocorreu um erro- , aceitação da mudança e alta qualidade.

Partindo desses valores temos as 12 práticas definidas por BECK (2004), que devem ser observadas para que um projeto XP tenha o melhor do que a metodologia oferece.

O jogo do Planejamento: essa prática existe para explicar a relação que há entre a área técnica e a área de negócios. A área de negócios precisa definir escopo, prazo, prioridades e composição, por outro lado, a área técnica deve auxiliar, definindo estimativas, o processo que será utilizado e cronograma detalhado, como podemos ver na afirmação de BECK (2004), que diz que nem considerações de negócio, nem as considerações técnicas devem prevalecer, pois o desenvolvimento de *software* é sempre um diálogo evolutivo entre o que é possível e o desejável.

Entregas freqüentes: as versões devem ser entregues na menor versão possível e devem agregar o máximo de valor possível para o cliente.

Uso da Metáfora: deve-se existir uma metáfora que comunique para a equipe o que será o *software* a ser desenvolvido, uma metáfora que guie a equipe.

Projeto Simples: um projeto correto de *software* deve executar todos os testes, não ter lógica duplicada, expressar as intenções de quem o desenvolveu e possuir o menor número possível de método. Um projeto simples está diretamente relacionado ao que será necessário

no momento, e não tem uma preocupação com implementações futuras, que podem ser abandonadas.

Testes: a visão é que um programa que não possua testes automatizados, simplesmente não existe segundo BECK (2004), mas afirma que nem tudo deverá possuir testes, é ideal identificar o que é mais importante para o funcionamento do *software* desenvolvido.

Refatoração: para manter o programa simples, deve-se refatorar quando necessário deve-se trabalhar no código para deixá-lo mais simples e de fácil entendimento.

Programação em Pares: dois programadores em apenas um computador, lado a lado. Isso é programação em pares, desde que os dois executem seus papéis, um pensando na melhor maneira de fazer o código e outro pensando no lado estratégico (testes, abordagem, modo de simplifição) .

Propriedade coletiva: o código feito é de todos, não ocorre de um trecho ser de um único desenvolvedor, como é comum nas áreas de desenvolvimento. Quando for necessário refatorar, a dupla que identificou a necessidade irá trabalhar nesse processo, pois essa é a função deles, uma vez que o código é de todos.

Integração contínua: o código deve ser testado e integrado pelo menos uma vez ao dia, e o ideal é ter um computador à parte para isso. Esse código integrado deve ser deixado com 100% dos testes passados e a dupla que ocupar esse computador deve executar os testes antes de integrarem, para garantir que um possível defeito seja realmente dessa integração e não uma falha ou algo esquecido por outra dupla. Assim, a identificação dos defeitos é facilitada, pois esse procedimento garantirá que não existia defeito antes da integração e a dupla saberá o escopo de busca desse defeito.

Semana de 40 horas: é importante estar descansado pela manhã para iniciar as atividades do dia e terminar o dia com a sensação de dever cumprido, satisfeito com trabalho realizado, para poder descansar à noite. As horas extras podem aparecer no XP, mas não podem virar o padrão, pois isso indica algum tipo de erro no projeto como observa BECK (2004).

Cliente presente: um cliente real, junto à equipe, disponível para responder aos questionamentos e, principalmente, definir prioridades.

Um cliente junto à equipe não ficará 100% do tempo respondendo a perguntas, e assim poderá desempenhar suas atividades diárias, o único problema é que ele ficará longe da equipe dele de trabalho.

Padrões de codificação: pelo fato de haver muitos profissionais trabalhando juntos num mesmo projeto e o código ser de propriedade coletiva, é importante manter um padrão na forma de programar para que todos entendam melhor e se comuniquem melhor através desses padrões no desenvolvimento do *software*.

2.1.4.3 Scrum

Considerando o desejo de fazer tudo conforme um planejamento inicial e completo, como no modelo de desenvolvimento de *software* em cascata, surge o problema da falta de flexibilidade na aceitação à mudança após o levantamento e definição da arquitetura do *software* a ser desenvolvido, como afirma BECK (1999). O Scrum assume que a fase de desenvolvimento e entrega de módulos do *software* é muito mais complexa e cheia de inserções, de novas características ou mudanças, do que o planejado inicialmente, conforme BACK (1996) a

palavra *scrum* é originada no jogo de *rugby* e significa que a equipe avançará em campo

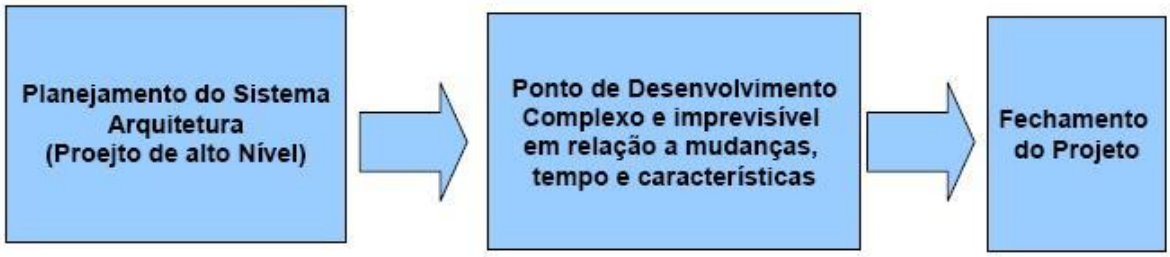


Ilustração 3 - Características Scrum (Back, 1996)

Segundo BACK (1996), o Scrum é um método incremental e interativo de desenvolvimento de *software*. Suas principais características podem ser vistas na ilustração 3 e possuem descrição conforme itens abaixo.

Planejamento: Desenvolvimento de uma lista de requisitos, chamado *Backlog List*, com a definição das datas de entregas e funcionalidades e escolha da *release* mais apropriada para começar o "jogo", mapeamento de pacotes, definição da equipe, avaliação de risco, avaliação e escolha das ferramentas usadas no projeto e estimativas de custos do projeto.

Arquitetura/Projeto de Alto Nível: Revisar o *backlog*, identificar mudanças, análise de domínio para refletir o contexto do sistema e seus requisitos, refinar arquitetura do sistema, identificar problemas na aceitação de mudanças, projetar a forma das reuniões de revisão.

Fase de desenvolvimento: Quando a atividade começa, inicia-se o primeiro Sprint e a equipe caminha *sprint* por *sprint* até finalizar o projeto. O Sprint é um momento de avanço no processo de

desenvolvimento, onde os requisitos são revisados, o planejamento é realizado e a equipe avança até terminar um trecho ou módulo do projeto.

Fechamento do Projeto: Esta é a fase em que o produto de *software* cumpriu as expectativas de requisitos, prazo e custos, e é liberado para uso. Documentos de marketing e material de treinamento são finalizados também nessa fase.

A parte mais importante no processo, conforme apresentado por BACK (1996) e, onde existe menos previsibilidade, é o *Sprint*, a fase de desenvolvimento dos pacotes, das revisões, e inserções de mudanças e novas características ao produto. O *Sprint* é formado por:

Desenvolvimento: Baseado no *backlog*, definições de pacotes, implementações, testes e documentação;

Empacotamento (*wrap*): Fechar o pacote e gerar uma versão executável conforme requisitos definidos no *backlog*;

Revisão: Ponto de revisão, análise de *backlog* e nova avaliação de riscos;

Ajustes: Consolidação das informações coletadas.

Deve-se observar que novos requisitos podem ser acrescidos ao projeto, ao final de um *Sprint*. Durante um *sprint,* nenhuma mudança vinda de fora é aceita segundo RISING (2000).

O Scrum baseia sua estrutura de funcionamento em pontos de controle, como apresentado por BACK (1996):

***Backlog*: Requisitos de funcionalidades do produto**

***Release*: Grupo de requisitos do *backlog* que representam uma *release* viável**

Pacotes: Componentes ou objetos que deverão serão alterados para atender ao *backlog*.

Mudanças: Mudanças que deverão ser realizadas nos componentes.

Problemas: Problemas técnicos que podem ocorrer para implementar um item do *backlog.*

Riscos: Riscos que afetam o sucesso do projeto.

Soluções: Soluções encontradas para solucionar problemas, geram alterações no *backlog*.

Questões: Outras situações que ocorrem no projeto que não afetam pacotes, mudanças e problemas.

2.1.4.4 FDD – Feature-Driven Development

O *Feature-Driven Development* (FDD) é um processo de desenvolvimento orientado por funcionalidades. Auxilia no desenvolvimento e gerenciamento da produção de *software* e possui duas fases distintas e cinco processos bem definidos, conforme HEPTAGON (2007).

Suas fases são:

- **Concepção e Planejamento**
- **Construção**

E possui cinco processos:

DMA (Desenvolver um Modelo Abrangente): esta é a fase inicial, em que é desenvolvida a visão geral do produto, quando a equipe deverá começar a ser formada, a análise de domínio é produzida, assim como o estudo da documentação utilizada, desenvolvimento de modelos

e refinamentos destes. Ao final alguns diagramas como de classe e de seqüência estarão disponíveis.

CFL (Construir a lista de Funcionalidades): neste Processo, a equipe irá desenvolver a lista de funcionalidades do *software*, as funcionalidades listadas são criadas a partir de valores que sejam gerados ao cliente e não valores técnicos.

PPF (Planejar por Funcionalidades): neste as funcionalidades são estudadas, são avaliadas as dependências entre classes, organizadas em seqüência de desenvolvimento e as equipes de desenvolvimento já têm atribuídas suas atividades. O esforço para desenvolvimento já é identificado neste processo.

DPF (Detalhar por Funcionalidades): aqui é realizado o trabalho para definição dos pacotes que atenderão as funcionalidades planejadas. Neste processo também é feito um refinamento dos modelos tanto de classe como de seqüência, além de uma inspeção nos documentos gerados.

CPF (Construir por Funcionalidades): este processo é o de transformar tudo em código, *software*, até gerar o pacote de instalação, passando por inspeções, testes antes de chegar à versão final.

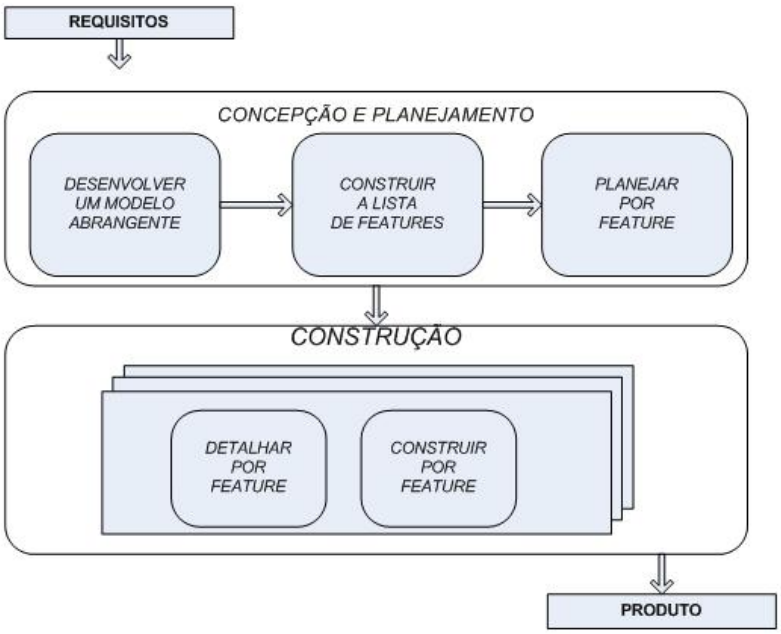


Ilustração 4 - Feature-Driven Development (HEPTAGON, 2007)

É importante observar que o FDD é voltado ao uso de análise orientada a objetos. O modelo gráfico, apresentado na Ilustração 4, apresenta as fases e os processos:

- **Estrutura FDD**
- **Requisitos**
- **Fase de concepção (DMA,CFL,PPF)**
- **Construção (DPF, CPF)**
- **Produto final.**

2.1.4.5 Crystal

O Crystal é uma família de metodologias de desenvolvimento rápido. Ela parte do ponto de que a cada projeto pode existir uma complexidade de trabalho e de organização. A partir desse conceito, existem cores que diferenciam o grau de empenho que a metodologia deverá possuir, relacionado principalmente ao número de pessoas que trabalham no projeto. As Cores são *Clear*, *Yellow*, *Orange*, *Red*, *Magenta*, *Blue*, *Violet* e outras defindas em COCKBURN (2002).

O Crystal Clear é a metodologia mais leve para equipes pequenas, mas as outras (Yellow, Orange etc) são utilizadas para produtos de *software* que necessitam de equipes maiores e mais processos de controle para manter a comunicação.

Existem alguns elementos principais no Crystal e suas características por cor. Sua Filosofia é que o *software* deve ser visto como um jogo cooperativo, com a meta primária de entregar *software* útil e funcionando, e uma meta secundária que é a preparação para a próxima parte do jogo. Essa filosofia leva a duas conseqüências:

- **Projetos diferentes necessitam ser executados de formas diferentes.**
- **A quantidade de modelagens e comunicações que as pessoas precisam para realizar o projeto é exatamente a quantidade que necessitam juntos para fazer o jogo continuar define COCKBURN (2002).**

A família Crystal compartilha: Valores e Princípios

Dois valores:

- **Intrinsecamente centrado em pessoas e comunicação**
- **Altamente tolerante a forma de trabalhar diferenciada de cada equipe.**

Esse dois valores mostram que ferramentas e métodos dão apoio ao ser humano e que a metodologia é tolerante a diferentes culturas humanas.

Existem duas regras que são comuns à família de metodologias Crystal:

- **O projeto deve ser desenvolvido através de incrementos (metodologia incremental)**
- **A equipe necessita de workshops prévios a um incremento e também posterior a esse incremento.**

Duas técnicas base são utilizadas. Uma primeira, para os ajustes no meio do percurso e outra, para verificar o reflexo dos workshops no projeto de acordo com COCKBURN (2002).

Crystal Clear

A metodologia Crystal Clear é utilizada em projetos pequenos, para um número reduzido de pessoas e que, devido à falta de mecanismos de comunicação, devem trabalhar numa mesma sala, para garantir a comunicação.

Os papéis exercidos nessa metodologia são: *Sponsor* (patrocinador), o programador/projetista *Senior*, Programador/projetista e um usuário do sistema por pelo menos uma parte do período.

As políticas padrões são:

- **O *software* é entregue de forma incremental e regularmente;**
- **O progresso é acompanhado através de datas chave consistentes das entregas do *software*;**
- **Deve existir um grupo de testes de regressão automatizados para as funções do *software*;**
- **Deve existir o envolvimento direto de um usuário;**

Há dois pontos de disponibilidades para o usuário durante a *release*:

- **Atividades finais da *release* são realizadas assim que as iniciais estejam estáveis o suficiente para ser revisadas**
- **As reuniões de ajustes na metodologia devem ser realizadas no início e no meio das iterações.**

O Crystal Clear deixa claro que as políticas são obrigatórias, caso seja necessário podem ser substituídas por equivalentes pertencentes aos outros processos de desenvolvimento de *software*.

Neste método, serão produzidos:

- **Seqüência de versões (*releases*)**
- **Agendamento das entregas e disponibilizações ao usuário**
- **Anotações dos casos de usos ou das características do produto de *software***

- **Desenhos de modelagem e anotações quando necessários**
- **Rascunhos de telas**
- **Um modelo de objeto comum**
- **O código funcionando**
- **O código de migração**
- **Os casos de teste**
- **Manual do usuário**

A equipe deve definir e manter:

- **Modelos para os trabalhos produzidos (como os listados acima)**
- **Padrões de codificação e de interface do usuário**
- **Padrões e detalhes de testes de regressão**

Além do compilador, COCKBURN (2002) define as ferramentas mais importantes para a equipe serão um *software* de versionamento e um quadro branco interativo (que imprime).

O Crystal Clear é aberto a alterações e aceitação de outros métodos ágeis, como reuniões diárias em pé, programação em pares e outras.

2.2 Teste de Software

Pode-se responder de várias formas a necessidade de fazer teste de *software*: ao apontar o ser humano pela incapacidade de desenvolver *software* sem defeitos, ou, ao analisar a complexidade em que o *software* está inserido e acabar por apontar essa complexidade. Muitos motivos reais ou imaginários podem ser citados, mas o importante é entender que independentemente do motivo apontado, entre esses citados ou quaisquer outros, deve-se trabalhar com a seguinte premissa: O *software* contém defeitos e encontrá-los é um exercício demorado e custoso.

PRESSMAN apud DEUTSCH (2001) afirma que o desenvolvimento de sistemas de *software* envolve uma série de atividades de produção, e que essas atividades geram grandes oportunidades de injeção de falhas humanas. Ainda mostra que a incapacidade que os seres humanos possuem ao executar tarefas e comunicar-se com qualidade leva ao desenvolvimento de *software* a necessidade de uma atividade de garantia de qualidade.

Aceitando esse fato, pode-se trabalhar de forma mais inteligente e mais direta. MYERS (1979) apresenta a importância de entender a verdade em teste de *software*, quando coloca a importância da análise psicológica na abordagem ao teste de *software* e definição clara de que o responsável pelo teste irá realizar uma atividade destrutiva, procurar por defeitos e, ao encontrá-los terá obtido sucesso. Longe do ponto de vista usual de que um teste de *software* é para mostrar que um programa não possui defeitos, essa abordagem, destrutiva, consciente, faz com que o responsável pelo teste procure efetivamente defeitos obtendo sucesso ao encontrá-los e não enfatizar que deseja provar que o *software* funciona.

Esse ponto de vista, de certa forma, instiga os profissionais de teste a melhorarem suas técnicas pela busca por defeitos, pois o sucesso é encontrá-los.

Quando o assunto é teste, não é possível não trabalhar com prazo e custo, pois os testes mostram defeitos, e defeitos geram retrabalho, afetando os prazos e, conseqüentemente, os custos.

Teste de *software* é a atividade que realiza validação e verificação em partes ou na totalidade do *software* desenvolvido, e tem a função de encontrar defeitos. Um teste bem sucedido é aquele que encontra defeito conforme MYERS (1979).

Uma atividade de teste não pode mostrar a ausência de defeitos, pode apenas mostrar se defeitos de *software* estão presentes.

E através dos princípios de Myers, o foco pode ser direcionado para a busca por defeitos, e não deixar ocorrer o risco de tentar provar a ausência de defeitos. Essa atividade pode ser realizada de várias maneiras, utilizando muitas técnicas diferentes.

Na realização de teste de *software*, separamos as técnicas em dois grupos distintos: teste de caixa branca e teste de caixa preta, que se diferenciam bastante na forma de se organizarem e nas técnicas utilizadas.

O teste de Caixa Preta (teste funcional) recebe esse nome por agrupar técnicas em que o profissional não tem interesse nas estruturas internas, há a preocupação com as entradas e respectivas saídas, deixando o processamento interno sem foco algum, existindo o interesse apenas nos resultados esperados para determinadas entradas. Chama-se caixa preta, por dar a idéia de que o processamento está no escuro, longe das vistas, ou dos interesses. A base para a criação dos casos de testes são os requisitos do sistema. PRESSMAN (2001b) define,

de forma simples, que os testes feitos nas interfaces são os de testes de caixa preta:

O teste de Caixa Branca (teste estrutural), por sua vez, agrupa técnicas em que a estrutura interna do programa é considerada na definição dos requisitos de teste, os casos de testes são desenhados para exercitar trechos de códigos, e devem realizar a maior cobertura do código possível, e assim garantir que defeitos serão encontrados.

Podemos observar que existem diferenças claras entre os dois métodos de trabalho, o teste de caixa preta exercita os requisitos, e aparentemente poderia ser o suficiente, porém o teste de caixa branca ocupa-se dos fluxos e, assim, pode executar testes em fluxos alternativos, dentro do código que os testes de caixa preta não exercitam. Acerca disso, Jones, apresenta razões claras para seu uso:

> **Erros lógicos e pressuposições incorretas são inversamente proporcionais à probabilidade de que um caminho do programa seja executado... Muitas vezes acreditamos que um caminho lógico não tem a probabilidade de ser executado quando, de fato, ele pode ser executado regularmente... Erros tipográficos são aleatórios. (PRESSMAN apud JONES, 2001, p. 786)**

Percebe-se, nesse contexto, que os erros podem aparecer em casos especiais, fora do fluxo principal, ou por não se dar atenção a fluxos alternativos, ou até mesmo erros de digitação podem ocorrer em locais remotos do fluxo principal e cotidiano.

2.3 Formas de realizar Teste

Existem formas diferentes de realizar os testes. Como visto anteriormente, existem os testes de caixa preta (teste funcional), e existe o teste e caixa branca (teste estrutural), algumas técnicas de

detecção de defeitos podem ser vistas no Quadro 4 e alguns tipos de testes podem ser vistos no Quadro 5.

Quadro 4 - Técnicas de detecção de defeitos

Técnica de Caixa Preta (Funcional)	**Particionamento de Equivalência** **Análise de Valor Limite**
Técnica de Caixa Branca (Estrutural)	**Caminho Básico** **Complexidade Ciclomática**

Deve-se observar que o Quadro 4 não apresenta todas as técnicas, mas somente algumas que são apresentadas no decorrer deste trabalho, dentre tantas técnicas estas foram destacadas pela facilidade de exposição para um desenvolvedor de software.

Quadro 5 - Tipos de teste

Teste de Unidade	**Teste realizado em unidades de programação**
Teste de Integração	**Teste realizado ao integrar unidades, para verificar que as unidades estão interagindo corretamente**
Teste de Regressão	**Teste realizado após uma mudança para verificar partes já aprovadas**
Teste de Sistema	**Teste para verificação de todos os requisitos**

Existem também muitos níveis de teste que podem ser desenvolvidos, como teste de unidade, teste de integração, teste de sistema, teste de aceitação e teste de regressão. Esses níveis merecem atenção e são explicitados nas seções seguintes.

2.3.1 Particionamento de Equivalência

O particionamento de equivalência é um teste que não utiliza a estrutura interna do programa para gerar os casos de testes. Este método faz parte do grupo dos testes de caixa preta e sua função é a de

identificar domínios de entrada e diferentes tipos de entrada de dados para gerar os casos de testes apropriados para cada entrada. Os valores de entrada podem ser numéricos, um intervalo de valores, um conjunto relacionado ou uma condição booleana. As classes de equivalência representam valores válidos ou não para os valores de entrada.

2.3.2 Análise de Valor Limite

Esta técnica difere da anterior principalmente devido à observância de que um número grande de defeitos ocorre nos limites dos valores que devem ser aceitos de entrada. Esse método complementa o método de particionamento de equivalência, pois define que os valores de entrada devem também estar no limite, um valor logo abaixo e outro logo acima desse limite esperado.

2.3.3 Caminho Básico

Essa técnica, teste de caminho básico, visa criar casos de testes que garantam que todo o código tenha sido exercitado pelo menos uma vez durante a fase de testes, conforme PRESSMAN (2001b).

2.3.4 Complexidade Ciclomática

A complexidade ciclomática, quando usada no contexto do método do caminho básico, demonstra o número máximo de caminhos independentes existentes. Um caminho independente é qualquer caminho através do programa que leve a, pelo menos, uma nova área do código ou uma nova condição (PRESSMAN, 2001).

2.3.5 Teste de Unidade

O teste de unidade é um processo que realiza a verificação de uma parte menor da construção do software MYERS (1979). Esse teste também é conhecido por teste de módulo.

HUNT (2004) define que o teste de unidade automatizado é um pequeno trecho de código escrito pelo programador que verifica uma área muito pequena e específica do código a ser testado.

Para HUNT (2004), os testes de unidade são feitos para provar que o código faz o que o programador acredita que fará.

2.3.6 Teste de Integração

O teste de integração tem como função verificar que todas as unidades do software irão interagir corretamente com outras unidades, segundo DUSTIN (1999).

CRAIG (2002) mostra que a função do teste de integração é validar, principalmente, as interfaces e que existem níveis diferentes de integração, sendo o nível baixo representado pelo teste de integração realizado pelo desenvolvedor de software e o nível mais alto, aquele teste realizado por uma equipe de teste.

BECK (2004) afirma que no XP os testes são escritos antes de codificar, minuto a minuto. Esses testes são preservados e executados todos juntos, freqüentemente, para fazer a revalidação após mudanças.

No XP, isso é uma forma de garantir que os trechos codificados pelos programadores e, que já foram testados somente

através de seus testes de unidade, não apresentem defeitos ao serem integrados ao produto que já fará parte de uma nova versão e que já passaram por testes de integração. Esse procedimento é realizado a cada nova inserção de uma funcionalidade ao produto a ser entregue.

JEFFRIES (2000) mostra que no XP o software passa por teste de integração quantas vezes forem possíveis, de preferência várias vezes ao dia. Isso recebe o nome de integração contínua.

2.3.7 Teste de Regressão

Segundo PETERS (2001), o objetivo do teste de regressão é o de fazer com que os testes sejam novamente executados sempre que ocorrer mudança em alguma parte do software.

Para CRAIG (2002), o teste de regressão ocorre para verificar se uma correção não injetou um novo defeito. Por isso, o teste de regressão é utilizado depois que as mudanças são feitas e para garantir que o sistema ainda funciona corretamente.

2.3.8 Teste de Sistema

MCCONNELLI (1997) define que um teste de sistema é aquele usado quando é realizada a verificação de que todos os requisitos foram implementados e que foram feitos em um nível de qualidade aceitável.

DUSTIN (1999), por sua vez, indica que o teste de sistema é comumente chamado de teste de caixa preta, isso porque a equipe trabalha principalmente com o que é externo na aplicação.

BEIZER (1995) nos mostra que o teste de sistema nos apresenta comportamentos que não podem ser apresentados por testes de unidade, componentes e integração.

Ao final, PETERS (2001) afirma que o teste de sistema deve garantir que o software esteja plenamente de acordo com os requisitos operacionais correspondentes.

2.4 Teste no Desenvolvimento Ágil

No desenvolvimento ágil, todo código é código testado, afirmação de LEFFINGWELL (2007) que ainda afirma que diferentemente do passado, onde existia uma muralha entre profissionais de desenvolvimento de software e os profissionais de teste, atualmente através do desenvolvedor ágil, o desenvolvedor possui habilidades e executa o teste de software assim que ele é escrito.

LEFFINGWELL (2007) apresenta os princípios do teste ágil:

- **Todo código é código testado;**
- **Os testes são desenvolvidos antes ou concomitantemente à produção de código;**
- **Testar é um esforço de equipe. Desenvolvedores e testadores produzem testes, e**
- **Automação é a regra no desenvolvimento ágil, e não a exceção.**

2.4.1 Extreme Testing

A programação extrema é muito direcionada ao *feedback* e algumas práticas são muito utilizadas para garantir a qualidade do produto a ser entregue. Os testes fornecem o *feedback* necessário para o desenvolvedor, para a equipe como um todo e principalmente para o cliente. Para o XP, realizar testes unitários antes de desenvolver o *software* que será testado, realizar testes de aceitação, realizar constantemente integração, e realizar os testes de regressão mostra o quanto testar é importante.

A importância de testes automatizados é de grande valor, pois sem a automatização dos testes não há possibilidade de fazer com que o XP seja praticado. Como define MYERS (2004), o modelo XP dá grande confiança aos testes de unidade e aos testes de aceitação, esse uso de testes a cada novo código, ou a cada mudança, é chamado de Extreme Testing (XT), ou em língua portuguesa, Testes ao extremo.

Características do *Extreme Testing, conforme* MYERS (2004):

- **O teste de unidade deve ser criado, antes da criação do código a ser testado.**
- **O teste de aceitação deve ser criado, antes da criação do código a ser testado.**
- **É importante o cliente participar na confecção do teste de aceitação.**
- **Os testes unitários devem passar 100%.**
- **Após mudanças, os testes devem ser reexecutados (regressão).**

- **Os testes devem ser, em sua maioria, automatizados.**

MYERS (2004) aponta ganhos com essas características. Uma delas é que os requisitos são melhor entendidos antes da codificação do código final, pois o desenvolvedor tem que pensar como será uma entrada de dados, ou para determinada fórmula se algum campo não foi esquecido na especificação e assim por diante. Além disso, um projeto passa a ter uma melhor construção à medida que o desenvolvedor passa a pensar na solução do código muito antes de chegar a codificá-lo.

CRISPIN (2001) mostra a importância do teste de aceitação ser criado muito cedo na iteração, pois os testes de aceitação mostram o caminho que a equipe de desenvolvimento deve seguir. O conjunto de testes de aceitação é o *Road Map* para a equipe.

Outro fator importante a ser considerado no XP, quando se discute, teste é que o XP produz apenas os testes de unidade e aceitação, estes representam o centro da estratégia XP, como afirma BECK (2004), e a equipe deve analisar quais tipos de teste serão necessários, além dos testes de unidade e dos testes de aceitação.

2.4.2 xUnit framework (Automação)

xUnit é um termo relacionado a qualquer framework de automação de testes de unidade e a maioria deles é orientado a objetos, (MESZAROS ,2007), que indica um conjunto básico de funcionalidades:

- **Cada teste é especificado como um método.**
- **Os valores testados são exercitados através de um método ASSERT.**

- **Agregam vários testes em um único suíte de testes que é executado em apenas uma operação.**

- **O resultado dos testes podem ser apresentados oriundos de um ou vários testes.**

No Quadro 6, podem ser observados vários *frameworks* de teste de unidade e a linguagem de programação que é associada ao xUnit, mas essa lista não esgota todos os *frameworks* disponíveis.

Quadro 6 - xUnit Framework e suas Linguagens (MESZAROS , 2007)

xUnit Framework	Linguagem de Programação
ABAP Unit	ABAP do SAP
CppUnit	C ++
CsUnit	C#
CUnit	C
Junit	Java
NUnit	.NET
PHPUnit	PHP
PyUnit	Python
runit	Ruby
SUnit	Smalltalk
VB Lite Unit	Visual Basic

O primeiro xUnit que se tem notícia é o sUnit, criado por Kent Beck para linguagem SmallTalk que, em seguida, foi portado por Erick Gama para linguagem Java (JUnit). Esse framework foi portado para vários outras linguagens, conforme HAMILL (2004).

2.5 Medições em Software

Medir faz parte do cotidiano, como verificar o preço de produtos, medir em relação ao preço de outro local, ao realizar uma avaliação com o aluno, medir seu grau de conhecimento, verificar o peso de uma criança entre muitas outras coisas.

O que é perceptível, segundo FENTON (1997), é que medir é um processo em que números ou símbolos são atribuídos a atributos de alguma entidade do mundo real, de forma a descrevê-lo de acordo com regras claramente definidas.

FENTON (1997) continua ao definir que o motivo de medir é para auxiliar na escolha, analisar resultados, ter uma forma clara e não tendenciosa de efetuar escolhas ou verificar a qualidade ou valor de uma atividade. No caso dos preços, por exemplo, temos números associados ao atributo valor da entidade produto.

Em PETERS (2001), temos a definição de que a medição de *software* é uma técnica ou método, por meio da qual podem ser aplicadas medidas de *software* a uma classe de objetos de engenharia de *software*, com a finalidade de atingir um objetivo pré-definido. Para isso, é necessário identificar:

- **O objeto que passará pela medição;**
- **Qual o motivo da realização dessa medição;**
- **A fonte da medida e**
- **O contexto da medição**

2.5.1 Medidas de tamanho de Software

FENTON (1997) apresenta três atributos de medidas para tamanho:

- **Comprimento**
- **Funcionalidade**
- **Complexidade**

Ainda FENTON (1997) define que o comprimento mede literalmente o tamanho físico do código, a funcionalidade mede o que é entregue para o cliente e a complexidade pode ser vista de várias formas:

- **A complexidade do problema.**
- **A complexidade do algoritmo.**
- **A complexidade estrutural.**
- **A complexidade cognitiva.**

O tamanho de código é uma forma de medida bastante usual, vejamos algumas considerações:

Representada por LOC, *lines of code*, indica o número de linhas de código de um software. FENTON (1997) apresenta a preocupação em definir como manipular detalhes dessas linhas, como:

- **Linhas em branco.**
- **Linhas comentadas.**
- **Declaração de dados.**
- **Linhas que possuem muitas instruções.**

PRESSMAN (2001) apresenta o uso de LOC, ou KLOC (K indica mil LOC) em uso com outras medidas como:

- **Número de pessoas por mês.**
- **Número de defeitos.**
- **Custo.**

E PRESSMAN (2001) continua com estimativas de produtividade, qualidade e outras da seguinte forma:

- **KLOC / Pessoas mês**
- **defeitos / KLOC**
- **Valor gasto / KLOC**
- **Páginas de documentos / KLOC**
- **Defeitos / Pessoa Mês**

Existem pontos negativos, segundo PRESSMAN (2001), no uso de linhas de código, como a dificuldade de interpretar código estruturado e bem projetado, que podem ser mais curtos, além da diferença entre linguagens.

Outra forma de medir bastante usual é a medida por ponto de função, onde é possível ter uma análise individualizada por função, segundo PRESSMAN (2001).

PRESSMAN (2001) afirma que essa é uma forma indireta, onde o ponto principal é a funcionalidade da aplicação. A mais conhecida é a medida por ponto de função, no qual a complexidade e medidas de informação são utilizadas. Para seu uso, é necessário verificar domínios da informação, como:

- **Número de entradas (input) do usuário;**
- **Número de saídas (output) para o usuários;**
- **Número de consultas do usuário;**
- **Número de arquivos e**
- **Número de interfaces externas.**

Para PRESSMAN (2001), teríamos estimativas de produtividade, qualidade e outras:

- **PF / Pessoa mês**
- **Defeitos / PF**
- **Valor gasto / PF**
- **Páginas de documentos / PF**
- **PF / por pessoa**

As medidas por linha de código ou por ponto de função apresentam indicações para avaliar o software em desenvolvimento e permite realizar cruzamentos de dados com outros projetos.

2.5.2 Métricas C-K Orientada a Objetos

CHIDAMBER (1994) apresenta as métricas C-K orientadas a objetos, que LAING (2001) afirma serem largamente aceitas para sistemas orientados a objetos. Essas seis métricas podem ser vistas a seguir:

- **WMC (Wighted Methods Per Class) é o método ponderado por Classe. O WMC mede a complexidade da classe. CHIDAMBER (1994) afirma que o número de classes e a complexidade dos métodos auxiliam diretamente na previsão de quanto tempo e esforço são necessários para desenvolver e manter uma classe.**
- **DIT (Depth of Inheritance Tree) é a profundidade da árvore de Herança, em classes que possuam múltiplas heranças, o DIT será o tamanho máximo até a raiz da árvore..**

- **NOC (Number of Children) representa o número de sub-classes imediatas subordinada a uma classe na hierarquia de classes.**

- **CBO (Coupling Between Object Classes) representa o número de classes que uma primeira classe está acoplada. Acoplamento elevado pode indicar fraqueza na encapsulamento e pode inibir a reutilização, que ainda define que, quanto maior o acoplamento, mais rigor terá o teste.**

- **RFC (Response for a Class) define o número de métodos que podem potencialmente ser executados através de uma mensagem recebida, que continua, ao afirmar que o nível de dificuldade ao testar e corrigir será mais complicado, quando for alto o número de mensagens, isso por exigir de um testador mais conhecimento sobre o funcionamento das classes.**

- **LCOM (Lack of Cohesion in Methods) representa a falta de coesão entre métodos.**

2.5.3 Métricas In-process

É difícil gerenciar uma atividade de teste de *software* sem o uso de métricas de *software* CRAIG (2002)

KAN (2001) mostra que o acompanhamento de métricas *in-process* (métricas utilizadas durante o desenvolvimento de um produto) desempenha um papel importante no desenvolvimento de software e principalmente para a atividade de teste.

KAN (2001) ainda afirma que existem poucos trabalhos sobre métricas *in-process*, e poucas métricas *in-process* são descritas com experiências de sua implantação e de sua utilidade.

São apresentadas as seguinte métricas *in-process* por KAN (2001):

- ***Test progress S Curve* (planejado, tentativa , atual)**
- ***PTR (defects)***
- ***PTR(test defect) backlog over time.***
- ***Product / release size over time.***
- ***CPU utilization during test***
- ***System crashes and unplanned IPLs***
- ***Mean time to IPL***

***Test progress S Curve* (Curva S do progresso de teste), é uma medida que faz o acompanhamento de casos de testes planejados de serem feitos, os casos de testes que foram utilizados (ocorreu a tentativa de busca por defeitos) e o número de casos de testes que obtiveram sucesso. O motivo de se utilizar esta medida é garantir que, em atrasos na equipe de desenvolvimento, os testes não sejam negligenciados para que se possam cumprir os prazos e o ponto crítico é o planejamento, conforme KAN (2001).**

PTR (defeitos), relatório de acompanhamento de problemas, auxilia no acompanhamento de número de defeitos (problemas) acompanhados durante as fases de testes, e é altamente recomendada por KAN (2001), embora a densidade de defeitos seja uma unidade de resumo e não realmente uma unidade *in-process*, KAN (2001) afirma que o padrão de chegado dos defeitos através do tempo resulta em informação.

PTR (*test defect*) *backlog over time* (os problemas aberto através do tempo), apresenta os defeitos que continuam em aberto ao decorrer do tempo, é simplesmente a diferença entre os defeitos reportados e os que já foram resolvidos.

Product / release size over time (tamanho de produto ou realease através do tempo) mostra o esforço de desenvolvimento, e pode ser relacionado aos defeitos que chegam e aos defeitos em aberto.

KAN (2001) ainda apresenta duas unidades de medida para situações criticas: *CPU utilization during test* e *System crashes and unplanned IPLs. T*emos, no primeiro, um acompanhamento do processamento da CPU e, no segundo, um acompanhamento das falhas que geram paradas ou reinicialização do sistema (*reboot*). Além destas medidas, uma medida de confiabilidade é utilizada, que é o caso da *Mean time to IPL*, ou tempo médio para a ocorrência de um IPL. Segundo KAN(2001), o IPL é o mais severa das falhas apresentadas por sistemas.

As medidas de defeitos *in-process* não se resumem aos aqui apresentados, na próxima sessão, será tratada as métricas STREW de medidas *in-process*.

2.5.4 Métricas STREW

O STREW (*Software Testing and Reliability Early Warning*) é um conjunto de métricas que coletam dados durante o processo de desenvolvimento do software, onde é obrigatório o uso de uma *framework* xUnit de teste de unidade.

Essa métrica surgiu dos trabalhos de NAGAPPAN (2004) e NAGAPPAN (2005), que buscam o uso de ferramentas xUnit para coleta de dos de teste e a busca antecipada por resultados de confiabilidade.

NAGAPPAN (2004) apresenta uma versão inicial do STREW com cinco métricas, o STREW por ser uma versão orientada a objetos, recebe a denominação inicial de STREW-OO, mas após vários estudos o STREW avança para nove métricas diferentes e, neste último trabalho, voltado para Java, passando a STREW-J. SHERRIFF (2005) também utiliza a métrica STREW e faz uma adaptação a linguagem Haskel e utiliza um *framework* de testes chamado HUnit, as métricas, nesse caso, recebem o nome de STREW-H. Com isso, percebe-se que o STREW possui uma dependência com algum *framework* xUnit, e também mostra que o STREW pode ser aplicado em projetos com linguagens de programação diversas, como pode ser visto no Quadro 7.

É importante observar que existem mais *frameworks* xUnit, para outras linguagens, e também salientar que a natureza *open source* não é obrigatória em todos os *frameworks* xUnit.

Como referência básica, a versão do STREW descrita será a de NAGAPPAN (2005b), com suas nove métricas, conforme Quadro 7.

Quadro 7 - Métricas do STREW (NAGAPPAN, 2005b)

Nº	Métrica	Descrição
SM 1	Número de ASSERTs SLOC	Conta o número de *ASSERTs* em todos os casos de teste para todo o código fonte.
SM 2	Número de Casos de Teste SLOC	Conta o número de Casos de teste criados para todo o código fonte.
SM 3	Número de ASSERTs Número de Casos de Teste	Conta o número de *ASSERTs* em todos os casos de testes para todo o código fonte. Conta o número de casos de teste para testar todo o código fonte.
SM 4	(TLOC + SLOC) (Nº classes de Teste Nº classes fonte)	Conta o número de classe por todos os códigos fonte Conta a número de classes por todos os arquivos de teste.
SM 5	∑ Complexidade cinclomática teste ∑ Complexidade cinclomática fonte	Calcula a soma da complexidade ciclomática para todos os arquivos de testes

		Calcula a soma da complexidade ciclomática para todo o código fonte
SM 6	**∑ CBO teste / ∑ CBO fonte**	**Calcula a coma de todos os CBOs de todos os arquivos de testes Calcula a coma de todos os CBOs de todo o código fonte**
SM 7	**∑ DIT teste / ∑ DIT fonte**	**Calcula a soma do DIT de cada arquivo para todos os arquivos de teste. Calcula a soma do DIT de cada arquivo para todo o código fonte.**
SM 8	**∑ WMC teste / ∑ WMC fonte**	**Soma todos os WMCs e cada arquivo para todos os arquivos de teste. Soma todos os WMCs e cada arquivo para todo o código fonte.**
SM 9	**SLOC / SLOC Mínimo**	**Divide o SLOC pelo SLOC do menor projeto usado para construir o Strew.**

As métricas (*Strew Metrics*) SM1, SM2, SM3 e SM4 fazem referência cruzada entre elas para contar estilos de codificação e de realização de teste, exemplo: um desenvolvedor coloca cinco *ASSERTs* num mesmo caso de teste, enquanto outro cria um caso de teste para cada *ASSERT*. Essas métricas são utilizadas para que não seja necessário o STREW indicar formas de como programar e dar mais liberdade ao desenvolvedor do software, segundo NAGAPPAN (2005b).

As métricas SM5, SM6, SM7 e SM8 trabalham taxas relacionadas ao controle de fluxo de complexidade e para um subconjunto das métricas CK.

A métrica SM9 é uma métrica de fator de ajuste relativo de tamanho.

2.5.5 Metodologia GQM

Novos processos para melhoria de qualidade necessitam de alguma ferramenta que dê suporte na avaliação de seus resultados. A forma como os dados são levantamento na pesquisa, direcioná-la ou não para o sucesso, uma escolha incorreta de métricas pode mascarar resultados e inviabilizar a pesquisa.

O Goal Question Metric é uma ferramenta que permite organizar de forma clara o procedimento para uma melhor escolha das métricas envolvidas no processo. É baseado em metas e largamente utilizado em projetos de engenharia de software.

O GQM, método proposto por Vitor Basili, é baseado no paradigma de medição orientada a metas, conforme FONTOURA (2004), ABIB (1999).

Ilustração 5 - Estágios do GQM (ABIB, 1999)

A Ilustração 5 mostra três partes estágios do GQM: Desenvolvimento do plano GQM, Execução do plano de medições, empacotamento da experiência.

No Plano de desenvolvimento, temos as fases de pré-estudo, elaboração do planejamento GQM e elaboração do planejamento de métricas.

Na Execução do Plano de Medições, temos a coleta de dados e a análise e integração dos dados.

O estágio de empacotamento da experiência prepara um documento final e possui a fase de construção da base de conhecimento que a experiência proporciona.

O estágio mais importante, para o sucesso do GQM, é o estágio de desenvolvimento do GQM.

Descrição das atividades:

1º. Pré-estudo: Nesta atividade, segundo FUGETTA (1998), devem ser coletadas as informações do contexto em que a atividade de mensuração será realizada.

2º. Identificação das Metas GQM: Baseado na descrição do contexto, as metas devem ser definidas, conforme FUGETTA (1998).

3º. Definição do Plano GQM: As metas, questões e métricas são definidas nesta fase, conforme ABIB (1999).

4º. Definição do plano de Mensuração GQM: É quando definimos quais informações coletar , quando e como, conforme FUGETTA (1998), ABIB (1999) define que, nesta fase, é que serão definidas as estratégias e técnicas que farão parte do processo de implementação das mensurações.

5º. Coleta dos dados: nesta fase, os dados são coletados e validados, conforme FUGETTA (1998) e ABIB (1999).

6º. Análise de Dados: Os dados coletados são analisados e interpretados, conforme FUGETTA (1998) e ABIB (1999).

7º. Empacotamento da Experiência: Os dados da experiência são armazenados para futuro, conforme FUGETTA (1998). ABIB (1999) define que essa fase ainda é dividida em uma parte para preparação do documento final e outra para a construção da base de conhecimento sobre o experimento realizado.

Em nosso projeto, iremos realizar a abordagem sugerida por FUGETTA (1998) e ABIB (1999). Cada uma das mudanças sugeridas (ADQn) terá um planejamento organizado, conforme o GQM, para realização de um experimento e avaliação dos resultados.

Para este trabalho, o GQM será utilizado e sua forma adaptada é explicada abaixo:

Um formulário, chamado Abstraction Sheet, baseado em ABIB (1999) será criado, conforme pode ser visto na Ilustração 6.

Meta	Objeto	Propósito	Foco de Qualidade	Ponto de vista	Ambiente
Foco de Qualidade *Descrever o foco de qualidade do processo de medição*			**Fatores de Variação** *Quais são os fatores que podem causar impacto ao foco de qualidade*		
Hipótese de *Baseline* *Qual é a estimativa do estado corrente de acordo com o foco de Qualidade*			**Impacto no *Baseline* de Hipótese** *Como os fatores de variação influenciam no foco de qualidade*		

Ilustração 6 - Uma Abstraction Sheet Geral (Abib, 1999)

.

Descrição do procedimento neste trabalho:

1º. Definir a Meta, baseado em KIRNER (1997), conforme Ilustração 7.

Objeto *(O que será analisado?)*	*Análise de* _
Propósito *(Por que o objeto será analisado?)*	*Com o propósito de* _
Foco de Qualidade *(qual a propriedade que será analisada?)*	*Em relação a* _
Ponto de Vista de *(Quem irá utilizar os dados coletados?)*	*No ponto de vista do pesquisador e proprietário da empresa.*
Ambiente *(Qual o ambiente que a atividade será realizada?)*	*Ambiente empresarial.*

Ilustração 7 - Formulário de uma Meta (Kirner, 1997)

2o. Definir *Quality focus*, *Variation Factors*, B*aseline Hypotesis* e *impacts on Baseline hypotesis*, conforme Ilustração 8.

Definição do Foco na Qualidade para montagem do Abstraction Sheet.	_ _
Definição dos Fatores de Variação para montagem do Abstraction Sheet.	_ _
Definição das Hypoteses de Baseline para montagem do Abstraction Sheet.	_ _
Definir o Impacto nas Hipóteses de Baseline para montagem do Abstraction Sheet.	_ _

Ilustração 8 - Quadrantes da Abstraction Sheet (Kirner, 1997)

3º. Revisar *Abstraction Sheet* com participantes do projeto, e fechamento do modelo que será tratado no projeto da abstraction sheet completa, como pode ser visto na Ilustração 6 .

4º. Reunião para definição de GQM:

a) **Definir Questões, (aqui não se deve procurar as métricas, mas sim pensar em como atingir a meta, através da resposta a uma questão), um formulário será usado para coletar essas questões, conforme Ilustração 9.**

Q1	_ _
Q2	_ _
Q3	_ _
Q4	_ _
Qn	_ _

Ilustração 9 - Formulário para Questões GQM

b) **Definir Métricas que responderão as questões, o formulário representado na Ilustração 10 terá a questão definida no passo anterior, e as possíveis métricas relacionas serão definidas.**

Q1 – Descrição da Questão 1
*M1:*_ _
*M2:*_ _
*M3:*_ _
*Mn:*_ _

Ilustração 10 - Formulário para descrição de métricas / Questão GQM

5º. Definir o plano de mensuração.

a) **Como os dados serão coletados. Deverão ser descritos por métrica quais serão os dados coletados, com uso do formulário apresentado na Ilustração 11.**

Q1 – Descrição da Questão 1
M1 – Descrição da métrica 1
Descrição do método de coleta: ______________________________________
Como: ______________________________________
Quando: ______________________________________

Ilustração 11 - Formulário de definição de coleta de dados

b) Como e quando os dados serão analisados, também apresentados na Ilustração 11.

9º. Fechamento (empacotamento) do experimento.

a) Preparar documento final;

b) Workshop com empresa onde ocorreu experimento;

c) Fechamento do documento de experiências.

3 METODOLOGIA

3.1 Modelo, Métodos e Técnicas

Este trabalho visa atuar com valores, princípios e técnicas ágeis no desenvolvimento de software, através dos pequenos modelos definidos por ADQs, propondo usos, formas de controle e inserção de novos aspectos no processo de desenvolvimento de software da pequena empresa produtora de software.

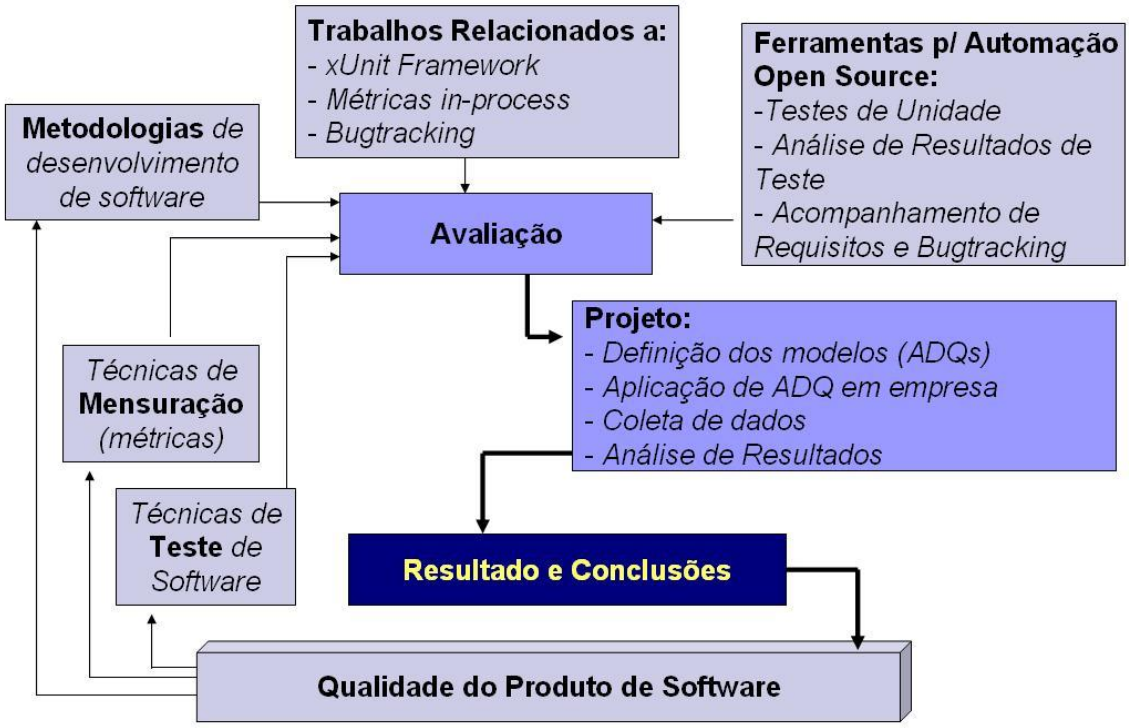


Ilustração 12 – Contextualização do trabalho

O projeto a princípio seria realizado com foco em uma pequena empresa, essa alternativa não obteve sucesso e foi abandonada no momento em que a empresa parceira passou por mudança na situação societária e reorganizou sua estrutura para novos projetos. Como alternativa a essa situação o projeto foi reorganizado para atender pequenas inserções no processo de desenvolvimento de software em algumas áreas distintas de empresas produtoras de software de forma

independente e ficou organizado em alguns ADQ, ou melhor, Adicionais de Qualidade, onde o foco foi fazer com que os modelos ADQs tivessem em sua estrutura valores e princípios de metodologias ágeis, práticas de metodologias ágeis e que fossem um apoio para a pequena empresa produtora de software organizar os processos de qualidade da empresa. O projeto trata modelos para áreas distintas do desenvolvimento de software e podem ser adaptados aos processos existentes, quando estes existirem. A figura Ilustração 12 apresenta a contextualização para realização dos trabalhos. ADQs devem:

- **Ser pequenos modelos de apoio na melhoria de qualidade de software;**
- **Ter um foco em Valores e Princípios Ágeis de desenvolvimento de software;**
- **Fazer uso de técnicas ou ferramentas utilizadas por equipes que seguem metodologias ágeis (exemplo: TDD);**
- **Foco em pequenas mudanças, mas que gere números para avaliação de resultados;**
- **Ser rápido no retorno de resultado e percepções sobre resultados de mudanças nas empresas pequenas;**
- **Para pequenas empresas produtoras de software que estejam ainda na fase caótica de desenvolvimento de software, os ADQs devem Introduzir nos conhecimentos de processos de desenvolvimento de software, no processo de controle através de métricas e nos novos conhecimentos relacionados aos testes;**

- **Um ADQ deve ser pequeno, mas deve intervir com resultados na empresa e ser factível;**

Adicionais de Qualidade:

ADQ1: atua na mudança nos requisitos, busca tornar o processo mais comunicativo entre clientes e desenvolvedores. Acaba por explicitar que o risco pelo não entendimento será assumido pela gerência e não pelo desenvolvedor como podemos ver nas pequenas empresas produtoras de software.

ADQ2: mudanças na fase desenvolvimento de software ao adicionar ao desenvolvedor a tarefa de testar melhor o software, isso é realizado através do uso testes de unidade automatizados com uso de framework xUnit, além da realização de treinamento dos conceitos básicos de teste e desenvolvimento adequado de testes de Unidade no framework xUnit adotado pela empresa.

ADQ3: segundo nível de mudança no processo de desenvolvimento de software, apresentando uma proposta de mudança para o TDD, que é o coração da metodologia ágil Extreme Programming por apresentar vantagens na qualidade do software gerado.

O controle dos resultados (ADQ-1,2,3,4 e 5) será realizado através de métricas, dando preferência para as de fácil compreensão como o uso de linhas de código, métricas in-process e controle de tempos nos estágios de desenvolvimento (horas planejadas, horas executadas, horas de retrabalho por defeito e horas de retrabalho por mudanças de requisitos).

Esses modelos, tratados neste documento por Adicionais de Qualidade, ou ADQ, existem para atender pequenas empresas produtoras de software e são organizadas aqui de forma separada, para

permitir seu uso individualmente e facilitar o processo de uso de cada uma, tendo em vista as melhorias individuais disponibilizadas.

O projeto prevê uma melhora significativa para as empresas em seus resultados, após a implantação de um dos modelos propostos. Os ADQs não visam sobrecarregar às equipes, por isso são individualizados. Além disso, o processo é concebido de forma que a empresa possa realizar a sua implantação, adaptando o ADQ a sua realidade.

Outros estudos estão sendo realizados para ampliação dos ADQs, como :

ADQ4: automação em outras fases de teste, para tornar o teste ágil e viável, através de testes de integração e regressão apoiados no framework xUnit.

ADQ6: controle nos defeitos reportados, criando um processo de interfaceamento entre os defeitos e os requisitos, através de ferramentas open source, dando a possibilidade de cruzamentos entre módulos problemáticos e requisitos.

ADQ6: controle in-process da qualidade de software, onde métricas *in-process* utilizadas por KAN (2001) podem auxiliar no acompanhamento durante o desenvolvimento ou o uso de métricas STREW-x, neste último é pré-requisito o ADQ2.

3.1.1 Pré-Requisitos Gerais

Para que seja possível utilizar os modelos apresentados neste documento, é obrigatório, para avaliar a melhora, fazer um levantamento da história da empresa em relação aos produtos de

software já desenvolvidos. Esse estudo visa criar uma linha base para verificação dos resultados após a implantação da melhoria.

Para auxiliar nesse processo, uma aplicação foi desenvolvida para gerar números iniciais, como o número de linhas de código, armazenamento e controle desses números, assim como um questionário para levantar outros dados e criar um relatório de como a empresa estava antes da mudança do processo.

Os dados mínimos que deveriam levantados para posterior comparação são os seguintes (por projeto):

- **Número de linhas de código;**
- **Esforço de desenvolvimento em horas**
  - **de Projeto;**
  - **da Realização;**
  - **de Retrabalho**
- **Número de defeitos reportados pelo cliente nos primeiros dois meses após implantação de projeto.**
- **Informações sobre os desenvolvedores**
  - **Idade;**
  - **Tempo de Experiência em Desenvolvimento;**
  - **Tempo de Experiência com a LP;**
  - **Horas trabalhada no projeto;**
  - **Horas de retrabalho de seus módulos;**
- **Percentual de Margem de lucro planejada;**

- **Percentual de Margem de lucro real**

Os indicadores acima foram escolhidos por serem de fácil compreensão e que podem gerar respostas rápidas para os responsáveis pelo gerenciamento dos projetos, mas as possibilidades podem aumentar dependendo dos resultados gerados nas reuniões GQM e seu respectivo fechamento, além de permitir ao pequeno produtor de software perceber os resultados em relação ao passado de sua empresa.

3.1.2 ADQ-1 - Requisitos

O desenvolvimento ágil afirma que o cliente deve participar do projeto de forma ativa, como pode ser visto em BECK (2001). No método proposto, os testes de aceitação devem ser criados pelo cliente ou na presença do cliente.

Nesta mudança, uma adequação ao processo de controle de requisitos é necessária, sem realizar profundas transformações aos procedimentos existentes.

Será necessário que os requisitos sejam armazenados e recebam um código (número) para controle. O modelo apresentado na Ilustração 13 apresenta como o processo da empresa será alterado.

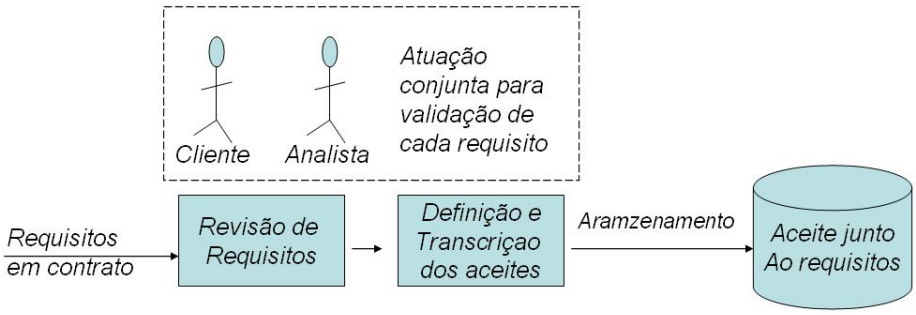


Ilustração 13 - Levantamento Aceite

A Ilustração 13 nos apresenta que um passo de validação de requisitos é acrescentado ao processo existente: o cliente estará junto nesse processo e irá montar, ao lado do analista, ou gerente o aceite de cada um dos requisitos. Após essa definição, os aceites serão gravados no mesmo documento de requisito para melhorar a comunicação com o desenvolvedor, ampliando assim a facilidade de o desenvolvedor entender o que o cliente solicita.

Os testes de aceitação criados devem possuir um formato simples de ser trabalhado, como o proposto por CRISPIN (2001), que sugere a criação através da definição de:

- **Ação**
- **Dados**
- **Resultados Esperados**

A ação indica o que o usuário realizará. Os dados são aqueles que serão utilizados para teste da ação definida, e o resultado esperado mostra o que o cliente espera dessa ação em conjunto com o uso dos dados utilizados. Esse resultado pode ser o sucesso, ou insucesso, ou uma mensagem na tela, ou um resultado no banco de dados, como pode ser visto na Ilustração 14.

Ação	Dados	Resultado Esperado
Login	*usuário em branco* *senha em branco*	*Apresentar mensagem:* *"Usuário ou senha inválidos"*
Login	*usuário: JSilva* *senha: em branco*	*Apresentar mensagem:* *"Usuário ou senha inválidos"*
Logi	*usuário: JSilva* *enha: !2#4%*	*Entrar na aplicação*

Ilustração 14 - Trecho de teste com ação/dados/resultado esperado

Nesse processo, temos ainda a inserção de uma nova fase, que tem a finalidade de manter a comunicação com o cliente, mesmo que este não esteja presente.

O desenvolvedor comumente recebe o requisito e inicia seu trabalho de codificação. Existem momentos em que o desenvolvedor não entende o que é realizado, ou tem uma compreensão pequena. Há casos em que o prazo pode inibir o desenvolvedor a expressar dúvidas e inclusive falta de espaço para dar alguma opinião sobre o requisito, este desenvolvedor inicia seu trabalho, mesmo com dúvidas e criando uma resposta para essas dúvidas sob a percepção própria.

Para evitar que o risco seja assumido pelo desenvolvedor, uma alteração na fase de recebimento do requisito se faz necessária.

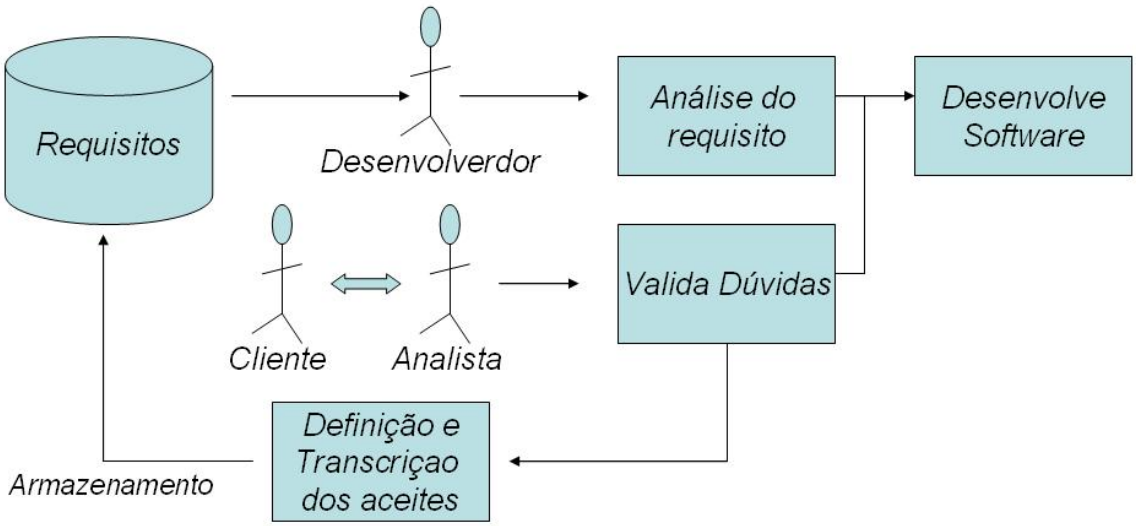


Ilustração 15 - Requisitos e o desenvolvedor

A Ilustração 15 define como o modelo deverá alterar o processo de recebimento do requisito pelo desenvolvedor. Ele passa a ter um momento de entendimento e validação, e de acesso ao Analista para tirar suas dúvidas e iniciar o processo de desenvolvimento. Caso o analista não consiga resolver a dúvida, ele deve procurar resolver com o

cliente e realizar as devidas mudanças no requisito e aceite. Aqui podemos perceber que deve existir coragem para não fazer algo com baixo entendimento e o *feedback* para o cliente e para o desenvolvedor é aumentado, pois ambos participam mais no processo que o habitual.

É importante que fique claro que o processo deve ter a participação do cliente no momento de tirar as dúvidas sobre o que fazer em um problema de entendimento, ou falta de mais dados sobre o requisito. Em caso da impossibilidade de acesso ao cliente, o risco deve ser estudado pelo analista de sistemas (ou gerente) e a melhor abordagem deve ser definida por ele, isso faz com que o risco seja de responsabilidade do analista de sistemas e não mais do programador, deixando o programador com foco apenas no código.

As alterações sugeridas irão gerar melhora significativa no entendimento dos requisitos, conseqüentemente ocorrerá uma menor injeção de defeitos por parte do desenvolvedor, isso por acrescentar maior quantidade de informação útil ao desenvolvedor e criar uma estrutura com a possibilidade de minimizar dúvidas sobre requisitos.

Outros benefícios podem ser tirados desse processo, através da melhoria na comunicação com o cliente, situações não descritas pelo cliente podem surgir e podem fazer parte do projeto permitindo a realização de um produto mais adequada ao cliente, e também permitindo um maior poder de negociação, antes de desenvolver o requisito e não depois que ele já esteja pronto, mas inadequado ao cliente.

3.1.3 ADQ-2 Uso de xUnit

Conforme visto anteriormente, a função de um framework xUnit é permitir que sejam desenvolvidos casos de testes de unidade através de codificação. Normalmente, o framework xUnit é desenvolvido e

criado na mesma linguagem de programação que o software que está sendo desenvolvido.

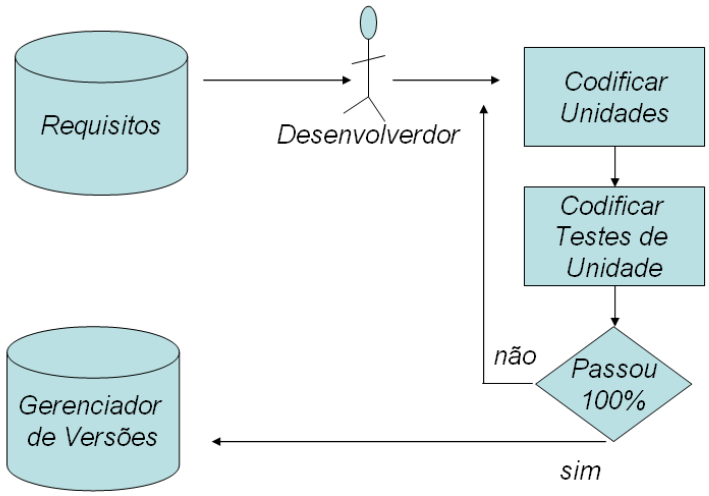


Ilustração 16 - xUnit

Neste processo, o desenvolvedor será treinado no uso do framework xUnit:

1o. Iniciar a codificação;

2o. Após ter uma primeira unidade testável, desenvolver o código de teste;

3o. Executar o Código de teste

Neste modelo, o desenvolvedor deve receber treinamento para uso do xUnit, além disso, recebe treinamento específico sobre técnicas de teste, pois isso fará com que o desenvolvedor entenda a real necessidade de se buscar defeitos nos testes e incentivará a busca pela qualidade contínua.

3.1.4 ADQ-3 TDD

Esta parte do modelo não implementa nenhuma nova ferramenta, mas tem como pré-requisito o uso de uma ferramenta de xUnit.

O *Test-Driven Development* é uma forma diferente de o desenvolvedor trabalhar. Ele deixa de pensar, em primeiro lugar, no código e passa a pensar em como realizar o teste desse código.

A mudança do procedimento do desenvolvedor deverá ser a seguinte:

1º. Deverá codificar o teste de unidade que irá testar a funcionalidade a ser desenvolvida;

2º. Deverá executar o teste de unidade, e este deverá mostrar que não passou, pois se um teste de unidade passar antes de sua codificação da unidade significa que esse teste não foi desenvolvido corretamente;

3º. Codificar a Funcionalidade;

4º. Executar o teste.

Observar que, no caso de teste encontrar defeitos, o desenvolvedor deverá corrigir os defeitos e voltar ao teste, até que este tenha 100% de aprovação.

A Ilustração 17 apresenta o modelo e mostra a necessidade de coletar informações sobre o processo de testes. Essa necessidade pode auxiliar a encontrar módulos mais suscetíveis a defeitos, bem como definir, em fases futuras de testes, a quantidade de esforço que determinados módulos deverão possuir.

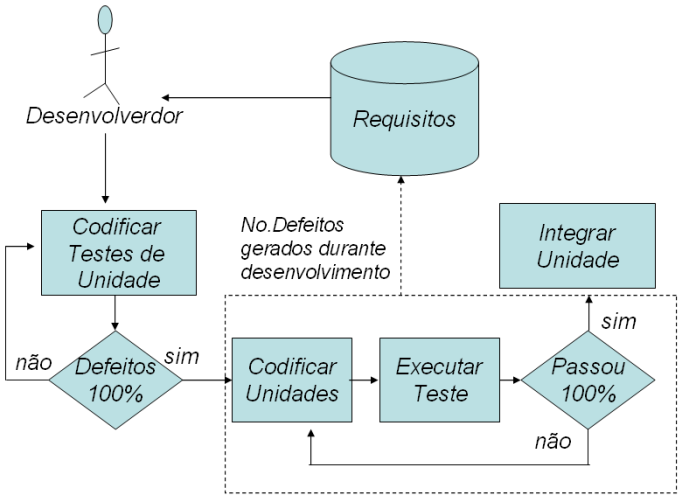


Ilustração 17 – TDD (Test-Driven Development)

Para isso, é necessário apenas coletar os resultados e armazená-los em Banco de Dados. Além disso, é importante fazer uma ligação com o código que se relaciona ao requisito.

Após a coleta desses dados, uma tabela com requisitos e defeitos deve ser realizada, na qual o cruzamento com o número de linhas passa a ser importante, para comparação com o tamanho e número de defeitos (densidade de defeitos por requisito). Este processo já auxilia na previsão dos módulos mais suscetíveis a defeitos.

3.2 Trabalhos Correlatos

Trabalhos similares, com apoio de ferramentas e com o foco na agilidade no processo de testes, não foram encontrados. Um processo para implantação, ou melhoria, de processo de testes em empresas, foi desenvolvido pelo CenPRA, mas essa metodologia não é direcionada para um trabalho ágil, mas sim direciona para o padrão de documentação do padrão definido no IEEE em 1998, conforme definido em CRESPO (2005).

Outros dois trabalhos, que não são similares, mas dão suporte à escolha da técnica de Test-Driven Development podem ser vistos a seguir:

- **O trabalho de WILLIAMS (2003) é um dos trabalhos relacionados ao TDD, o resultado do uso de TDD realizado na IBM apresenta 40% menos defeitos nos softwares.**

- **Outro trabalho com resultado importante é o apresentado por BHAT (2006), realizado na Microsoft e o resultado apresentado mostra que a melhora é significativa na qualidade do código, segundo métricas da empresa, é duas vezes maior a qualidade dos produtos gerados através do TDD.**

3.3 Proposta de Trabalho

A proposta deste trabalho é o de realizar medições iniciais nas empresas participantes, criar um perfil da empresa, em seguida, aplicar um dos ADQs e avaliar o resultado.

O GQM fará parte do processo para auxiliar na escolha e definição das métricas com o auxílio e percepção dos proprietários das empresas e funcionários relacionados ao processo.

Ao final de cada uma das atividades, o resultado esperado será o de melhoria para a empresa, e é esperado que essa melhoria motive as empresas a continuarem em mudanças em função da melhoria da qualidade.

3.4 Planejamento padrão para todas as empresas

No projeto, tivemos a participação inicial de três empresas, que serão identificadas apenas pelas letras A, B e C.

Neste documento, teremos o detalhamento para as ocorrências e resultados de cada empresa, mas todas elas inicialmente deveriam obedecer à seguinte programação definida previamente nestas fases:

1º. Fase: esclarecimento do projeto e coleta inicial de dados.

Nesta fase, o objetivo é o de fazer com que os participantes, funcionários das empresas, tenham uma visão macro do que seria executado, as atividades, aprendizagens e tudo o mais necessário para a condução do projeto. Ficou definido que existiria no mínimo:

- **Um Workshop com a equipe apresentando objetivos e novos conhecimentos envolvidos no projeto;**
- **Treinamento necessário para condução do projeto;**
- **Levantamento de dados históricos da empresa.**

Temas que seriam abordados nesta fase:

- **Qualidade de software;**
- **Influência da qualidade nos resultados para a empresa e para o funcionário;**
- **Importância da diminuição da quantidade de defeitos produzidos;**

- **Entendimento do modelo aplicado na empresa para execução de mudanças voltadas a melhoria da qualidade do software produzido.**

2º. Fase GQM.

Nesta fase, o objetivo é o de apresentar para a área gerencial da empresa a metodologia que será aplicada, definição clara do usuário chave, que será aquele que irá atuar diretamente no processo de mudança e condução do GQM para auxiliar no sucesso do planejamento do projeto, com base nisso, há as seguintes atividades:

- **Apresentação do que é GQM e sua aplicação.**
- **Apresentação e desenvolvimento da Fase de definição GQM e uso de reuniões curtas para definições iniciais, como:**
  - **Reunião para definição de metas.**
  - **Reunião para definição de questões.**
  - **Reunião para definição de métricas necessárias.**
- **Reuniões com o usuário chave para uma pré-preparação do planejamento GQM para debate e fechamento e fechamento posterior com outros funcionários de nível gerencial;**
- **Reunião de fechamento do Planejamento GQM**

Desta fase, é esperado que as empresas já possuam um planejamento de metas e métricas a serem analisadas após a implantação do ADQn definido para a empresa. Ainda, nesta fase, um cronograma-base deverá ser fechado para garantir a execução do Plano GQM.

3º. Fase de execução GQM.

Esta fase deveria seguir o que foi planejado na fase anterior. Para esta fase, as seguintes atividades foram programadas:

- **Execução das atividades;**
- **Acompanhamento;**
- **Documentação.**

Ao final desta fase, o esperado é que existam informações para análise de seus resultados, e que seja possível a revisão nas atividades programadas previamente.

4º. Fase: Análise e conclusão dos Resultados

Nesta fase, os dados coletados serão avaliados e seus resultados apresentados aos responsáveis em cada uma das empresas participantes.

Desta fase, saem os dados que darão suporte ou não à validade das mudanças realizadas em cada empresa, apresentando dados que possam ser confrontados com os dados históricos e que possam auxiliar na verificação do sucesso ou não do uso dos ADQs utilizados em cada empresa.

3.5 Organização dos Projetos em cada Empresa

Temos aqui as definições e resultados de cada empresa participante no projeto. Observa-se que não foram as três empresas que seguiram o processo até o final, mas isso também pode ser utilizado para lições futuras.

A identificação das empresas não será feita neste documento, as empresas serão tratadas por: Empresa A, Empresa B e Empresa C.

3.6 Empresa A (São Roque)

A empresa localiza-se próximo a cidade de São Paulo, na cidade de São Roque. Possui muitos funcionários, mas para a área de desenvolvimento possuía apenas 7, contando o gerente de área. A equipe era formada por programadores com pouca experiência, o único programador sênior era um freelancer que recebia as demandas consideradas mais críticas.

A condução do projeto, nesta empresa, sofreu vários problemas, pois ocorreu mudança de sócios durante o processo e, nesse período, ocorreu pausa. Mais tarde, quando reiniciadas as atividades, o usuário chave deixou a empresa.

Ao final, nesta empresa, não ocorreu nada além da execução de treinamentos de testes, algumas reuniões para definição de usuário chave, poucas reuniões sobre o entendimento do GQM, como seria a estruturação do processo.

Dados históricos foram levantados, mas seu uso ficou de lado, pois a principal função era traçar um paralelo com os dados após uso do modelo ADQ1 que, em visita, à empresa, ficou claro que iria não só melhorar a comunicação entre o desenvolvedor e o cliente, e dar suporte para a melhora esperada, como iria praticamente estruturar esse procedimento, já que a empresa tinha no vendedor o gerador de requisitos e o contato com o cliente só era realizado pelo gerente em situações críticas. Essa empresa por estar sempre em atraso com seus projetos, preferia, em reuniões, levar código em funcionamento, mas

normalmente esse código apresentava problemas de interpretação em relação aos desejos do cliente. Em meio aos problemas de rotativdade de pessoal, mudanças estruturais na direção da empresa entre outros, o projeto passou a ser considerado como uma oportunidade futura e não mais algo para aquele momento. Até o fechamento, deste projeto, o assunto não tinha voltado em pauta nas reuniões da empresa.

3.7 Empresa B (São Paulo / ZN)

3.7.1 Caracterização da Empresa.

Empresa localizada na Zona Norte de São Paulo, formada por um gerente geral que era o proprietário, um coordenador de desenvolvimento e cinco programadores, sendo apenas um deles Sênior e os outros programadores Júniores. A empresa utilizava a linguagem de programação PHP, e suas principais atividades eram pequenas demandas para softwares que já estavam em uso por seus clientes. A empresa se caracterizava por ser de baixo custo operacional, portanto com baixo custo de serviço para os clientes.

3.7.2 Metodologia aplicada.

Em reunião de acerto do projeto, ficou definido que o ADQ2 seria utilizado na empresa e a avaliação de qualidade não seguiu o planejamento Padrão previamente organizado para a pesquisa, que seria o uso do GQM.

Após apresentada a metodologia GQM, ficou claro através da experiência da empresa em projetos e também para minimizar

desconfortos futuros, que a realização da fase GQM fosse posterior a um projeto piloto, pois não só envolvia mudanças no processo, mas também aprendizados de técnicas de testes e do uso do framework PhpUnit.

O pré-projeto foi chamado de Pré-piloto por não ser o desenvolvimento de algum novo código, mas sim a escolha de uma parte de algum projeto, a criação dos testes de unidade e verificação de seus resultados para melhoria do know-how da equipe e, principalmente, verificação da aderência do ADQ2 à empresa.

Para organizar essa fase, algumas reuniões foram realizadas e os seguintes passos foram definidos para o pré-piloto:

- **Treinamento básico de testes;**
- **Treinamento de PhpUnit;**
- **Escolha do trecho de código que seria utilizado no projeto piloto;**
- **Desenvolvimento de testes de unidade;**
- **Avaliação de Resultados;**
- **Avaliação de Aderência;**
- **Planejamento GQM para a nova fase do projeto.**

Para esta primeira parte do projeto, o pré-piloto, foram escolhidos para participar um coordenador de desenvolvimento, um desenvolvedor e um estagiário que representava o embrião da área de qualidade que o proprietário desejava criar a partir dessa iniciativa.

3.7.2.1 Treinamento teórico de testes.

O treinamento foi organizado em três partes, buscando responder às seguintes questões:

O que é Teste de Software? Identificando claramente o que é o sucesso em um teste.

Quais são os tipos mais comuns de testes? Identificando diferenças entre caixa preta e caixa branca, além de algumas técnicas para melhorar o esclarecimento, mas o foco será nos dois testes de caixa preta que foram inicialmente pensados para uso nesta fase: particionamento de equivalência e análise valor limite.

O que é um teste de unidade? Neste ponto, também é apresentada a automação através dos diversos frameworks xUnit.

Para minimizar o impacto nos prazos da empresa e desenvolvedores, os treinamentos seriam organizados em três ou mais dias diferentes em períodos de 60 minutos cada.

3.7.2.2 Treinamento de PhpUnit

Nesta fase, iniciamos o conhecimento de testes de unidade e sua aplicabilidade no phpUnit.

Embora a programação fosse apenas o uso de 60 minutos, esta fase foi realizada em aproximadamente três horas e essa atividade dividida em dois dias da semana, pois o uso do phpUnit não é trivial e necessita de um esforço extra, que a equipe de desenvolvedores se empenhou em realizar.

Neste treinamento, foram vistos:

- **Como configurar para utilizar o phpUnit;**
- **Como criar testes de unidade com phpUnit;**
- **Como verificar o resultado desses casos de teste.**

Os casos de teste, neste treinamento, foram realizados manualmente, mas dos participantes surgiu a idéia de utilizar a automação na geração de testes de unidade através do phpDoc, atividade essa testada e aprovada para uso futuro, não para essa fase do pré-piloto.

Observar que o phpDoc permite que os asserts sejam feitos em linhas comentadas no código php, e automaticamente, os testes de unidade podem ser gerados. Considerado ótimo na produção de novo software, poderia ser um problema para o pré-piloto.

3.7.2.3 Escolha do trecho de código.

O projeto escolhido para definição do trecho de código que passaria por testes foi realizado pelo proprietário, que escolheu o projeto que mais gerava problemas de qualidade.

A escolha do código foi realizada pelo pesquisador junto ao desenvolvedor, atividade essa que não foi simples, pois, para o pesquisador, o conhecimento da ferramenta e suas funcionalidades tiveram que ser explicitadas pelo desenvolvedor para poder auxiliar na escolha.

Uma constatação interessante, nesse ponto, foi que os projetos eram feitos em php com programação estruturada, sem o uso de Orientação a Objetos. Nesse projeto escolhido, existia uma classe de

validações que era utilizada junto com o restante do código, portanto existia somente uma classe de validações genéricas no projeto e foi essa a escolhida.

Motivos da escolha foram:

- **Classe de fácil compreensão;**
- **Facilidade no domínio pelo desenvolvedor.**

Essa classe possuía trinta métodos, dos quais apenas nove foram escolhidos para o pré-piloto, para não impactar no tempo disponibilizado pelo desenvolvedor e coordenador (quando este participava) para a condução da atividade.

É importante ressaltar a preocupação do pré-piloto em não atrapalhar as demandas diárias. Ocorreram várias interrupções no decorrer das atividades de treinamento e de desenvolvimento dos testes, não planejadas pela equipe, mas esperadas pelo pesquisador.

3.7.2.4 Desenvolvimento dos testes de unidade.

Para o desenvolvimento dos testes, um novo ambiente foi criado que não era o ambiente de pré-produção, e também não era o ambiente de desenvolvimento, o que demandou um tempo maior que o esperado.

Avaliação dos requisitos feita através dos documentos quando estes existiam, mas, na maioria dos casos, os métodos foram estudados para entender seu funcionamento, observando software produzido pela empresa.

Após entendimento dos requisitos ocorreu a criação de uma planilha com os testes de unidade que seriam realizados, sem ter

programado no xUnit Framework. Essa fase foi uma revisão da técnica de limites apresentada no treinamento de testes.

Em seguida, o desenvolvimento das linhas de código de testes (TLOC).

Ao final, os resultados foram coletados e trabalhados em planilha para geração de gráficos e análise de seus resultados.

3.7.2.5 Avaliação de Resultados

A execução do pacote de casos de testes e avaliação dos resultados podem ser vistos nos dados abaixo:

Total de métodos: 31.

Total exercitado: 9.

Casos de teste criados: 52.

Casos de teste que expuseram defeito: 22.

Casos de teste que não expuseram defeito: 30.

Esta atividade piloto tinha a função principal de melhorar o conhecimento e avaliação da aderência aos processos da empresa, mas com esses resultados o pré-piloto ultrapassou em muito o esperado dele, pois 42% dos casos de testes apresentaram defeitos e isso mostrou a vulnerabilidade no código que a empresa produzia.

Esses números redirecionaram os esforços dos proprietários em projetos futuros de melhoria de qualidade e também explicitaram a necessidade de métricas para acompanhamento de qualquer incursão nas atividades de qualidade para projetos futuros.

3.7.2.6 Avaliação de Aderência.

Embora o resultado do pré-piloto tenha sido de muito sucesso devido à exposição de várias fragilidades no código, esses resultados surgiram de um trecho orientado a objetos de um projeto em que todo ele era codificado no paradigma estruturado, excetuando uma única classe, que foi testada. Embora fosse possível realizar testes de unidades com phpUnit em trechos estruturados, o esforço era enorme e a dificuldade de sua implementação não tinha relação com a agilidade no processo que foi a busca desse projeto.

Para continuidade do projeto, deveríamos ter uma nova demanda em programação e essa nova demanda deveria ser realizada no paradigma orientado a objetos, isso foi um delimitador para a fase seguinte, pois essa mudança não havia sido planejada ainda pelo coordenador da equipe.

Uma característica que deve ser ressaltada aqui é que a empresa tem alguns produtos já implantados e as vendas normalmente estão relacionadas à customização desses produtos que já existem, portanto a demanda por um novo produto não existia em nenhum planejamento ou prospecção de venda.

Ao final de algumas reuniões entre proprietário, coordenador de desenvolvimento e pesquisador, a continuidade do projeto ficou condicionada a alguma nova proposta de desenvolvimento e o projeto foi congelado até essa data.

3.7.2.7 Planejamento atividades futuras.

O planejamento de atividades futuras foi cancelado devido à incerteza de data e tempo hábil para execução desse trabalho.

3.8 Empresa C (São Paulo / ZO)

3.8.1 Caracterização da Empresa.

Empresa localizada na Zona Oeste de São Paulo, constituída por dois sócios, um responsável pela área de arte digital e outro pela área de desenvolvimento de software. Sua produção eraa, principalmente, baseada em aplicações PHP e a criação de portais através de CMS específico. Possuía um grupo de sete programadores, sendo três deles mais experientes, considerados sêniores, dois desenvolvedores plenos e três juniores (ou iniciantes). Um ponto forte nessa empresa foi a contratação de um gerente de projetos que atuou nas mudanças de processos para adequação do ADQ2 na empresa.

3.8.2 Metodologia aplicada.

Em reunião de acerto do projeto ficou definido que o ADQ1 seria utilizado na empresa e a avaliação de qualidade teria toda condução do projeto baseado no GQM, conforme pode ser visto nas sessões seguintes.

3.8.2.1 Reunião Definição Metas.

A identificação do objeto de estudo exige esforço para não ser superficial. Deve-se evitar o engano ao escolher como objeto uma característica do que realmente se deseja como Meta.

Na primeira reunião de construção da fase GQM, tivemos à mesa o pesquisador acadêmico, o gerente de tecnologia, o usuário chave e os dois proprietários da empresa que desempenham funções administrativas, técnicos e comerciais.

Previamente, em planejamento anterior, ficou definido que as reuniões GQM deveriam ter entre 30 minutos de duração e uma hora, mas ficou claro que o material gerado a cada reunião deveria ser fechado no dia, portanto o atraso seria aceito apenas para fechamento dos entregáveis por reunião. Nesta primeira reunião, o entregável seria o documento que define Objeto, Propósito, o Foco de Qualidade, Ponto de Vista e Ambiente, documento este chamado de formulário de Meta.

Ao final dessa reunião, tivemos:

Objeto: qualidade no produto de software

Propósito: melhora na qualidade

Foco de Qualidade: a qualidade será avaliada em relação ao histórico da empresa nos últimos projetos entregues.

Ponto de Vista: ponto de vista do proprietário da empresa e do pesquisador.

Ambiente: empresarial.

Nesta primeira reunião, ficou decidido que as outras reuniões deveriam ter as mesmas pessoas envolvidas, mesmo que gerasse atraso para agendamento de todas em datas nas quais todos

estivessem disponíveis. Essa decisão surgiu, quando os proprietários perceberam que o sucesso do projeto poderia depender de como essas reuniões seriam conduzidas e dos entregáveis gerados a partir dessas reuniões.

3.8.2.2 Reunião Definição Questões.

Para a reunião de definição de Questões, foi decidido que faríamos um brainstorm para encontrar quantas questões fossem possíveis em relação ao objeto proposto na reunião de meta, mas também ficou decidido que o tempo dessa reunião não deveria exceder uma hora e trinta minutos, caso fosse necessário, terminaríamos o assunto na manhã seguinte.

Essa reunião iniciou-se tarde, e teve uma hora e quinze minutos de trabalhos. Com as várias questões e ainda não tendo fechado o grupo de questões desse entregável, os participantes decidiram finalizar na manhã seguinte, quando as seguintes questões foram aprovadas para o planejamento:

Q1: Qual é a quantidade de defeitos encontrados?

Q2: O quanto a quantidade de defeitos representa no total de trabalho realizado.

Q3: Qual o tempo previsto e gasto para correção dos defeitos?

Q4: Qual o custo de correção dos defeitos?

Q5: Qual é a influência por desenvolvedor na qualidade geral do produto?

Q6: Uma sexta questão foi definida e, mais tarde, colocada na análise da questão cinco: Qual é o percentual de defeitos por faixas de experiência dos programadores?

Ficou claro que a preocupação da questão cinco e seis auxiliaria em treinamentos mais ajustados às necessidades dos desenvolvedores. Ainda, ao final, foi identificado que as perguntas que identificamos já eram perguntas que passavam em reuniões dos proprietários, mas que, devido à falta de organização, nunca tinham colocado em prática alguma ação para identificar possíveis resultados ou respostas as mesmas.

Outra preocupação que ficou explícita dessa reunião foi o fato de os defeitos terem que possuir classificações, para melhor acompanhamento e priorização. O local onde o defeito foi exposto também passou a ser algo de valor para a empresa, pois isso poderia diminuir o desgaste em relação aos clientes, situação essa que não existia na planilha de correções que era passada para a equipe de desenvolvimento. Não existiam diferenças entre defeitos, quem havia descoberto o defeito, o cliente ou alguém da equipe. Além desses dois, também foi identificada a necessidade de adicionar nessa planilha os defeitos descobertos ainda em desenvolvimento, mas que não poderiam ser corrigidos por algum motivo qualquer.

3.8.2.3 Reunião Definição das métricas.

Nesta reunião, fizemos a identificação das métricas por questão, independentemente de elas serem repetidas ou não. Seguem questões e suas métricas.

Q1: Qual é a quantidade de defeitos encontrados?

M1: Total de defeitos encontrados em testes

M2: Total de defeitos reportados pelo Cliente

M3: Total de defeitos conhecidos pelo desenvolvimento

M4: Total de defeitos conhecidos.

Q2: quanto a quantidade de defeitos representa no total de trabalho realizado.

M1: Medida de tamanho de código (LOC)

M2: Densidade de defeitos

M3: Densidade de defeitos por programador.

M4: Densidade de defeitos por experiência de programadores.

Q3: Qual o tempo previsto e gasto para correção dos defeitos?

M1: Horas Trabalhadas na correção

M2: Horas planejadas para correção

M3: percentual de diferença entre planejado e executado nas horas de correção.

Q4: Qual o custo de correção dos defeitos?

M1: Tempo gasto por programador

M2: Tempo gasto por Experiência para correção

M3: Custo por defeitos (aproximação).

Q5: Qual é a influência por desenvolvedor na qualidade geral do produto?

M1: Densidade de defeitos por programador

M2: Densidade de defeitos por experiência do programador.

3.8.2.4 Reunião Definição da coleta de dados por Questão.

Neste ponto, ficou evidenciada a necessidade de organizar melhor os processos da empresa, pois não se tinha idéia, em princípio, de onde encontrar as métricas dentro do que acontecia na empresa.

Na definição da coleta, foi buscada a resposta a qual seria o método de coleta, como seria realizado e quando, como pode ser visto a seguir:

Q1: Qual é a quantidade de defeitos encontrados?

Método de coleta:

- Institucionalizado um período de testes e um conjunto de técnicas de teste que todos utilizariam.

- Lançamento em planilha de cálculo os defeitos encontrados, planilha esta em rede com acesso a todos e protegida por software de controle de versão.

Como:

- Cada módulo desenvolvido possuiria 30% do tempo previsto para testes, e seria realizado por outro programador da equipe, não mais pelo desenvolvedor do módulo.

- Os testes inicialmente seriam realizados de forma completamente manual, digitação em planilha de testes e organizados em fases: preparação, execução e fechamento.

- Cada programador tinha um nome (de desenvolvedor no papel de testador) para passar a planilha e a atividade de teste; em caso de problemas de conflito, o gerente receberia essa atividade e alocaria alguém para sua realização.

- Outro ponto importante é que o desenvolvedor que fazia testes, quando em suas atividades de desenvolvedor não inverteria de posição com o colega em que ele foi o testador.

- Ficou definido que quinzenalmente a dupla desenvolverdor / testador seria alternado na empresa.

Observar que, neste ponto, pelo fato de a empresa ser pequena e a mudança no hábito dos desenvolvedores ser grande, os proprietários preferiram uma abordagem mais informal, mas que num futuro próximo, provavelmente iriam criar sua área de teste, este passo foi considerado o embrião da área de testes da empresa.

Quando:

- Após validação do código pelo programador.

Q2: O quanto a quantidade de defeitos representa no total de trabalho realizado.

Método de coleta:

- Usar um programa para contagem de linhas.

- Lançamento em planilha própria.

Como:

- A cada entrega para testes, o desenvolvedor que irá fazer a função de testar irá executar o programa de testes e lançar o número de linhas na planilha própria para isso.

Quando:

- Toda vez que o código for enviado para teste.

Q3: Qual o tempo previsto e gasto para correção dos defeitos?

Método de coleta:

- Alteração na Time Sheet (planilha de horas de trabalho) para apresentar detalhes das horas trabalhadas.

- Todo trabalho passado ao programador terá uma previsão de tempo que será lançada pelo gerente ou, quando necessário em conjunto com o desenvolvedor. (quem passa a demanda coloca uma previsão, essa é a meta).

Como: Lançamento de planilha.

Quando: No momento de passar atividade para o programador.

Q4: Qual o custo de correção dos defeitos?

Método de coleta:

- Uso do valor hora do programador

Como: Cruzamento entre a planilha de horas e o valor/hora por programador

Quando:

- Atividade semanal do gerente, a cada fechamento de planilha de horas da semana.

Q5: Qual é a influência por desenvolvedor na qualidade geral do produto?

Método:

- uso de planilhas da questão 1

- Uso da time-sheet para comparação de produção e desenvolvimento e cruzamento dos dados por programador.

Como:

- Uso de interligação entre planilhas eletrônicas através de macros.

Quando:

- Automático.

3.8.3 Fase Coleta dos dados

O trabalho de coleta de dados foi realizado em fases, a primeira, visava à verificação e à adequação, dados esses que foram desconsiderados para o resultado de pesquisa, considerada essa primeira como fase de aprendizado e ajustes. Esse primeiro período ocorreu em aproximadamente três semanas.

A coleta de dados, após várias intervenções, passou a ser bastante automática e tranqüila para o gerente, após um mês de trabalhos a equipe ficou habituada aos processos implementados pela empresa, o gerente e o usuário chave foram pontos importantes para que essas modificações fossem realizadas.

3.8.4 O Processo e sua implantação

Uma das primeiras ocupações da equipe foi montar a estrutura de processos e papéis que seria criada e alterada. Como

preocupação inicial, ficou definido que os controles seriam pouco automatizados e que depois de consolidados a empresa poderia desenvolver uma ferramenta de apoio. O ponto crítico para todo o processo seria o documento de requisitos passado ao desenvolvedor. Neste ponto ficou decidido que um documento impresso, com anotações seria o material que iria para o programador e a definição de um formulário bem organizado com explicações sobre o requisito, banco de dados, dependências e campos para coleta de dados das métricas definidas no GQM. Para isso, foi criado um formulário, conforme Ilustração 18, que aos poucos foi melhorado e através deste muitos dados seriam coletados.

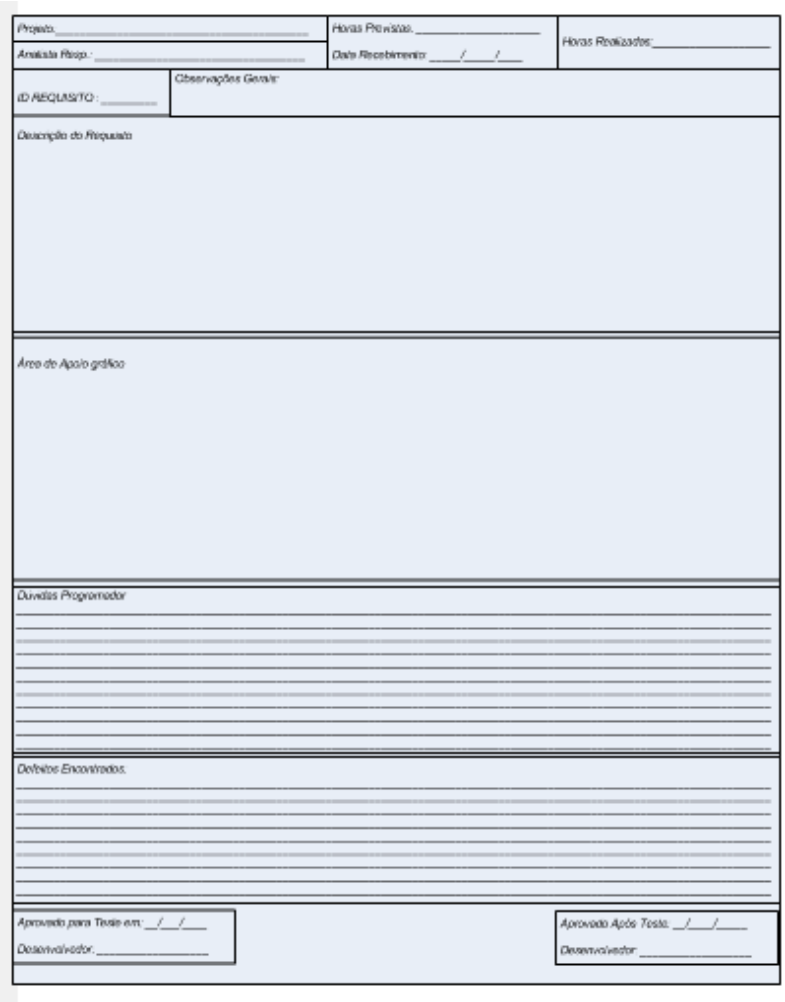


Ilustração 18 - Formulário de Requisito

Observar que o gerenciamento dos requisitos continuará em uma planilha já existente na empresa e os aceites inicialmente definidos junto ao cliente também estarão nessa mesma planilha, mas em área separada e que os desenvolvedores tem acesso livre e também para o testador.

Como o procedimento, com o formulário de requisito, seria manual, o trâmite desse documento também tinha que ser extremamente simples e claro, para evitar problemas na implantação do processo, o modelo foi adaptado nesta empresa com caixas para cada desenvolvedor, o desenvolvedor possuía duas caixas em sua mesa, uma, para depósito de novas demandas e outras, para demandas executadas e terminadas.

O gerente também teria duas caixas em sua mesa: uma para recebimento de requisitos com dúvidas e outra para recebimento de requisitos testados e aprovados.

As demandas poderiam ser:

Formulário de Requisito para desenvolvimento.

Formulário de Requisito pós-desenvolvimento (para testes).

Formulário de Requisito para correção (pós-teste).

Formulário de Requisito aprovado.

A Ilustração 19 mostra o processo e, em seguida, uma explicação do que seriam procedimentos para cada um dos personagens desse processo. Observar que o conceito de papel exercido é usado nesse processo, o desenvolvedor e o testador podem ser um mesmo desenvolvedor, porém em momentos diferentes no processo.

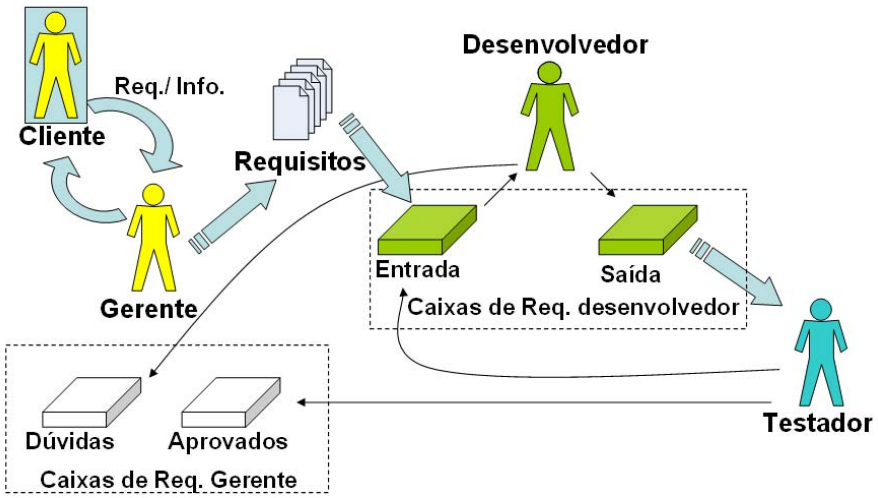


Ilustração 19 - Processo de tratamento de requisitos

Fase 1: Gerente é responsável pelos requisitos e pelo processo de documentação e entendimento junto ao cliente. Em caso de dúvida em um requisito, seu entendimento junto ao cliente é prioritário e executado pelo Gerente.

Fase 2: O desenvolvedor, todos os dias, avalia os requisitos em sua caixa de Entrada, quando há um novo requisito, ele deve verificar seu entendimento, não entendendo algum dos requisitos, estes devem ser verificados junto ao gerente, de forma escrita no próprio formulário, falada ou por e-mail e o requisito é devolvido ao gerente (colocado na caixa de dúvias). Caso o entendimento seja claro e o tempo previsto estiver de acordo, ele anota data e hora no formulário de aceitação do requisito e à medida que termina o desenvolvimento vai liberando os formulários na caixa de saída , ou caixa de testes.

Fase 3: Neste momento, entra o papel do testador, que é um outro desenvolvedor que irá desempenhar o papel de testador. Toda semana os desenvolvedores são organizados em pares de desenvolvedor/Testador. O testador de um desenvolvedor específico

(desenvolvedor A) não pode ter esse (desenvolvedor A) como o seu testador, isso é para evitar acomodações.

Fase 4: O testador, após realização dos testes e verificação da passagem ou não dos mesmos, irá devolver para o responsável pelo desenvolvimento do requisito, colocando na caixa de entrada como os defeitos encontrados. Caso o Testador não encontre problema algum, ele coloca o formulário na caixa de requisitos aprovados na mesa do gerente.

O gerente é responsável por fazer o fechamento do requisito em sua planilha, lançar os dados necessários em suas respectivas planilhas.

Observação: Este foi o procedimento realizado e que permitiu análises de falhas e sugestões dos programadores. O que foi percebido logo nos primeiros dez dias é que, algumas vezes, a caixa de entrada possuía muitos requisitos retornados de testes, novos e alguns que ainda não tinham sido iniciados e a priorização desses serviços dependia de gerenciamento. Para evitar isso, o stand-up meeting (atividade utilizada no Scrum) foi implementado para todos os dias, onde rapidamente todos os programadores no início dos dias relatavam as pendências e o gerente organizava, priorizava os requisitos e até reorganizava as demandas entre os desenvolvedores.

Segue lista de atividades de cada um dos personagens do processo:

Gerente (Analista Sênior):

Organizar os requisitos (planejamento de desenvolvedor e prazo por requisito).

Entrega demanda ao desenvolvedor.

Receber dúvidas de requisitos e avaliar.

Contata o cliente para resolução de dúvida.

Recebe requisito aprovado e integrado.

Desenvolvedor:

Analisa e avalia entendimento do requisito

Passa o requisito de volta ao gerente em caso de não entendimento ou não concordar com a previsão de horas para o requisito.

Desenvolve e valida funcionamento

Passa para o testador (caixa de saída).

Recebe requisitos com defeitos para correção.

Corrigir defeitos e disponibilizar novamente na pasta de saída para teste.

Testador:

Recebe o requisito, avalia o aceite definido pelo cliente (ou gerente).

Integra em ambiente de pré-produção.

Faz os testes de aceite, teste integração e testes aleatórios.

Em caso de defeitos descreve em planilha e devolve para o programador.

Anotar no formulário de requisito, após aprovação, o número de linhas do código.

Verificar / lançar os dados de controle no formulário que deveriam ser laçados pelo desenvolvedor.

3.8.5 Dados coletados no processo x GQM.

3.8.5.1 Q1: Quantidade de defeitos.

M1: (Número de defeitos) retirado do formulário de requisitos

M2: (Número de defeitos percebido pelo cliente) recebido pelo gerente e lançado em planilha de requisitos, esses defeitos percebidos pelo cliente geram novos requisitos para o desenvolvedor, um requisito de correção.

M3: Quando o desenvolvedor perceber defeitos que surgem de seus ajustes nos requisitos, ele os descreve no documento, mas isso só ocorrerá em situações em que se sabe que outra atividade irá sanar esse requisito e que essa está em trâmite, mas é importante ser registrada.

M4: (total de defeitos conhecidos) somatória de M1, M2 e M3.

Observar que esse formato de atuação gerou desconforto, pois o trabalho manual não é muito aceito entre os desenvolvedores, mas em reunião com todos ficou claro que o processo deveria ser solidificado antes de se pensar em criar uma ferramenta de apoio e todos ao participar dessa reunião aceitaram o desconforto. Devido à experiência deles, ficou claro que uma ferramenta automatizada que não atende é pior que uma atividade manual que tem flexibilidade para mudanças (anotações de roda pé por exemplo).

3.8.5.2 Q2: Quanto a quantidade de defeitos representa no total.

M1: linhas de código por requisito, após aprovação, esse dado é coletado e lançado na folha do requisito aprovada e coletado pelo gerente e lançado em planilha própria.

M2: a densidade de defeitos é definida por M4 de Q1 / M1. (M4 de Q1 é a métrica 4 da questão 1).

M3: através dos requisitos é possível agrupar os defeitos gerados por programador e calcular sua densidade em planilha própria do gerente.

M4: é realizada, seguindo a mesma lógica de M3, mas agrupando os resultados entre grupos de programadores por experiência no mercado (Sênior, Pleno e Junior).

Nesta métrica, o uso de linhas de código por requisito gerava problemas quando o requisito utilizava outras linhas de código, ou era um requisito de correção. Para solucionar isso foi, utilizado o total de linhas de código.

3.8.5.3 Q3: Tempo previsto e realizado.

M1: Previsão de correção era responsabilidade do desenvolvedor que lançava antes de iniciar a correção e o gerente coletava no formulário.

M2: As horas realizadas eram retiradas da time sheet dos desenvolvedores que passou a ser mais detalhada.

M3: Percentual calculado entre M1 e M2 (exemplo: realizado estourou em 10% o previsto), calculo esse realizado pelo gerente.

M4: Tempo previsto por requisito é lançado no formulário pelo gerente e revisto pelo programador, que em caso de dúvida voltava para o gerente reavaliar. Dado esse coletado pelo gerente ao receber o requisito aprovado.

M5: as horas realizadas eram retiradas da time sheet , que passou a ter importância maior que apenas apontar horas trabalhadas.

3.8.5.4 Q4: Custo de correção de defeitos.

M1: O tempo gasto por desenvolvedor retirado pelo da *time sheet* e lançada em planilha de controle do gerente.

M2: Uso da M1, porém organizado por grupos definidos por experiência de mercado (Sênior, Pleno e Junior).

M3: O custo por defeito era definido por M1 / por Q2.M1 (M1 de Q2 é a métrica 1 da questão 2) e lançado em planilha própria pelo gerente.

3.8.5.5 Q5: Influência do desenvolvedor na qualidade total do produto.

M1: uso de Q2.M3 (M3 de Q2 é a métrica 3 da questão 2) retorna a densidade de defeitos por programador, neste ponto ficou definido que o gerente faria um gráfico que apresentasse o que cada um dos programadores representariam em percentualmente em relação ao total desenvolvido.

M2: uso de Q2.M4 (M4 de Q2 é a métrica 4 da questão 2) da mesma forma que a M1 acima.

4 RESULTADOS

4.1 Empresa A

Nesta empresa, não ocorreu nada além da execução de treinamentos de testes, algumas reuniões para definição de usuário chave, uso do GQM, como seria a estruturação do processo.

Dados históricos foram levantados, mas seu uso ficou de lado, pois a principal função era realizar uma comparação em relação ao uso do modelo ADQ1, que, em visita à empresa, ficou claro que iria não só melhorar a comunicação entre o desenvolvedor e o cliente, e dar suporte para a melhora esperada, como iria praticamente estruturar esse procedimento, já que a empresa tinha no vendedor o gerador de requisitos e o contato com o cliente só era realizado pelo gerente em situações críticas. Essa empresa por estar sempre em atraso com seus projetos, preferia em reuniões levar código em funcionamento, mas normalmente esse código apresentava problemas de interpretação em relação aos desejos do cliente.

Embora essa fosse a empresa que mais necessitaria do desenvolvimento de qualquer plano de melhoria, essa foi a empresa que menos se envolveu com o processo, e após a saída do usuário chave, ficou impossível qualquer tipo de ação.

4.2 Empresa B

Esta empresa obteve bons resultados em seu pré-projeto, ao expor defeitos quando estes não eram esperados, pois a função do pré-piloto, como era chamado, era o de aprendizado do processo e minimizar o impacto das mudanças que seriam implantadas.

Os números apresentados, no projeto piloto, foram expressivos, pois o software avaliado já estava em produção há muito tempo e o trecho escolhido apresentou fragilidade. Dos 52 casos de testes criados, 22 deles expuseram defeitos. Ou seja, 42% dos casos de testes obtiveram sucesso em sua execução.

Para continuidade com o ADQ2, existiria a necessidade de mudanças no paradigma de programação utilizado, já que em sua maioria os xUnits são orientados a objetos e também devido ao aprendizado no pré-piloto, onde o tempo necessário para desenvolver testes em aplicação não orientada a objetos ser alta para realizar os testes de unidade com o phpUnit.

O que se percebeu é que os métodos não possuíam validações nos limites, e também que a responsabilidade de testar esses valores estava na camada de interface, através de JavaScript, o que tornava a situação extremamente replicada, pois a cada uso desses métodos, a cada página, um script era utilizado para realizar essas validações. Neste momento, uma idéia ficou para análise posterior, se não seria interessante utilizar o framework xUnit de JavaScript. Essa nova idéia não passou a ser viabilizada até o fechamento deste documento.

4.3 Empresa C

Nesta empresa, os números foram coletados e nas páginas seguintes seguem análises de seus resultados.

Ocorreu uma diminuição de aproximadamente 35% dos defeitos em relação ao total histórico, porém o mais importante foi em relação aos percebidos pelo cliente que diminuíram em aproximadamente 73% no período analisado. Isso mostra que menos defeitos foram injetados e que a percepção de qualidade dos clientes aumentou à medida que menos defeitos são expostos pelo usuário da aplicação.

Os custos de projeto ficaram mais baratos por KLOC, no geral.

4.3.1 Dados históricos

Para traçar um perfil da empresa com algumas variáveis, dois projetos foram escolhidos pelo gerente de tecnologia, temos os dados abaixo:

Projeto A

- **Tamanho (LOC): 7375**
- **Quantidade de defeitos: 181**
- **Densidade de defeitos: 181 / 7,375 = 24,5 def. /KLOC**
- **Tempo previsto em horas: 360 horas**
- **Tempo realizado em horas: 495 horas**

- **Diferença de horas: 135 horas**
- **% lucro planejado: 50 %**
- **% lucro obtido: 18,75%**

Projeto B

- **Tamanho (LOC): 6140**
- **Quantidade de defeitos: 137**
- **Densidade de defeitos: 137 /6,14 = 22,31 def./KLOC.**
- **Tempo previsto em horas: 290**
- **Tempo realizado em horas: 330**
- **Diferença de horas: 40 horas**
- **% lucro planejado: 50%**
- **% lucro obtido: 43,10**

Os valores considerados foram a média dos dois projetos, que segue logo abaixo

<u>Métricas históricas consideradas</u>

- **Densidade de defeitos: 23,4 defeitos / KLOC.**
- **Diferença entre previsto e realizado: 87,5 horas**
- **% lucro planejado: 50% (padrão)**
- **% lucro obtido: 30,9%**

4.3.2 Dados gerados no processo

Os dados definidos no GQM foram mais extensos que os dados recuperados do histórico da empresa, portanto para análise só serão utilizados os dados que auxiliem em qualquer tipo de comparação e análise.

Métricas geradas no GQM:

- **Total de defeitos encontrados em testes: 42**
- **Total de defeitos reportados pelo Cliente: 29**
- **Total de defeitos conhecidos do desenvolvimento: 3**
- **Total de defeitos conhecidos: 74 defeitos**
- **Medida de tamanho de código (LOC): 4720**
- **Densidade de defeitos: 15,04 defeitos / KLOC**
- **Densidade de defeitos por programador :**
  - **Programadores A: 20,9 defeitos/KLOC**
  - **Programadores B: 6,2 defeitos/ KLOC**
  - **Programadores C: 18,8 defeitos/ KLOC**
- **Densidade de defeitos por experiência de programadores:**
  - **Programadores Sênior: 20,9 defeitos/KLOC**
  - **Programadores Pleno: 6,2 defeitos/ KLOC**
  - **Programadores Juniores: 18,8 defeitos/ KLOC**

Observar que, no projeto analisado, apenas os três desenvolvedores participaram, portanto densidade por desenvolvedor e densidade por experiência são idênticos.

- **Horas Trabalhadas na correção: 60 horas.**
- **Horas planejadas para correção: 40 horas.**
- **% de diferença entre planejado e executado nas horas de correção: 20 horas (50% a mais do planejado).**
- **Tempo gasto por programador na correção**
  - **Programadores A: 10 horas**
  - **Programadores B: 15 horas**
  - **Programadores C: 35 horas**
- **Tempo gasto por Experiência para correção**
  - **Programadores Sênior: 10 horas**
  - **Programadores Pleno: 15 horas**
  - **Programadores junior: 35 horas**
- **Custo por defeitos (aproximação): R$ 11, 28**

4.3.3 Cruzamento e análise dos dados

Temos uma diferença razoável entre a densidade média de defeitos entre os projetos A e B, que é de 23.4 defeitos / Kloc para a densidade de defeitos após as mudanças, que é de 15,04 defeitos /Kloc.

Na qualidade alcançada, os números exposição de defeitos para os clientes, que antes era em média 159 (181 para projeto A e 137 para projeto b) e passa a ser de apenas 29 defeitos. A percepção de qualidade do produto para o cliente foi muito expressiva, quando pensamos em densidade de defeitos temos 6,14 defeitos / Kloc que foram descobertos pelo cliente.

Outros valores estudados pelo GQM fazem parte dos estudos que a empresa começa a fazer de seu perfil de qualidade e busca pela melhoria contínua, onde ela pretende após o término do segundo projeto com essa reestruturação montar números base para os projetos futuros e trabalhar para a diminuição dos defeitos com outras técnicas de testes.

5 CONCLUSÕES E SUGESTÕES

5.1 Conclusão

O trabalho realizado apresenta resultados favoráveis tanto no projeto realizado pela empresa B, que não chegou a ser desenvolvido até o fim, mas que gerou poucos dados para a pesquisa, mas gerou informações suficientes para a motivação de novos projetos de qualidade. Com a empresa C, foi possível trabalhar mais e fazer mais análises, sendo produtivo tanto para a pesquisa quanto para o empresário. No caso da empresa A, infelizmente não ocorreu ganho para nenhuma das partes aparentemente, pois o processo foi interrompido por diversas vezes, ocorreram mudanças internas nos recursos, muitas vezes, até os treinamentos deveriam ser repetidos caso fosse possível ter continuado a parceria na pesquisa. Isso se deve ao fato dessa empresa estar passando por reestruturação há algum tempo, desde que houve uma mudança societária na empresa e ocorreu divisão da empresa e de projetos (clientes) e, conseqüentemente divisão, do financeiro da empresa.

Aspecto percebido de forma empírica: a dificuldade em desenvolver projetos em empresas, devido à dinâmica do mercado e às constantes mudanças na empresa, talvez maiores em empresas pequenas, onde o funcionário, ao ter mais experiência, procure oportunidades melhores do ponto de vista financeiro e gere uma maior rotatividade no quadro funcional.

A hipótese inicial de que os ADQs por proporcionarem menores mudanças na empresa teriam uma mais fácil entrada nos processos internos não fica evidenciado, isso talvez devido à falta de

material para análise, apenas uma das empresas conseguiu chegar ao final do processo.

A hipótese de uma pequena mudança, apresentando rapidez nos resultados poder motivar mais a empresa a continuar com melhoria nos processos. Isso ficou evidente após reunião com duas das empresas, onde os responsáveis indicaram interesse em continuar com outras pesquisas, ou até iniciando dentro dessas organizações uma área de qualidade.

A hipótese de que ocorreria melhoria significativa, e isso significa algo em torno de 30% a 40% na melhoria da qualidade da empresa C, onde ocorreu uma queda de 35.7 % na densidade de defeitos, mas que não representa a visão de qualidade passada para o cliente, que passou a perceber uma melhoria de 73.28% na diminuição de defeitos que eram descobertos quando o software estava em uso.

Para melhorar os valores referentes a este processo, um novo levantamento de dados se faz necessário em momento futuro, quando a influência do pesquisador na atuação seja menor e a motivação inicial de melhoria de qualidade alcance os patamares de normalidade, onde apenas o processo irá influenciar nos resultados.

5.2 Sugestão

Para um novo trabalho, o ideal seria um projeto mais longo, passando por mais de um ADQ e tendo os dados analisados a cada implantação de novo processo e, ao final, analisar a evolução da qualidade desde antes do primeiro ADQ, até o último a ser implantado para análises individuais e por análises de evolução geral.

Em experimentos no ambiente empresarial, todo e qualquer detalhe externo pode influenciar no desenvolvimento, por isso o

tempo, para qualquer atividade desse tipo, não pode ser muito restrito, pois a falta de tempo pode inviabilizar todo o processo.

Para trabalhos futuros, o grupo de ADQs pode ser aumentado, já existindo estudos para a definição de três novos ADQs, sendo eles:

ADQ4 relacionado aos teste, expandindo os procedimentos gerados pelo ADQ2 e ADQ3, automatizando mais procedimentos de testes como os de integração, aceitação e regressão.

ADQ5 relacionado aos procedimentos após implantação, criando um processo de interfaceamento entre os defeitos e os requisitos, onde o uso de ferramenta open source pode fazer a diferença nos custos e na aquisição de *know-how*, facilitando a identificação dos defeitos, classificação e aumentando a participação do cliente através da web.

ADQ6 está relacionado a métricas in-process utilizadas por KAN (2001), que podem auxiliar no acompanhamento durante o desenvolvimento ou o uso de métricas STREW-x.

6 REFERÊNCIAS

ABIB, Janaina C.; KIRNER, Tereza G. A GQM-Based Tool to support the Development of Software Quality Measurement Plans. ACM SIGSOFT, vol. 24, no. 4, USA, 1999.

AMBLER, Scott W. Modelagem Ágil: Práticas eficazes para a Programação Extrema e o Processo Unificado, Editora Bookman: Porto Alegre, Brasil, 2004.

ALDAY E. Contreras. O Plajenamento Estratégico dentro do Conceito de Administração Estratégica, Revista da FAE. Curitiba, UniFAE, v. 3, n. 2, p. 9-16. 2000.

BACK, James. SCRUM Software Development Process. Controlchaos.com. 1996. Disponível em www.controlechoaos.com/old-site/scrumwp.htm. Acesso em 27/01/2007.

BECK. Kent et al. Manifesto for Agile Software Development. Agilesoftware.org. 2001. Disponível em www.agilemanifesto.org. Acessado em: 15/01/2007.

BECK, Kent. Embrace Changing with eXtreme Programming. *IEEE Computer*. V. 32, n.10, p.70 – 77, 1999.

BECK, Kent. Programação Extrema: Acolha a mudança, Editora Bookman: Porto Alegre, Brasil, 2004.

BEIZER, Boris. Black Box Testing. Editora John Wiley & Son, USA, 1995.

BEZERRA, Cicero. A qualidade do processo de desenvolvimento de software a partir da gestão de projetos: um estudo de caso. SBQS 2004, Brasília, Brasil, 2004.

BHAT, Thirumalesh; NAGAPPAN, Nachiappan. Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies. ISESE´06, Rio de Janeiro, Brasil, 2006.

BOEHM, Barry. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, v.21 n.5, p. 61-72. 1988.

CHIDAMBER, Shyam R.; KEMERER, Chris F. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, v. 20, n. 6, p. 476-493. 1994.

COCKBURN, Alister. Agile Software Development, Editora Addison Wesley: Boston, USA, 2002.

CRAIG, R. D.; JASKIEL, S.P. Systematic Software Testing, Editora Artech House Publisher: USA, 2002.

CRESPO, Adalberto N. et al. Uma Metodologia para Teste de Software no Contexto da Melhoria de Processo. CENPRA, Campinas, Brasil, 2005.

CRISPIN, Lisa. Testing eXtreme Rules of The Road, STQE Maganzine, USA, 2001.

CRISPIN, Lisa; HOUSE, Tip. Extreme programming testing, Editora Addison Wesley: Boston, USA, 2003.

DUSTIN, Elfriede; RASHKA, Jeff; PAUL, John. Automated Software Testing. Editora Addison Wesley: Boston, USA, 1999.

FENTON, Norman E.; PFLEEGER, Shari L.. Software Metrics, Editora PWS Publishing: Boston, USA, 1997.

FONTOURA, Lisandra M.; PRICE, Roberto T.. Usando GQM para Gerenciar Riscos em Projetos de Software. 18º. SBES 2004, Brasília, Brasil, 2004.

HAMILL, Paul. Unit Test Frameworks. Editora Oreilly e Assoc.:USA, 2004.

HEPTAGON. FDD - Feature-Driven Development. heptagon.com.br, 2007. Disponível em:http://heptagon.com.br/?q=fdd. Acessado em: 25/01/2007.

HIGHTOWER, Richard; LESIECKI, Nicholas. Java Tools for eXtreme programming, Editora Jonh Wiley & Sons: NY, USA, 2002.

HUNT, Andy; THOMAS, Dave. Progmatic Unit Testing in C# with Nunit. Editora Pragmatic, USA, 2004.

JEFFRIES, Ron; ANDERSON, Ann; HENDRICKSN, Chet. Extreme Programming Installed. Editora Addison Wesley: Boston, USA, 2000.

KAN, Stephen H.. Metrics and Models in Software Quality Engineering, Editora Addison Wesley: Boston, USA, 2003.

KAN, Stephen H. ; PARRISH, J; MANLOVE, D. In-Process metrics for softgware testing. IBM System Journal, v. 40, n.1, 2001.

KIRNER, Tereza G. ; ABIB, Janaina C. Inspections of Software Requirements specification Documents: A pilot Study. SIGDOC, Utah, USA, 1997.

LAING, V.; COLEMAN, C. Principal Components of Orthogonal Object-Oriented Metrics White Paper Analyzin Results of NASA Object-Oriented Data(323-08-14), Software Assurance Technology Center, NASA, USA, 2001.

LEFFINGWELL, Dean. Scaling Software Agility: Best Practices for Large Enterprises. Addison Wesley: Boston, USA, 2007.

LEWIS, Willian E; VEERAPILLAI, Gunasekaran. Software Testing and Continuous Qualitgy Improvement, 2 Ed., Editora CRC Press: Florida, USA, 2004.

MCT. Programa Brasileiro da Qualidade e Produtividade em Software, 4 Ed. Brasília, Brasil, 2006.

MCT/SEPIN. Caracterização das empresas produtoras de Software no Brasil, MCT/SEPIN, 2005. Diponível em: http://www.mct.gov.br/index.php/ content/view/4495.html. Acessado em: 11/11/2007.

MCT/SEPIN. Qualidade e Produtividade no Setor de Software Brasileiro, Brasília, Brasil, 2000.

MCCONNELLI, Steve. Software Project Survival Guide. Editora Microsoft Press, USA, 1997.

MESZAROS, Gerard. xUnit Test Patterns: Refactoring Test Code. Addison Wesley, Boston, USA, 2007.

MYERS, Glenford J. The Art of Software Testing, Editora John Wiley & Sons, New Jersey, USA, 1979.

MYERS, Glenford J. The Art of Software Testing, 2 ed., Editora John Wiley & Sons, New Jersey, USA, 2004.

NAGAPPAN, Nachiappan. Toward a Software Testing and Reliability Early Warning Metric Suite. ICSE´04. Edinburgh, Scotland, UK, 2004.

NAGAPPAN, Nachiappan; WILLIAMS, Laurie; VOULK, Mladen; OSBORNE, Jason. Early Estimation of Software Quality Using In-Process Testing Metrics: A Controlled Case Study. WoSQ´05, Missouri, USA, 2005.

NAGAPPAN, Nachiappan. Tese de doutorado. A Software Testing and Reliability Early Warning (STREW) Metric Suite. Noth Carolina state University. Raleigh, Carolina do Norte, USA, 2005.

PETERS, James F.; PEDRICZ, Witold. Engenharia de software, teoria e prática, 2 Ed., Editora Campus, Rio de Janeiro, Brasil, 2001.

PFLEEGER, Shari L.. Software Engineering: Theory and practice, 2 Ed., Editora Prentice Hall: New Jersey, USA, 2001.

PRESSMAN, Roger S.. Software engineering: a practitioner's approach, 5 ed., Editora McGraw-Hill: Nova York, USA, 2001.

PRESSMAN, Roger S.. Software Engineering, Editora IEEE Computer Society Press: Los Alamos, USA, 2001.

RISING, Linda; JANOFF, Normam S. The Scrum Software Development Process for Small Teams, IEEE Software, v.17 n.4, p.26-32, Julho 2000.

SHERRIFF, Mark; NAGAPPAN, Nachiappan; WILLIAMS, Laurie; VOUK, Mladen. Early Estimation of Defect Density Using In-Process Haskell Metrics Model. WoSQ´05, Missouri, USA, 2005.

SOMMERVILLE, Ian. Engenharia de Software, 6 ed., Editora Pearson Addison Wesley: São Paulo, Brasil, 2003.

VLIET, Hans Van. Software Engineering - Principles and Practice, Editora John Wiley & Sons, New Jersey, USA, 2002.

WILLIAMS, Laurie.; MAXIMILIEN, E.Michael; VOUK, Mladen. Test-Driven Development as a Defect-Reduction Practice. ISSRE. 2003. Denver, USA, 2003