Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência Computação

TÚLIO CARNEIRO DE CASTRO OLIVEIRA

AS ATIVIDADES DE TESTE E O MÉTODO XP DE DESENVOLVIMENTO DE SOFTWARE

Belo Horizonte
2011

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência Computação
Especialização em Informática: Ênfase: Engenharia de Software

AS ATIVIDADES DE TESTE E O MÉTODO XP DE DESENVOLVIMENTO DE SOFTWARE

por

TÚLIO CARNEIRO DE CASTRO OLIVEIRA

Monografia de Final de Curso

Prof. Rodolfo Resende
Orientador

Belo Horizonte
2011

TÚLIO CARNEIRO DE CASTRO OLIVEIRA

AS ATIVIDADES DE TESTE E O MÉTODO XP DE DESENVOLVIMENTO DE SOFTWARE

Monografia apresentada ao Curso de Especialização em Informática do Departamento de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Especialista em Informática.

Área de concentração: Engenharia de Software.

Orientador: Prof. Rodolfo Resende

Belo Horizonte
2011

Oliveira, Túlio Carneiro de Castro
Extreme Programming / Túlio Carneiro de Castro Oliveira. – 2011.
vi, 43f. :il.

Orientador: Rodolfo Resende.
Monografia (especialização) – Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.

1. Computação – Monografias 2. Engenharia de Software – Monografias. Construção – Monografias. I. Resende, Rodolfo. II. Universidade Federal de Minas Gerais. Departamento de Ciência da Computação. III. Título.

AGRADECIMENTOS

A Deus, por me dar saúde e disciplina para a conclusão deste trabalho acadêmico.
Ao meu pai, que mesmo distante é possível sentir sua presença sempre me iluminando e me dando força para continuar.
Salomão Lopes, pelo incentivo e pelas trocas de experiências sempre enriquecedoras e valiosas.
Rodolfo Resende, pela disposição e as inúmeras orientações para manter este trabalho no trilho certo.
Aos amigos, por compreenderem a minha ausência nos momentos em que deveríamos estar juntos.
E finalmente, a minha mãe e irmãs, pelo amor carinho e paciência.

RESUMO

O tema deste trabalho baseou-se na constante busca por partes das empresas de desenvolvimento de software em melhorar a qualidade do seu produto final. Para alcançar esta meta as empresas vêm investindo uma fatia considerável do seu orçamento em seu processo de software, mais especificamente nos testes de software. Os testes vão além da detecção e correção de erros, eles são também indicadores da qualidade do produto. Há diferentes metodologias de desenvolvimento de software, e existem metodologias que estão com seus esforços de testes focados em diferentes pontos do desenvolvimento de software. Na metodologia ágil, uma grande porção dos testes de software é colocada sobre a responsabilidade do programador. Já nas metodologias tradicionais, muitas das execuções dos testes só acontecem após o processo de codificação estar completo. Como os testes de código são de responsabilidades dos programadores, qual é a função de um testador dentro de um time ágil? Com a automação dos testes, o testador está livre para focar em áreas que gere muito mais valor como os testes exploratórios, usabilidade e testar a aplicação de maneira que o programador não previu.

Palavras-chave: Teste ágil, Programação Extrema, métodos ágeis.

ABSTRACT

The theme of this study was based on the constant search for shares of companies that develop software to improve the quality of your final product. Companies have been investing a considerable portion of their budget on their software process, specifically Software Testing. The tests, more than a means of detecting and correcting errors, are indicators of product quality. There are different models of software development, and there are models that have their testing efforts focused on different points of software development. In Agile, a large portion of software testing responsibility is placed on the programmer. In more traditional models, the test executions occur mainly after the coding process is complete. Since testing of code is the responsibility of the programmers, what is the role of a tester in an Agile team? With the automation of testing, the tester is free to focus on areas that generate more value such as exploratory testing, usability and testing the application in ways not anticipated by the programmer.

Keywords*:* Agile testing, Extreme Programing, Agile.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

CMMI Modelo de Maturidade e Capacitação Integrado *(Capability Maturity Model Integration)*
DCI Aumento do Custo dos Defeitos (*Defects Cost Increase*)
DSDM Método de Desenvolvimento de Sistemas Dinâmicos *(Dynamic Systems Development Method)*
FDD Desenvolvimento Dirigido a Funcionalidade *(Feature Driven Development)*
IEC
ISO Organização Internacional para Padronização *(International Organization for Standardization)*
QA Garantia da Qualidade (*Quality Assurance)*
SEI Instituto de Engenharia de Software *(Software Engineering Institute)*
XP Programação Extrema (Extreme Programing)

SUMÁRIO

1. INTRODUÇÃO

Muito se têm discutido sobre como implementar um processo de teste eficaz junto as metodologias ágeis. Já é sabido pela comunidade ágil que as técnicas tradicionais de testes usadas nas abordagens tradicionais não são aplicadas a um time ágil justamente porque é entregue software funcionando (mesmo que parcialmente) com um alto grau de qualidade para o cliente em ciclos curtos que duram entre uma semana e um mês.

A Programação Extrema (Extreme Programming – XP) é um dos principais métodos ágeis. A literatura em português se refere à Programação Extrema utilizando as expressões "o XP" (o método) e também "a XP" (a programção). A programação Extrema – XP foi concebida principalmente por Kent Beck. Segundo Beck (BECK, 2000) a base dos testes neste método são os testes de unidade realizados pelos programadores e os testes de aceitação que ele previa poderem ser escritos pelo cliente.

Contudo, a capacidade dos testes de unidade de detectar defeitos é limitada. Segundo Caper Jones, os testes de unidade detectam de 25% a 30% dos defeitos de um software. Enquanto isso, Rex Black afirma – no artigo *How Agile Methodologies Challenge Testing* – que até 85% dos defeitos podem ser detectados antes do sistema entrar em produção se forem realizados testes de sistema no software por uma equipe independente. Portanto, apesar dos testes de unidade auxiliarem na detecção de defeitos, os testes de sistema ainda são melhores para detectar a grande maioria dos defeitos do software.

Ao falar dos pápeis que precisam ser preenchidos para que um time de XP funcione Beck (BECK, 2000) fala sobre o testador como um papel focado no cliente. Onde o testador é responsável por ajudar o cliente a escolher e escrever testes funcionais, além de executar todos os testes regularmente e divulgar os resultados dos testes e garantir que as ferramentas para testes executem bem.

Crispin e Gregory (CRISPIN L. e GREGORY J., 2009) vão além ao afirmar que o testador ágil é o profissional que acolhe as mudanças, colabora com ambos os times: técnico e de negócio, compreende o conceito de testes para o documento de requisitos e o desenvolvimento dirigido por testes. O testador ágil tem habilidades técnicas, sabe como colaborar com outros para automatizar testes, e é o experiente em testes exploratórios.

1.1 OBJETIVO GERAL

A partir do tema proposto, o objetivo principal deste trabalho é abordar o processo de testes de software, dando ênfase aos pontos fortes e fracos de alguns métodos de software existentes no desenvolvimento de sistemas mas com ênfase no método XP.

1.2 OBJETIVOS ESPECÍFICOS

1. Apresentar o conceito de teste de software;
2. Apresentar o modelo tradicional cascata e o seu processo de teste de software;
3. Apresentar o manifesto ágil e os principais métodos ágeis;
4. Apresentar o Extreme Programming (XP);
5. Descrever as atividades de testes no método ágil XP;
6. Pesquisar e apresentar as novas responsabilidades do testador em um time ágil de desenvolvimento.

2 METODOLOGIA “TRADICIONAL” VERSUS ÁGIL

2.1 PROCESSO

Segundo Walkins (WALKINS, J., 2004) um processo visa identificar e reutilizar elementos comuns de alguma abordagem particular para a realização de uma tarefa, e aplicar os elementos comuns a outras tarefas afins. Sem esses elementos comuns, reutilizáveis, um processo irá lutar para proporcionar um meio eficaz e eficiente de atingir essas tarefas, e encontrará uma difícil aceitação e utilização por outros profissionais que trabalham nesse campo.

Um processo, segundo o IEEE, é uma sequência de passos executados com um determinado objetivo. Uma boa visão de processo é nos dada por Pádua (PÁDUA, W., 2009) em que ele afirma que processo é uma receita que é seguida por um projeto; o projeto concretiza uma abstração, que é o processo. Não se deve confundir um processo (digamos, uma receita de risoto de camarão) com o respectivo produto (risoto de camarão) ou com a execução do processo através de um projeto (a confecção de um risoto de camarão por determinado cozinheiro, em determinado dia).

A seguir apresentamos os principais processos de software correspondentes tanto a metodologia tradicional quanto aos métodos ágeis de desenvolvimento de software.

2.2 ABORDAGEM “TRADICIONAL”

2.2.1 CASCATA

O ciclo de vida denominado Cascata serviu como base de um dos primeiros processos de desenvolvimento de software. No modelo de ciclo em cascata, segundo Pádua (PÁDUA, W., 2009), os principais subprocessos são executados em estrita sequência, o que permite demarcá-los com pontos de controle bem definidos. Cada etapa tem associada ao seu término uma documentação que deve ser aprovada para que a etapa posterior possa ter início.

Existem diversas representações semelhantes, mas basicamente o processo consiste nas seguintes fases: Requisitos, Análise, Desenho (Projeto), Implementação, Testes e Implantação, como ilustra a figura 1.

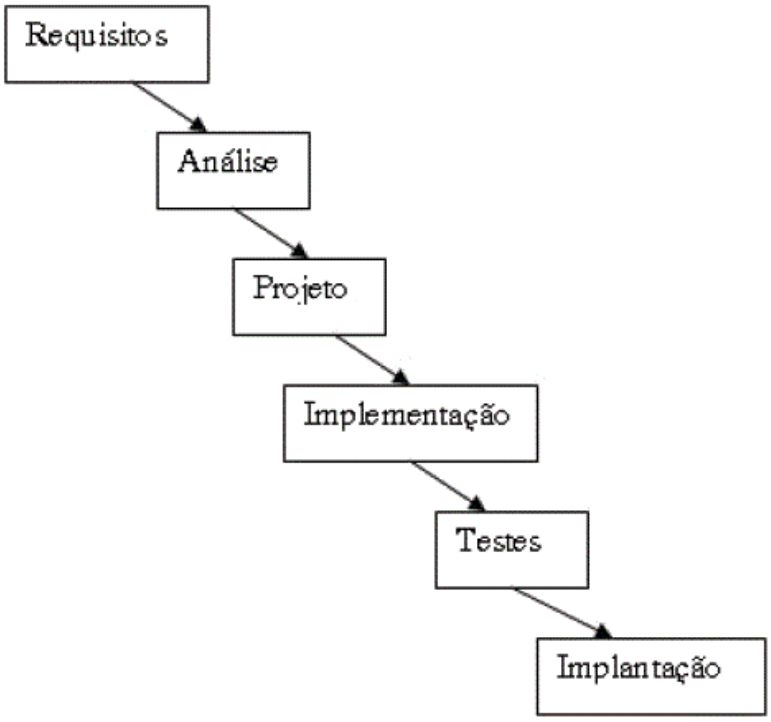

Figura 1 - O modelo em cascata (Acervo pessoal).

Segundo Pádua (PÁDUA, W., 2009), estes pontos de controle facilitam muito a gestão dos projetos, o que faz com que este processo seja, em princípio, confiável e utilizável em projetos de qualquer escala. Por outro lado, se interpretado literalmente, é um processo rígido e burocrático, em que as atividades de requisitos, análise e desenho têm de ser muito

bem dominadas, pois, teoricamente, o processo não prevê a correção posterior de problemas nas fases anteriores. O modelo em cascata puro é de baixa visibilidade para o cliente, que só recebe o resultado final do projeto.

2.2.2 ESPIRAL

No modelo em espiral, segundo Pádua (PÁDUA, W., 2009) o produto é desenvolvido em uma série de iterações. Cada nova iteração corresponde a uma volta na espiral. Por liberação (*release*) entende-se um estágio parcialmente operacional de um produto, que é submetido à avaliação de usuários em determinado marco de um projeto.

Liberações escolhidas podem ser designadas como versões oficiais do produto, e colocadas em uso. Isso permite construir produtos em prazos curtos, com novas características e recursos que são agregados à medida que a experiência descobre sua necessidade.

O ciclo de vida em espiral permite que os requisitos sejam definidos progressivamente, e apresenta alta flexibilidade e visibilidade. Esse modelo permite que, em pontos bem-definidos, os usuários possam avaliar partes do produto e fornecer realimentação quanto às decisões tomadas. Facilita também o acompanhamento do progresso de cada projeto, tanto por parte de seus gerentes como dos clientes. Entretanto, também requer gestão sofisticada, e o desenho deve ser muito robusto, para que a estrutura do produto não se degenere ao longo das iterações. Além disso, requer equipe muito disciplinada e experiente. Esse modelo de ciclo de vida é aplicado em processos ágeis, em particular no XP.

2.3 ABORDAGEM ÁGIL

No final dos anos 90 e início dos anos 2000, movidos pela observação de que equipes de desenvolvimento de software nas mais diversas organizações estavam presas por processos cada vez mais burocráticos, um grupo de profissionais reuniu-se para delinear os valores e princípios que permitiriam às equipes de desenvolvimento produzir rapidamente e responder às mudanças. Eles chamaram a si mesmos de Aliança Ágil. Trabalharam por dois

dias para criar um documento relacionando, dentre outros, um conjunto de valores (MARTIN, 2010). O resultado foi o Manifesto da Aliança Ágil.

Manifesto para Desenvolvimento Ágil de Software

Estamos descobrindo maneiras melhores de desenvolver software, fazendo-o nós mesmos e ajudando outros a fazerem o mesmo. Através deste trabalho, passamos a valorizar:

Indivíduos e interações **mais que processos e ferramentas**
Software em funcionamento **mais que documentação abrangente**
Colaboração com o cliente **mais que negociação de contratos**
Responder a mudanças **mais que seguir um plano**

Ou seja, mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda.

É importante destacar que não há ruptura entre os itens da esquerda e os da direita e sim de ênfase. A comunicação irá auxiliar em cada um dos itens da esquerda e será o grande facilitador desse processo.

Segundo Abrahamsson (ABRAHAMSSON, 2002), um método é considerado ágil quando efetua o desenvolvimento de software de forma incremental (liberação de pequenas versões, em iterações de curta duração), colaborativa (cliente e desenvolvedores trabalhando juntos, em constante comunicação), direta (o método em si é simples de aprender e modificar) e adaptativa (capaz de responder às mudanças até o último instante).

São considerados metódos ágeis: Extreme Programming (XP), Scrum, Crystal, Feature Driven Development (FDD), Dynamic Systems Development Method (DSDM) e Open Source Software Development dentre outros.

2.3.1 PROGRAMAÇÃO EXTREMA - XP

Dentre os métodos ágeis mais utilizados, destaca-se o Extreme Programming (XP), atribuído a Kent Beck, Ron Jeffries e Ward Cunningham (BECK, 2000).

Segundo Beck (BECK, 2000) o XP é uma disciplina de desenvolvimento de software. Por ser uma disciplina há certos elementos que devem estar presentes na prática do XP. Por exemplo, se o programador não escreve testes, então ele não está praticando o XP de forma correta.

O XP prioriza a satisfação do cliente, o metódo visa entregar o software de acordo com a necessidade do cliente. Para tanto Walkins (WALKINS, 2004) o XP procura melhorar o sucesso do projeto em quatro áreas:

- Comunicação - o XP incentiva a equipe a desenvolver e testar o software para manter uma comunicação eficaz e frequente tanto com os outros membros da equipe e com o cliente.
- Simplicidade - o XP promove a idéia do "Você não vai precisar disso" (You aint't gonna need it – YAGNI), ou seja, o XP incentiva aos programadores que eles optem pela solução mais simples e clara para seus códigos.
- Retroalimentação (Feedback) - Em todo o desenvolvimento do projeto, os programadores obtêm retornos de seu trabalho de outros programadores e, crucialmente, do cliente, que fica rapidamente em contato com o produto de software graças às entregas logo no seu início.
- Coragem - programadores estão aptos a responder de forma positiva e proativa à evolução das necessidades dos clientes e mudanças no ambiente de desenvolvimento.

Ao adotar esses quatro princípios, projetos de desenvolvimento são mais ágeis e compreensivos, melhor comunicação entre a equipe significa que o desenvolvimento do sistema está integrado com mais facilidade e com menos erros, mais a melhoria da comunicação com o cliente, combinada com boa retroalimentação garante que o software fornecido melhor corresponda às necessidades dos usuários (WATKINS, J., 2004).

Além destes, recentemente observou-se a importância de se considerar um novo valor, o respeito. Para Beck (BECK, 2000) se os membros do time não se importarem uns com os outros e com o que eles estão fazendo, os resultados não são adequados.

Segundo Beck (BECK, 2004) o XP se utiliza de determinadas práticas que, se bem organizadas, oferecem um reforço mútuo com o intuito de se complementarem:

- O Jogo do Planejamento – Determine brevemente o escopo da próxima versão combinando prioridades de negócio e estimativas técnicas. Quando a realidade se opuser ao planejamento, atualize o planejamento.
- Entregas frequentes – Coloque um sistema simples rapidamente em produção, e depois libere novas versões em ciclos curtos.
- Metáfora – Guie o desenvolvimento com uma simples história, compartilhada por todos, sobre como o sistema funciona como um todo.
- Projeto simples – O sistema deve ser projetado da maneira mais simples possível em qualquer momento. A complexidade desnecessária é removida assim que for descoberta.
- Testes – Os programadores escrevem testes de unidade continuamente, os quais devem executar sem falhas para que o desenvolvimento prossiga. Os clientes escrevem testes demonstrando que as funções estão terminadas.
- Refatoração – Os programadores reestruturam o sistema sem alterar seu comportamento a fim de remover duplicidade, melhorar a comunicação, simplificar e acrescentar flexibilidade.
- Programação em Pares – Todo código de produção é escrito por dois programadores em uma máquina.
- Propriedade Coletiva – Qualquer um pode modificar qualquer código, em qualquer lugar do sistema, a qualquer momento.
- Integração Contínua – Integre e atualize as versões do sistema várias vezes por dia, cada vez que uma tarefa for terminada.
- Semana de 40 horas – Como regra, trabalhe no máximo 40 horas por semana. Nunca faça hora extra por duas semanas seguidas.
- Cliente presente – Inclua um cliente real no time, disponível todo o tempo para responder a questões.
- Padrões de codificação – Os programadores escreverão código respeitando as regras que enfatizam comunicação através do código.

Este trabalho adota o XP como metódo ágil por ser o metódo mais focado com as práticas de engenharia de software.

2.3.2 SCRUM

O método ágil Scrum tem como objetivo, segundo Schwaber (SCHWABER, 2004), definir um processo para projeto e desenvolvimento de software orientado a objeto, que seja focado nas pessoas e que seja indicado para ambientes em que os requisitos surgem e mudam rapidamente.

De acordo com Fowler (FOWLER, 2011) Scrum concentra-se nos aspectos de gestão de desenvolvimento de software, dividindo o desenvolvimento dentro de iterações (chamadas de “sprints”) e aplicando melhor acompanhamento e controle com reuniões diárias. Ele coloca muito menos ênfase nas práticas de engenharia e muitas pessoas combinam a sua abordagem de gerenciamento de projetos com as práticas de engenharia do XP.

O método baseia-se ainda, conforme Schwaber (SCHWABER, 2004), em princípios como: equipes pequenas de, no máximo, sete pessoas; requisitos que são pouco estáveis ou desconhecidos; e iterações curtas. Divide o desenvolvimento em intervalos de tempos de, no máximo 30 dias, também chamadas de Sprints. Este método não requer ou fornece qualquer técnica ou método específico para a fase de desenvolvimento de software, apenas estabelece conjuntos de regras e práticas gerenciais que devem ser adotadas para o sucesso de um projeto. As práticas gerenciais do Scrum são: Lista de requisites pendentes do produto (Product Backlog), Reunião Diária (Daily Scrum), Corrida (Sprint), Reunião de Planejamento da Corrida (Sprint Planning Meeting), Requisitos pendentes da Corrida (Sprint Backlog) e Reunião da Revisão da Corrida (Sprint Review Meeting).

3 MODELO DE DESENVOLVIMENTO DE SOFTWARE CASCATA VERSUS MÉTODO XP DE DESENVOLVIMENTO DE SOFTWARE

Segundo Chrissis (CHRISSIS, 2003) “o propósito do CMMI é estabelecer um guia para melhorar o processo da organização e sua capacidade para gerenciar o desenvolvimento, aquisição e manutenção de produtos e serviços”. Ahern (AHERN, 2004) destaca que um dos principais objetivos do modelo CMMI é assegurar a compatibilidade com a norma ISO/IEC 15504 permitindo a análise de áreas independentes do nível de maturidade. O modelo CMMI define atividades, artefatos e outros aspectos do que é denominado Área de Processo. O modelo CMMI possui quatro categorias que agrupam áreas de processo: Engenharia de Sistemas, Engenharia de Software, Produto Integrado e Desenvolvimento de Processo e, finalmente, a Aquisição. Estas categorias, auxiliam no planejamento da melhoria do processo de toda organização.

A implementação de uma ou mais das áreas de processo destas categorias ao mesmo tempo com uma única terminologia e infra-estrutura de treinamento e avaliação é considerada uma grande vantagem do modelo CMMI, pois a organização determina em quais disciplinas deseja melhorar seu processo.

As áreas de processo, quando implementadas, determinam a melhoria do processo. Segundo Fiorini (FIORINI, 1998), área de processo é um conjunto de práticas relacionadas em uma área que quando executadas coletivamente satisfazem um conjunto de objetivos importantes para a melhoria significante daquela área.

O modelo CMMI, segundo o SEI (Software Engineering Institute), Chrissis (CHRISSIS 2003) e Ahern (AHERN, 2004), é dividido em duas representações: Estágiada e Contínua. A representação Estágiada define 5 níveis de maturidade (Inicial, Gerenciado, Definido, Gerenciado Quantitativamente e De Otimização). Cada nível (estágio) possui diversas áreas de processos onde cada uma se encontra em um único nível. A representação Contínua tem 6 níveis para dimensão da capacitação (Incompleto, Executado, Gerenciado, Definido, Gerenciado Quantitativamente, Otimização). Diferentemente na representação Estágiada, as áreas de processo na representação Contínua são independentes dos níveis de maturidade, ficando relacionadas apenas com a capacidade (Capability) do processo, ou seja, uma determinada área de processo em particular poderá ter sua capacidade avaliada independente das outras áreas de processo. O modelo CMMI possui pouco mais de vinte áreas de processo. As áreas de processo do modelo CMMI possuem práticas específicas e genéricas.

A prática específica (SP - Specific Practice) é uma descrição detalhada das atividades que são consideradas fundamentais para alcançar os objetivos específicos. As práticas genéricas estão relacionadas com várias áreas de processo. Uma prática genérica é a descrição de uma atividade fundamental para alcançar os objetivos genéricos.

Para simplificar este trabalho foi dedicada atenção somente a duas áreas de processo: Verificação (VER) e Validação (VAL), que se encontram no nível 3 de maturidade. A área de processo Verificação tem por objetivo garantir que produto de trabalho selecionados atendam aos respectivos requisitos. Já a área de processo Validação visa demonstrar que um produto ou componente satisfaz ao uso que se pretende dele, quando colocado no ambiente alvo. E, dentro destas áreas, serão analisadas as suas práticas específicas.

3.1 VERIFICAÇÃO E VALIDAÇÃO

Existem muitas diferenças entre o que o cliente necessita, e o que o analista projeta. A Figura 2 tenta capturar com algum humor o que pode acontecer durante a construção de um sistema.

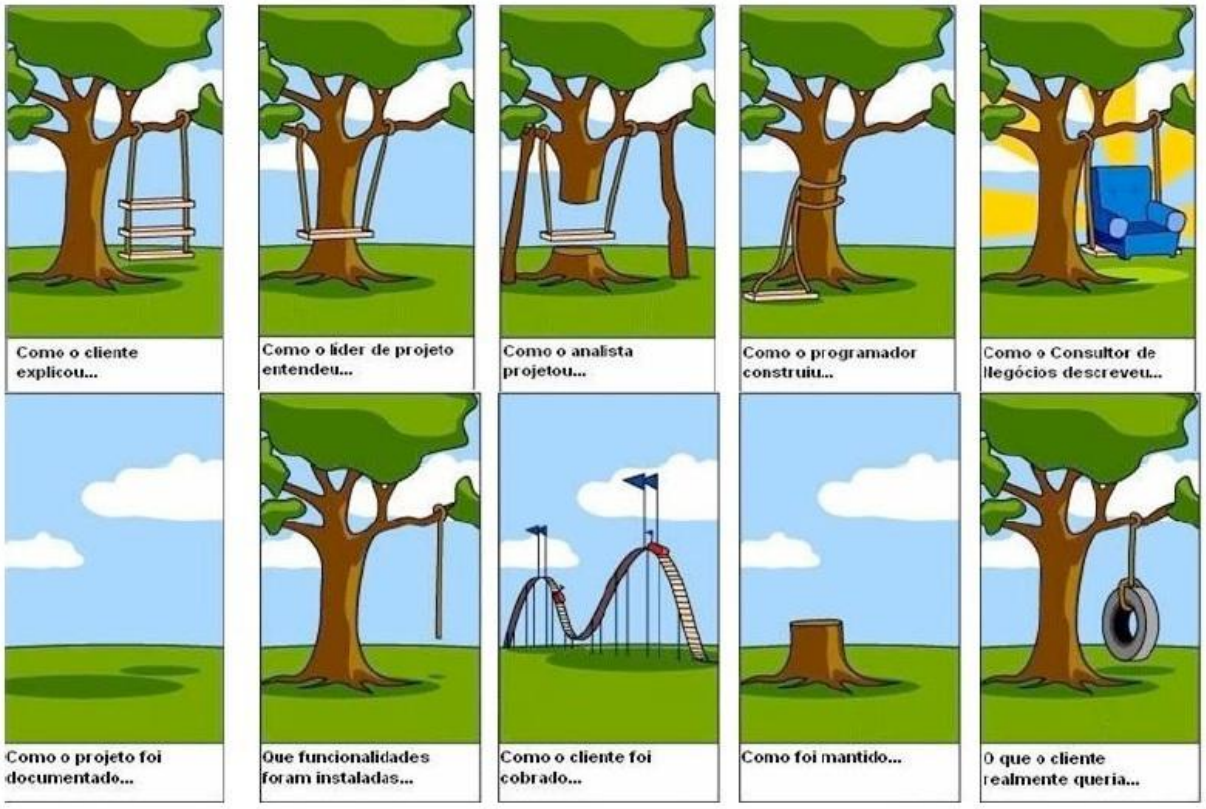

Figura 2: Exemplo ilustrativo de validação (NETO, 2007)

Os testes de verificação são tipos de teste que geralmente são realizados sobre os documentos que são produzidos durante o ciclo de desenvolvimento. Essas atividades são iniciadas antes da codificação do sistema, elas começam na fase de requisitos e continuam através da codificação do produto. O teste consiste na realização de inspeções, revisões e/ou auditorias sobre os produtos gerados em cada etapa do processo, evitando que dúvidas e assuntos mal resolvidos passem para a próxima fase do processo de desenvolvimento.

A verificação tem por objetivo provar que o produto vai ao encontro dos requisitos especificados. Ela garante a qualidade do software na produção e na manutenção.

Os testes de validação também são aplicados diretamente no software. Eles podem ser aplicados em componentes isolados, módulos existentes ou em todo o sistema. Esse tipo de teste avalia se o sistema atende aos requisitos e especificações analisadas nas fases iniciais do projeto, se ele vai ao encontro dos requisitos do usuário.

Testes de verificação e validação são complementares, eles não são atividades redundantes. Eles possuem objetivos e naturezas distintas, contribuem para a detecção de erros e aumentam a qualidade final do produto.

3.2 TÉCNICAS DE TESTE DE SOFTWARE

Atualmente existem muitas maneiras de se testar um software. Mesmo assim, existem as técnicas que sempre foram muito utilizadas em sistemas desenvolvidos sobre linguagens estruturadas que ainda hoje têm grande valia para os sistemas orientados a objeto. Apesar de os paradigmas de desenvolvimento serem diferentes, o objetivo principal destas técnicas continua a ser o mesmo: encontrar falhas no software.

As técnicas de teste são classificadas de acordo com a origem das informações utilizadas para estabelecer os requisitos de teste. Elas contemplam diferentes perspectivas do software e impõe-se a necessidade de se estabelecer uma estratégia de teste que contemple as vantagens e os aspectos complementares dessas técnicas. As técnicas existentes são: técnica funcional (ou teste caixa-preta) e estrutural (ou teste caixa-branca) (MAIDASANI, 2007).

3.2.1 TÉCNICA ESTRUTURAL (OU TESTE DE CAIXA-BRANCA)

Técnica de teste que avalia o comportamento interno do componente de software (Figura 3). Essa técnica trabalha diretamente sobre o código fonte do componente de software para avaliar aspectos tais como: teste de condição, teste de fluxo de dados, teste de ciclos e teste de caminhos lógicos (PRESSMAN, 2005).

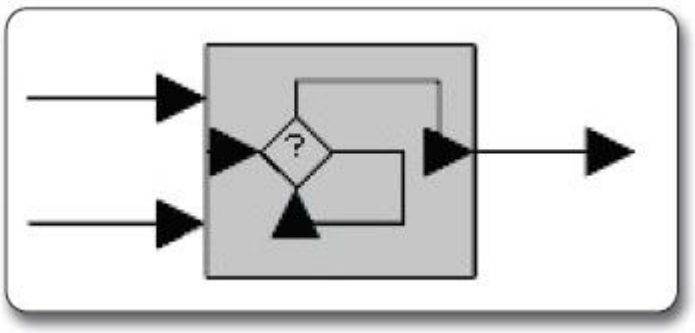

Figura 3: Técnica de Teste Estrutural (NETO, 2007).

Os aspectos avaliados nesta técnica de teste dependerão da complexidade e da tecnologia que determinarem a construção do componente de software, cabendo, portanto, avaliação de outros aspectos além dos citados anteriormente.

O testador tem acesso ao código fonte da aplicação e pode construir códigos para efetuar a ligação de bibliotecas e componentes. Este tipo de teste é desenvolvido analisando-

se o código fonte e elaborando-se casos de teste que cubram todas as possibilidades do componente de software.

Dessa maneira, todas as variações originadas por estruturas de condições são testadas. O quadro 1 apresenta um código fonte, extraído do trabalho de Barbosa e outros (BARBOSA, 2000) que descreve um programa de exemplo que deve validar um identificador digitado como parâmetro, e a Figura 5 apresenta o grafo de programa extraído a partir desse código, também extraído do trabalho de Barbosa e outros. A partir do grafo deve ser escolhido algum critério baseado em algum algoritmo de busca em grafo (exemplo: visitar todos os nós, arcos ou caminhos) para geração dos casos de teste estruturais para o programa (PFLEEGER, 2004).

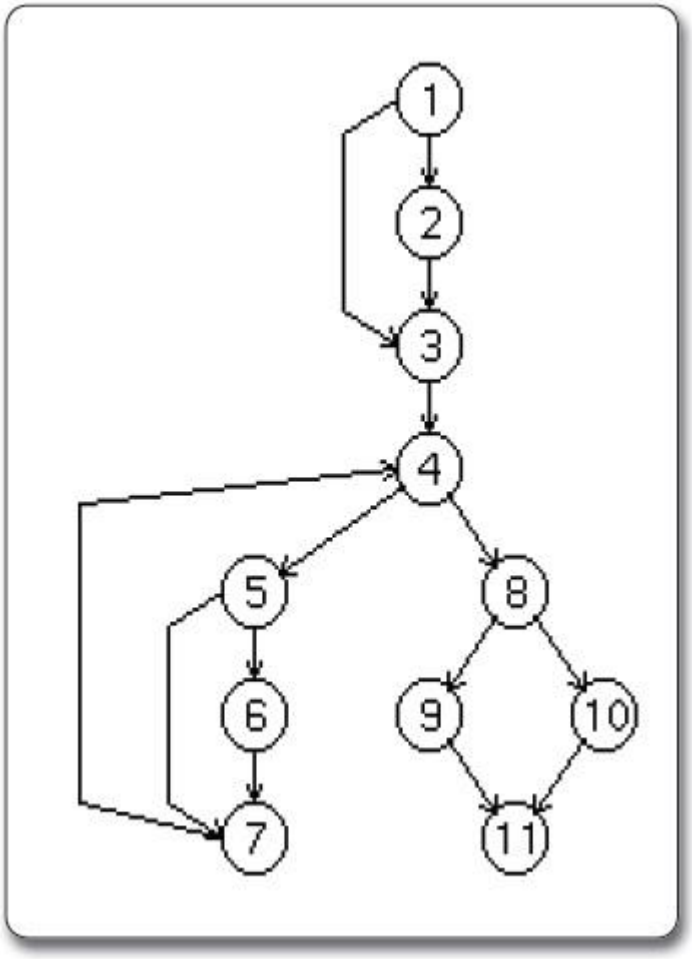

Figura 4: Grafo de Programa (Identifier.c) (BARBOSA, 2000).

```
/* 01 */  {
/* 01 */  char  achar;
/* 01 */  int   length, valid_id;
/* 01 */  length = 0;
/* 01 */  printf ("Digite um possível identificador\n");
/* 01 */  printf ("seguido por <ENTER>: ");
/* 01 */  achar = fgetc (stdin);
/* 01 */  valid_id = valid_starter (achar);
/* 01 */  if (valid_id)
/* 02 */            length = 1;
/* 03 */  achar = fgetc (stdin);
/* 04 */  while (achar != '\n')
/* 05 */  {
/* 05 */            if (!(valid_follower (achar)))
/* 06 */                      valid_id = 0;
/* 07 */            length++;
/* 07 */            achar = fgetc (stdin);
/* 07 */  }
/* 08 */  if (valid_id && (length >= 1) && (length < 6) )
/* 09 */            printf ("Valido\n");
/* 10 */  else
/* 10 */            printf ("Invalido\n");
/* 11 */   }
```

Quadro 1: Código fonte do programa identifier.c (BARBOSA, 2000)

Um exemplo bem prático desta técnica de teste é o uso da ferramenta livre JUnit para desenvolvimento de casos de teste para avaliar classes ou métodos desenvolvidos na linguagem Java. A técnica de teste de Estrutural é recomendada para os níveis de Teste da Unidade e Teste de Integração, cuja responsabilidade principal fica a cargo dos desenvolvedores do software, que são profissionais que conhecem bem o código-fonte desenvolvido e dessa forma conseguem planejar os casos de teste com maior facilidade.

Dessa forma, podemos auxiliar na redução dos problemas existentes nas pequenas funções ou unidades que compõem um software.

3.2.2 TESTE FUNCIONAL (OU TESTE DE CAIXA-PRETA)

Técnica de teste em que o componente de software a ser testado é abordado como se fosse uma caixa-preta, ou seja, não se considera a estrutura e o comportamento interno do mesmo (Figura 5).

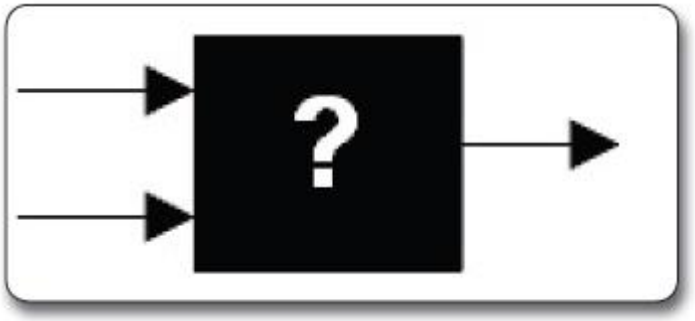

Figura 5: Técnica de teste funcional (NETO, 2007).

Dados de entrada são fornecidos, o teste é executado e o resultado obtido é comparado a um resultado esperado previamente conhecido. Haverá sucesso no teste se o resultado obtido for igual ao resultado esperado.

O componente de software a ser testado pode ser um método, uma função interna, um programa, um componente, um conjunto de programas e/ou componentes ou mesmo uma funcionalidade. A técnica de teste funcional é aplicável a todos os níveis de teste (PRESSMAN, 2005).

Para adoção da estratégia do teste de caixa preta não é necessário conhecer a tecnologia utilizada nem a arquitetura do sistema. Isso facilita a modelagem dos testes. O que é necessário é o conhecimento dos requisitos, o comportamento esperado do sistema para que se possa avaliar se o resultado produzido é o correto.

Eles são conhecidos como mais simples de implementar que os de caixa branca.

3.3 NÍVEIS DE TESTE

Há várias tarefas diferentes que precisam ser feitas para entregar testes eficazes e eficientes, e em uma variedade de níveis de testes como: componente, unidade, testes de desenvolvedor, através de testes de integração, módulo, em teste de sistemas e, através do teste de aceitação (WALKINS, J., 2000).

Segundo Rocha (ROCHA, 2001), definimos que os principais níveis e teste de software são:

- Teste de Unidade: Tem por objetivo explorar uma unidade convencional do projeto, procurando provocar falhas ocasionadas por defeitos de lógica e de implementação em

cada unidade, separadamente. Uma unidade usual são os métodos dos objetos ou mesmo pequenos trechos de código.

- Teste de Integração: visa provocar falhas associadas às interfaces entre as unidades ou módulos quando esses são integrados para construir a estrutura do software que foi estabelecida na fase de projeto.
- Teste de Sistema: avalia o software em busca de falhas por meio da utilização do mesmo, como se fosse um usuário final. Dessa maneira, os testes são executados nos mesmos ambientes, com as mesmas condições e com os mesmos dados de entrada que um usuário utilizaria no seu dia-a-dia de manipulação do software. Verifica se o produto satisfaz seus requisitos.
- Teste de Aceitação: são realizados geralmente por um restrito grupo de usuários finais do sistema. Esses simulam operações de rotina do sistema de modo a verificar se seu comportamento está de acordo com o solicitado.
- Teste de Regressão: Teste de regressão não corresponde a um nível de teste, mas é uma estratégia importante para redução de "efeitos colaterais". Consiste em se aplicar, a cada nova versão do software ou a cada ciclo, os testes que já foram aplicados nas versões ou ciclos de teste anteriores do sistema. Pode ser aplicado em qualquer nível de teste.

3.4 ELEMENTOS TRADICIONAIS DO PROCESSO DE TESTE

Quando um modelo sequencial como o modelo cascata é usado para o desenvolvimento de software, testadores devem estar especialmente preocupados com a qualidade, integridade e estabilidade dos requisitos. A falta de esclarecimento e definição dos requisitos no início do projeto irá provavelmente resultar no desenvolvimento de um projeto de software que não era o que os usuários queriam ou precisavam. Pior, a descoberta destes defeitos poderá ser adiada até o fim do ciclo de vida, na execução dos testes (CRAIG e JASKIEL, 2002).

Segundo Craig e Jaskiel (CRAIG e JASKIEL, 2002), modelos sequenciais são, em particular, difíceis de implementar no ponto de vista de testes. Como pode ser visto na figura 6, no modelo em cascata, o teste é considerado somente no final.

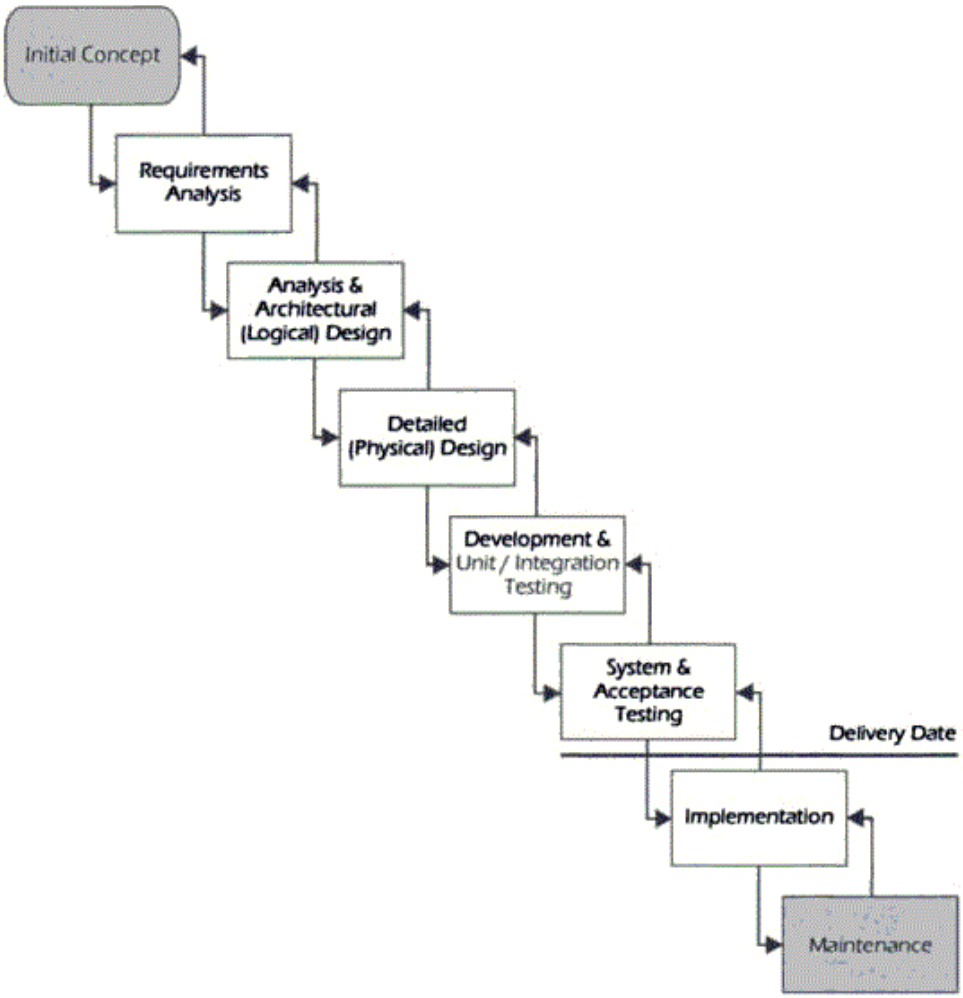

Figura 6: Modelo cascata (CRAIG e JASKIEL, 2002).

Outro problema do modelo cascata é que os testadores quase sempre se encontram no caminho crítico de entrega do software. Esta situação é agravada porque muitas vezes o software é entregue aos testadores tardiamente, e o tempo para a entrega não pode ser alterado. O resultado, claro, é que a janela para o teste é cada vez menor (CRAIG e JASKIEL, 2002).

3.4.1 MODELO V

O modelo-V divide o processo de teste em níveis em que o teste é realizado de forma incremental em conjunto com a implementação do sistema (PYHÄJÄRVI e RAUTIAINEN, 2004), (Figura 7).

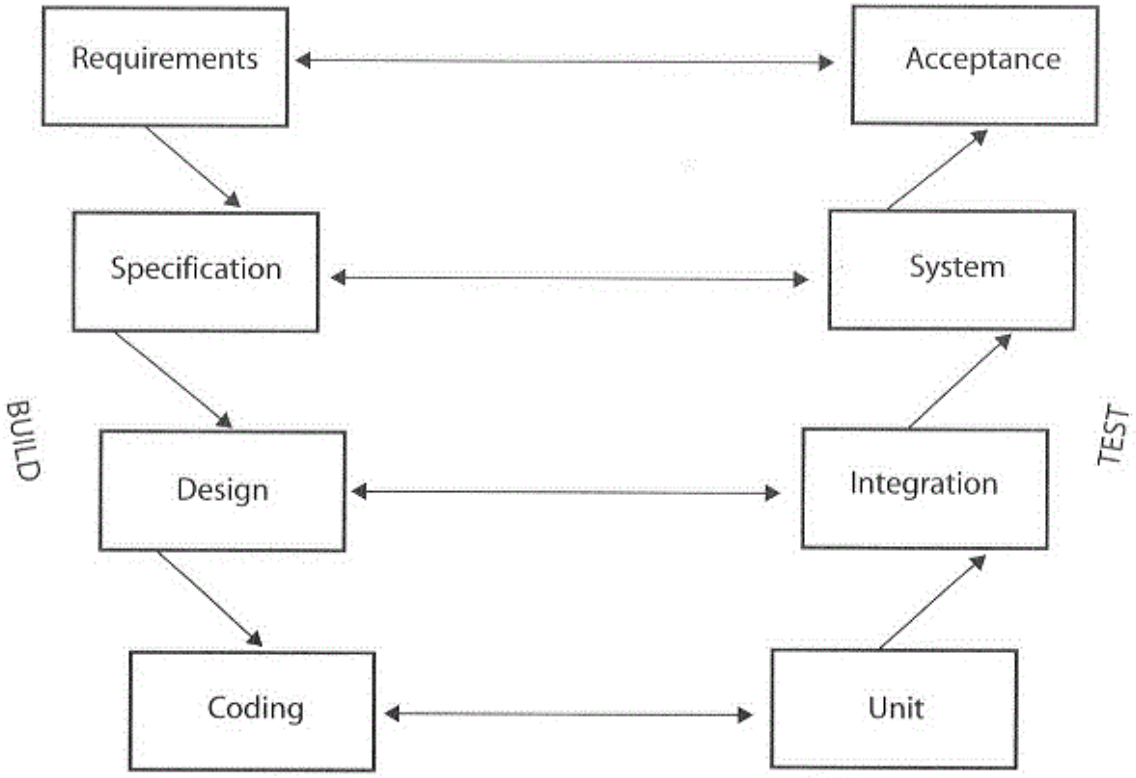

Figura 7: Especificações → Plano de Teste → Teste (PYHÄJÄRVI e RAUTIAINEN, 2004).

Seguindo a Figura 7, considerando a dimensão vertical como sendo na parte alta menos próximo do código e na parte baixa como próximo ao código, o planejamento e projeto dos testes devem ocorrer de cima para baixo, ou seja:

1. Inicialmente é planejado o teste de aceitação a partir do documento de requisitos;
2. Após isso é planejado o teste de sistema a partir do desenho de alto nível do software;
3. Em seguida ocorre o planejamento dos testes de integração a partir o desenho detalhado;
4. E por fim, o planejamento dos testes considerando o código.

Já a execução dos testes ocorre no sentido inverso.

Ainda segundo Pyhäjärvi e Rautiainen (PYHÄJÄRVI e RAUTIAINEN, 2004), o modelo-V é uma extensão do modelo cascata, onde cada fase do processo que lida com a implementação tem associada a ela uma atividade de verificação e validação chamada *test level*. O modelo-V mostra como a atividade de teste pode (e deve) ser levado em conta muito antes de haver algum código fonte para ser testado efetivamente.

O modelo-V, bem como suas variações herda alguns problemas do modelo de desenvolvimento em cascata. O modelo cascata assume que para prosseguir em um novo nível, o anterior deve estar finalizado. Ainda segundo Pyhäjärvi e Rautiainen (PYHÄJÄRVI E

RAUTIAINEN, 2004) existe uma certa tolerância para redefinição e redesenho, mas as mudanças se tornam mais caras à medida que o processo avança, porque muito se depende do que já foi feito.

3.5 MÉTODO XP

Vamos iniciar esta seção comparando o teste ágil e o teste no desenvolvimento de software tradicional. Considere a figura 8.

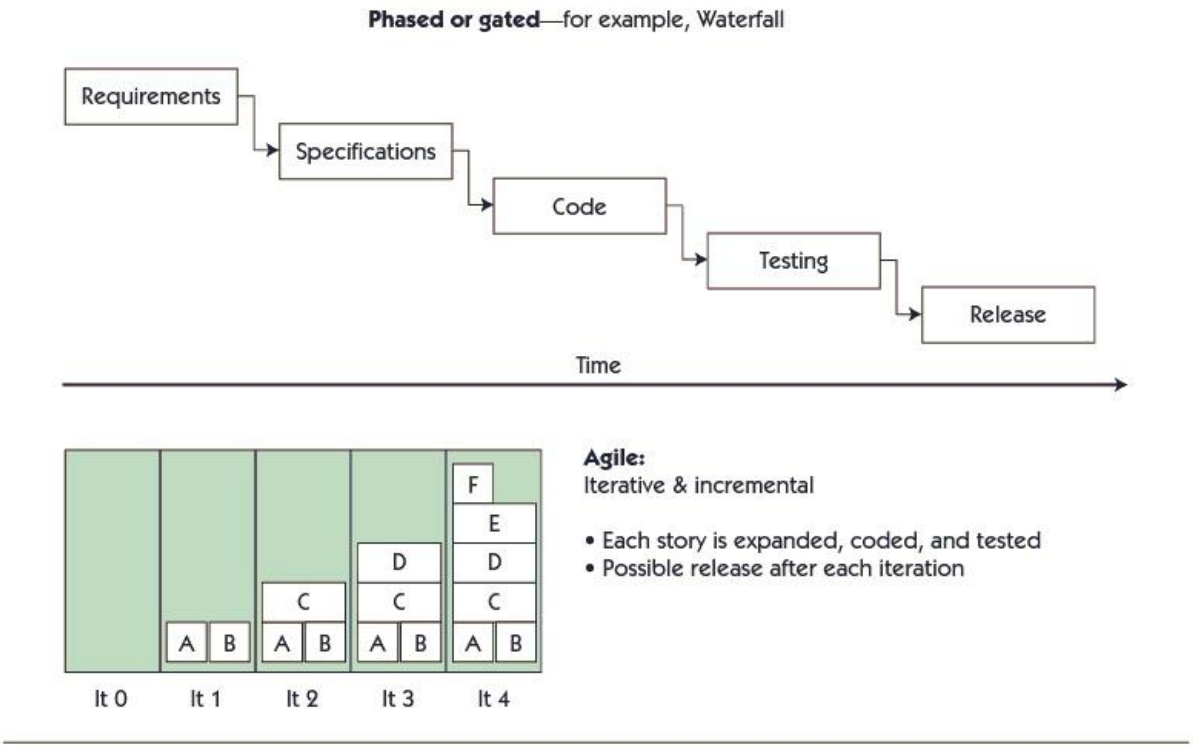

Figura 8: Teste tradicional versus Teste ágil (CRISPIN L. e GREGORY J., 2009).

No diagrama abordagem gradual, é evidente que o teste acontece no final, mesmo antes do lançamento. O diagrama é idealista, porque dá a impressão de que há tempo para ensaios, tal como existe para a codificação. Em muitos projetos, não é esse o caso. O teste é "esmagado" por causa de codificação que leva mais tempo do que o esperado, e porque as equipes tendem a entrar em um ciclo de codifica-conserta no final (CRISPIN L. e GREGORY J., 2009).

Ágil é interativo e incremental. Isso significa que os testadores testam cada incremento de código assim que eles são finalizados. Uma interação pode ser curta como uma semana, ou longa como um mês. O time constrói e testa uma pequena parte do código, tendo certeza que ele funcione corretamente, e então o time passa para a próxima parte que precisa ser construída. Programadores nunca passam adiante dos testes, porque a história não está "pronta" até estar testada (CRISPIN L. e GREGORY J., 2009).

Segundo Beck (BECK, 2000) grande parte da responsabilidade pelos testes pertence aos programadores, o papel do testador em um time de XP é, na verdade, focado no cliente. Ele é o responsável por ajudar o cliente a escolher e escrever testes funcionais.

De acordo com Beck (BECK, 2004) teste no XP é uma atividade técnica que lida diretamente com defeitos. O XP aplica dois princípios para aumentar o custo-eficácia do teste: verificação dupla (Double-checking) e o aumento do custo do defeito (Defect Cost Increase - DCI).

Teste de software é uma verificação dupla. O programador fala o que ele espera de um cálculo enquanto escreve um teste. Ele falar é completamente diferente de quando ele implementar o cálculo. Se as duas expressões do cálculo forem equivalentes, o código e o teste estão em harmonia e provavelmente estarão corretos (BECK, 2004).

O aumento do custo do defeito é o segundo princípio aplicado ao XP para aumentar o custo-eficácia do teste. Para Beck (BECK, 2004) DCI é uma das poucas verificações empíricas verdadeiras sobre desenvolvimento de software: O quão cedo encontrarmos um defeito, mais barato é o seu conserto.

DCI implica que os estilos de desenvolvimento de software com longos ciclos de retroalimentação são mais caros e tem muitos defeitos remanescentes, conforme ilustra a figura 9.

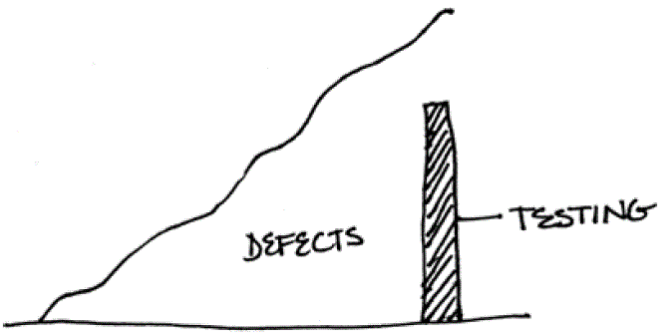

Figura 9: Tardiamente, testes são caros deixam muitos defeitos (BECK, 2004).

O XP usa o DCI em sentido inverso para reduzir o custo das correções dos defeitos e o número de defeitos na implantação. Trazendo testes automatizados para dentro do ciclo de programação (vide figura 10), o XP tenta corrigir os defeitos mais cedo e de forma mais barata.

Figura 10: Frequentemente, testes reduzem custos e defeitos (BECK, 2004).

Segundo Beck (BECK, 2004) há várias implicações para os testes mais frequentes. Uma delas é que a mesma pessoa quem comete o erro é a que tem que escrever o teste. Se o intervalo de tempo entre criação e detecção de erros são meses, faz total sentido que estas duas atividades pertençam a pessoas diferentes. Se o intervalo são minutos, o custo de comunicar expectativas entre duas pessoas seria proibitivo, mesmo com um testador dedicado a cada programador.

Já que são os programadores que escrevem os testes, então eles precisariam de uma outra perspectiva do sistema. O programador trás para seu código e teste um ponto de vista singular da funcionalidade do sistema, perdendo algum do valor da dupla verificação. Dupla verificação funciona melhor quando dois pensamentos distintos chegam à mesma resposta. Para obtermos todos os beneficios da dupla verificação, o XP têm dois conjuntos de testes: um conjunto é escrito pela perspectiva dos programadores, e o outro conjunto pela perspectiva do cliente ou usuários. Estes dois conjuntos de testes se complementam, se os testes dos programadores são perfeitos, os testes realizados pelos clientes não vão pegar quaisquer erros (BECK, 2004).

3.5.1 TESTES DE ACEITAÇÃO

Segundo Wells (WELLS, 2011) testes de aceitação podem ser criados a partir de histórias de usuário. Durante uma iteração, as histórias de usuários selecionadas durante a reunião de planejamento da iteração serão traduzidas em testes de aceitação.

De acordo com Beck (BECK, 2000) os clientes escrevem testes história por história. A pergunta que eles precisam fazer a si mesmo é “O que precisaria ser conferido para

eu ter certeza de que essa história está OK?”. Cada cenário que eles inventam torna-se um teste, neste caso, um teste funcional.

Testes de aceitação são testes de caixa preta. Cada teste de aceitação representa algum resultado esperado do sistema. Os clientes são responsáveis por verificar a exatidão dos testes de aceitação, e analisando os resultados dos testes decidir qual falha nos testes é de máxima prioridade. Testes de aceitação podem ser também utilizados como parte dos testes de regressão, antes de uma versão para produção (WELLS, 2011).

Tipicamente, os clientes não conseguem escrever testes funcionais sozinhos. Eles precisam da ajuda de alguém que possa primeiro traduzir os dados de teste em testes e, com o tempo, possa criar ferramentas que possibilitem ao cliente escrever, executar e manter seus próprios testes. É por isso que um time de XP de qualquer tamanho tem ao menos um testador dedicado. A função do testador é traduzir as idéias de testes às vezes vagas do cliente em testes reais, automatizados e isolados (BECK, 2000).

Um papel que os testadores de software devem aceitar é o desenvolvimento do processo do teste de aceitação. Isso significa que eles irão desenvolver um processo para a definição dos critérios de aceitação, desenvolver um processo para a construção de um plano de teste de aceitação, desenvolver um processo para executar o plano de teste de aceitação, e desenvolver um processo de registo e apresentar os resultados dos testes de aceitação (MAIDASANI, 2007).

O nome “teste de aceitação” foi alterado para teste funcional. Isso reflete melhor a intenção, que é a garantia de que uma exigências dos clientes foram atingidas e que o sistema é aceitável.

3.5.2 TESTES DE UNIDADE

Os testes de unidade são um dos pilares do XP, segundo Beck (BECK, 2000) qualquer aspecto/função do programa que não tenha teste automatizado simplesmente não existe.

Os testes de unidade permitem a propriedade coletiva. Quando você cria testes de unidade você guarda a sua funcionalidade de ser acidentalmente alterada. Exigir que todos os códigos passem em todos os testes de unidade antes de ser liberado garante que toda a funcionalidade sempre funcione. Propriedade individual do código não é necessário se todas as classes são guardados por testes de unidade (WELLS, 2011).

Os testes de unidade permitirá a refatoração também. Após cada pequena mudança, os testes de unidade podem verificar que uma mudança na estrutura não introduziu uma alteração na funcionalidade (WELLS, 2011).

Testes de unidade proporcionam uma rede de segurança dos testes de regressão e dos testes de validação para que você possa refatorar e integrar de forma eficaz. Criando o teste de unidade antes do código ajuda ainda mais a solidificar os requisitos, melhorando o foco do desenvolvedor (BECK, 2000).

A construção de uma suíte de testes universal única para testes de validação e de regressão permite a integração contínua. É possível integrar todas as mudanças recentes de forma rápida, então execute a versão mais recente do seu próprio conjunto de testes. Quando um teste falha as suas versões mais recentes são incompatíveis com a versão mais recente do time. Corrigindo pequenos problemas em poucas horas leva menos tempo do que consertando enormes problemas pouco antes do prazo. Com testes de unidade automatizados é possível juntar um conjunto de mudanças com a última versão lançada e fazer o lançamento em um curto período de tempo (WELLS, 2011).

3.5.3 REFATORAÇÃO

Ao implementar uma função do programa, os programadores sempre se perguntam se existe uma maneira de alterar o programa existente para fazer com que a adição da nova função seja simples. Depois que eles a adicionaram, se perguntam como podem simplificar o programa, mantendo ainda todos os testes executando. Isso é chamada *refatoração* (BECK, 2000).

Quando nós removemos redundância, eliminamos funcionalidades não usadas e rejuvenescemos desenhos obsoletos, nós estamos refatorando. Refatoração em todo o ciclo do projeto economiza tempo e aumenta a qualidade (WELLS, 2011).

3.5.4 PROGRAMAÇÃO EM PARES

Segundo Beck (BECK, 2000) programação em pares é um diálogo entre duas pessoas tentando programar simultaneamente (e analisar, projetar e testar) e entender em conjunto como programar melhor. É uma conversação em muitos níveis, assistida por um computador e focada nele.

Pelo fato de os pares serem embaralhados o tempo todo, a informação se difunde rapidamente pelo time (BECK, 2000). Os testes em par contribuem para um espaço de aprendizado contínuo dentro da equipe. Os mais experientes ou quem tem mais conhecimento sobre um assunto termina passando informações valiosas para seus pares.

Essa prática pressupõe uma comunicação contínua entre os testadores que formam o par ajudando-os a se manterem focado no teste. Não é necessário interrupções para se tomar nota, tirar dúvidas, consultar manual ou documentação, replicar o erro em outra máquina. Essas atividades podem ser realizadas pelo outro testador. Ela também melhora o relato dos erros, pois facilita a reprodução e tudo que é relatado é revisado por outra pessoa.

A prática de teste em par pode ser desenvolvida tendo como participantes um testador e uma pessoa com conhecimento profundo do sistema, o programador sênior do projeto por exemplo. Duas pessoas com focos tão diferentes tendem a somar qualidade no final do processo. Enquanto o foco de um está em mostrar como a funcionalidade deve se comportar a do outro busca falhas e caminhos alternativos. Esta prática ajuda tanto no processo de teste como no processo de desenvolvimento. Testadores absorvem a forma de pensar dos desenvolvedores e os desenvolvedores absorvem a forma de pensar dos testadores, ocorrendo assim maior cuidado no desenvolvimento das funcionalidades tendo o objetivo não apenas nos fluxos principais, mas também nos fluxos alternativos e de exceção que podem levar a erros no sistema.

3.5.5 INTEGRAÇÃO CONTÍNUA

De acordo com Beck (BECK, 2000) nenhum código deve permanecer não-integrado por mais do que algumas horas. Ao fim de cada episódio de desenvolvimento, o código é integrado com a última entrega, e todos os testes precisam executar com 100% de correção.

Uma maneira simples de fazer isso é ter uma máquina dedicada apenas à integração. Quando a máquina estiver livre, um par com código para ser integrado a ocupa, copia a versão atual, copia as modificações (conferindo e resolvendo qualquer colisão) e executa os testes até que eles passem (100% corretos).

Integrar apenas um conjunto de modificações de cada vez é uma prática que funciona bem porque fica óbvio quem deve fazer as correções quando um teste falha: o último par de programadores que fez a última integração. Afinal o par anterior havia executado todos os testes com resultados corretos. E se não conseguirmos deixar todos os testes corretos, devemos abandonar o que havia sido feito e recomeçar, já que obviamente não havia conhecimento suficiente para programar aquela função.

3.6 OUTROS TESTES

Segundo Beck (BECK, 2000) enquanto os testes funcionais e de unidade são o centro da estratégia de aplicação de testes no XP, existem outros que podem ser úteis de vez em quando sempre que o time de XP sentir que não está no caminho certo; estes outros testes são:

- Teste em paralelo – um teste projetado para provar que o novo sistema funciona exatamente como o sistema antigo. Na verdade, o teste mostra a forma como o novo sistema difere do sistema antigo, para que uma pessoa de negócio possa tomar uma decisão de negócios quando a diferença é suficientemente pequena para colocar o novo sistema em produção.
- Teste de carga (stress) – um teste projetado para simular diferentes níveis de cargas de trabalho. Testes de carga são adequados para sistemas complexos, em que as características de performance não são fáceis de prever.

- Teste de macaco – um teste projetado para garantir que o sistema reage de forma adequada frente a entradas sem sentido.

4 PORQUE O TESTADOR ÁGIL É DIFERENTE?

4.1 O QUE É TESTADOR ÁGIL?

Para Crispin e Gregory (CRISPIN L. e GREGORY J., 2009) um testador ágil é um profissional de teste que não repudia as mudanças, colabora bem com ambos os times: técnico e de negócios, e compreende os conceitos de utilização de testes de documento de requisitos e desenvolvimento das unidades.

Os testadores também estão na equipe de desenvolvimento, porque o teste é um componente central do desenvolvimento ágil de software. Testadores são defensores da qualidade em nome do cliente e ajudam a equipe de programadores a entregar o máximo de valor de negócio.

As equipes do cliente e do desenvolvedor trabalham em estreita colaboração em todos os momentos. Idealmente, eles são apenas uma equipe com um objetivo comum. Esse objetivo é agregar valor à organização. Projetos ágeis têm o seu progresso em iterações, que são os ciclos de desenvolvimento de pequeno porte que geralmente duram de uma a quatro semanas. A equipe do cliente, com a contribuição dos desenvolvedores, vai priorizar as histórias a serem desenvolvidas, e a equipe de desenvolvedores irá determinar o quanto de trabalho eles podem ter. Eles vão trabalhar em conjunto para definir os requisitos com testes e exemplos, e escrever o código correto correspondente ao teste. Testadores devem se posicionar tanto no mundo do cliente quanto no mundo dos desenvolvedores, conforme ilustra a figura 9.

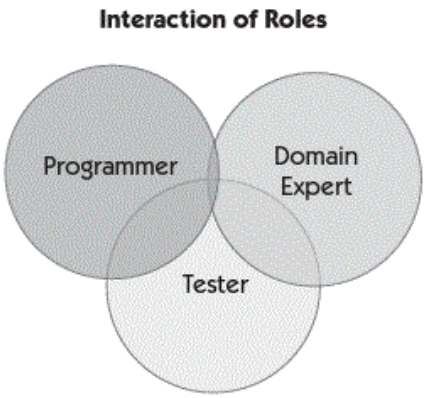

Figura 11: Interação dos papéis (CRISPIN L. e GREGORY J., 2009).

4.2 PRÍNCIPOS PARA O TESTADOR ÁGIL

O trabalho de Crispin e Gregory (CRISPIN L. e GREGORY J., 2009) define uma lista dos princípios ágeis de testes, que é parcialmente derivada dos princípios do manifesto ágil:

- Prover retroalimentação (feedback) contínua – O papel tradicional de um testador é ser um “provedor de informação” e fazê-la valiosa para um time ágil.
- Entregar valor para o cliente – Para garantir que entregaremos algum valor em cada interação, o time analisa cada história para identificar o “caminho crítico” da funcionalidade necessária. Após concluir esta tarefa, o time volta e completa os demais recursos. No pior cenário apenas a funcionalidade básica é liberada. Isso é melhor do que entregar nada ou algo que funcione parcialmente.
- Habilitar a comunicação face-a-face – Não há substituto para a comunicação face-a-face. Desenvolvimento ágil depende de colaboração constante. O testador irá constantemente buscar o cliente e membros do time técnico para discutirem e colaborarem entre si.
- Ter coragem – Coragem é um valor fundamental no XP, e para que um testador saia da sua zona de conforto e se junte ao time para dividir com ele as responsabilidades tanto para o sucesso quanto para o fracasso é preciso coragem.
- Manter o simples – Teste ágil significa fazer o mais simples teste possível para verificar que a funcionalidade existe ou que ele vai de encontro aos padrões de qualidade do cliente.
- Praticar a melhoria contínua - Testadores ágeis e suas equipes estão sempre à procura de ferramentas, habilidades e práticas que possam ajudá-los a agregar mais valor ou obter um melhor retorno sobre o investimento do cliente.
- Responder a mudança - Responder a mudança é um valor fundamental para os profissionais ágeis, mas descobrimos que ele é um dos conceitos mais difíceis para os testadores. Estabilidade é o que os testadores anseiam para que eles possam dizer: "Eu testei isso; ele está feito". No entanto, como testadores ágeis, temos de ser receptivos a mudança.
- Auto-organização - O testador ágil é parte de uma equipa auto-organizada. A cultura de equipe permeia a filosofia de testes ágeis. Quando os programadores,

administradores de sistema, analistas, especialistas em banco de dados, e a equipe do cliente pensam constantemente sobre o teste e automação de testes, testadores irão desfrutar de uma perspectiva completamente nova.

- Foco em pessoas - Times ágeis que aderirem à verdadeira filosofia ágil dão a todos os membros do time o mesmo peso. Testadores ágeis sabem que eles contribuem com um valor exclusivo para seus times e times de desenvolvimento descobriram que têem mais sucesso quando seu time inclui pessoas com habilidades específicas em testes.
- Divertir-se - O desenvolvimento ágil recompensa a paixão do testador ágil pelo seu trabalho. O trabalho como testadores ágeis são particularmente satisfatório, porque a sua perspectiva e habilidades vão agregar valor real para os seus times.

Juntos, estes princípios trazem valor de negócio para o time. No desenvolvimento ágil, o time tem a responsabilidade de entregar um software de alta qualidade que encanta os clientes e torna o negócio mais rentável (CRISPIN L. e GREGORY J., 2009).

4.3 TRABALHANDO EM UM TIME TRADICIONAL

Garantia da Qualidade (Quality Assurance - QA) são as organizações mais tradicionais de desenvolvimento de software, é realizado por uma organização separada ou em grupo, com uma ou mais equipes de engenheiros de QA sob a supervisão de um gerente de QA. Equipes de QA podem trabalhar em estreita colaboração com os analistas de negócios ou desenvolvedores, mas frequentemente com limites bem estabelecidos. Ambos, desenvolvedores e QA recebem dos analistas de negócios os requisitos de negócio e os cronogramas detalhados. Enquanto o desenvolvimento escreve o código, QA escreve planos e casos de teste projetados para garantir que o software entregue corresponda aos requisitos fornecidos pelo analista de negócios. Quando o desenvolvimento do produto se presume completa, o aplicativo é entregue ao QA para testes. QA em seguida, executa um ciclo completo de testes de produto e envia relatórios de defeitos para o desenvolvimento. 'Teste' é composto por diferentes atividades especializadas, incluindo os testes funcionais, testes de desempenho e testes de regressão. Quando o teste estiver completo, a área de desenvolvimento corrige defeitos e entrega uma nova revisão ao QA. Este ciclo é repetido até que o produto seja considerado pronto para o lançamento.

Ao longo dos anos, alguns dos pontos fracos desta abordagem têm se tornado aparente. Considere o atraso entre quando o software é escrito e quando o desenvolvimento recebe uma retroalimentação. Este atraso exige que os desenvolvedores atualizem um código que foi escrito semanas ou meses atrás, o que pode ser um código fundamental que tem implicações importantes quando alterados, o que amplia o escopo dos testes subsequentes.

Segundo Crispin e Gregory (CRISPIN L., GREGORY, J., 2009) como membro de um time de QA, nós temos a ilusão de controle. Sendo que não podemos controlar o modo como o código foi escrito, ou mesmo se os programadores testaram seus códigos.

Equipes tradicionais estão focados em garantir que todos os requisitos especificados são entregues no produto final. Se tudo não estiver pronto até a data de lançamento, a liberação é geralmente adiada. As equipes de desenvolvimento geralmente não têm entrada sobre as funcionalidades no lançamento, ou como eles devem trabalhar. Programadores individuais tendem a se especializar em uma área particular do código. Testadores estudam os documentos de requisitos para escrever os seus planos de teste, e então eles esperam pelo trabalho a ser entregue a eles para o teste (CRISPIN L., GREGORY, J., 2009).

4.4 TRABALHANDO EM UM TIME DE XP

Um time ágil não faz todos os trabalhos de requisitos para um sistema, depois todos os trabalhos de desenvolvimento e, em seguida, todos os trabalhos de testes consecutivamente. Em vez disso, o time ágil considera uma pequena parte do sistema e trabalha em conjunto para concluí-la.

Completando uma parte, que se refere ao conjunto ou a uma história, significa que a gestão do produto, desenvolvimento e testes trabalharam em conjunto para alcançar um objetivo comum. O objetivo é que a história esteja com o estado de 'feito'. As histórias são selecionadas com base em sua prioridade e estimativa de esforço. A estimativa de esforço é outra atividade do time, que também inclui testadores. A equipe também identifica as dependências, desafios técnicos, testes. Toda a equipe concorda sobre os critérios de aceitação final para uma história para determinar quando ela está realmente "feita".

Durante uma iteração, várias histórias podem estar em vários estágios de desenvolvimento, teste, ou de aceitação. O teste ágil é contínuo, já que todos em um time ágil

realizam testes. No entanto, o foco e o tempo dos testes são diferentes dependendo do tipo de teste que é executado. Desenvolvedores assumem a liderança em testes de nível de código, enquanto o testador do time ágil proporciona retroalimentação rápida durante todos os estágios de desenvolvimento, ajuda ou tem conhecimento no nível de código para realizar os testes, assume a liderança na automação dos testes de aceitação construindo planos de teste de regressão e revela cenários de teste adicional através dos testes exploratórios.

Contrapondo Beck (2000) com o Beck (2004) é perceptível a mudança de postura do autor em relação ao testador. O testador ágil garante a cobertura adequada dos testes de aceitação, lidera os esforços de automação em testes integrados, em nível de sistema, mantém os ambientes de teste e os dados disponíveis, identifica problemas de regressão e compartilha conhecimentos técnicos de teste (BECK, 2004).

Equipes ágeis trabalham em estreita colaboração com o negócio e têm uma compreensão detalhada dos requisitos. Eles estão focados no valor que eles podem oferecer, e eles podem ter uma grande contribuição para priorizar os recursos. Testadores não sentam e esperam por trabalho, eles levantam e olham as maneiras de contribuir durante todo o ciclo de desenvolvimento.

5 CONCLUSÃO

Hoje as companhias de software precisam responder para o cliente de forma rápida para continuar competitivas. Por praticarem métodos ágeis, times de desenvolvimento podem entregar produtos rapidamente e com alta qualidade.

Como o teste ainda é uma atividade crítica em um time de desenvolvimento ágil, o testador ágil é o elemento chave dentro do time ágil. De acordo com Beck (2000) o testador dentro de um time de XP tem um papel limitado. Ele é responsável por auxiliar o cliente a escrever os testes de aceitação, executar os testes e dar visibilidade aos seus resultados.

Crispin e Gregory (2009) vão muito além ao falar que o testador ágil é um agente de mudança dentro do time. Ele tem a missão de “infectar” todo o time com as práticas de teste, participando desde a criação dos testes de aceitação junto ao cliente e garantindo que os requisitos foram bem assimilados pelos programadores provendo uma rápida retroalimentação ao time assim que os testes forem executados.

Acreditamos que o testador é uma ponte importante entre o time de negócios (cliente) e o time de desenvolvimento (programadores). E em um time ágil novas habilidades são requeridas para que este profissional se sobressaia, o testador tem que participar ativamente dentro do time, testes exploratórios passam a ser uma habilidade indispensável, entender de automação e integração contínua, ser um excelente comunicador e dar visibilidade ao seu próprio trabalho ao time.

O testador é um dos responsáveis pelo processo de teste dentro do seu ambiente de trabalho e deve assegurar que este processo está sendo corretamente cumprido pelos membros do time de desenvolvimento e pelos membros do time de negócios.

REFERÊNCIAS

ABRAHAMSSON, Pekka et al. Agile software development methods: Review and analysis. VTT Publications 478. Oulu, Finland: VTT Publications, 2002.

AHERN, Dennis. CLOUSE, Aaron. TURNER, Richard. CMMI Distilled: a practical introduction to integrated process improvement. Boston: Addison Wesley, 2004.

BARBOSA, E.; MALDONADO, J.C.; VINCENZI, A.M.R.; DELA MARO , M.E; SOUZA, S.R.S. e JINO, M.. "Introdução ao Teste de Software. XIV Simpósio Brasileiro de Engenharia de Software", 2000.

BECK, Kent. *Extreme Programming Explained*: Embrace Change. 1ª Edição. Bookman. 2000. 224 p.

BECK, Kent. ANDRES, Cynthia. *Extreme Programming Explained*: Embrace Change. 2ª Edição. Bookman. 2004. 224 p.

BLACK, Rex. Managing the Testing Process, Microsoft Press, Junho de 1999.

CHRISSIS, Mary B.; KONRAD M.; SHRUM S. CMMI: Guidelines for Process Integration and Product Improvement. SEI, Addison Wesley, 2003.

CRAIG, R.D., JASKIEL, S. P., "Systematic Software Testing", Artech House Publishers, Boston, 2002.

CRISPIN, L., GREGORY J., Agile Testing: A Practical Guide for Testers and Agile Teams. 1ª Edição. 2009. 577 p.

FIORINI, Soeli. STAA, Arndt, BAPTISTA R. Engenharia de software com CMM. Rio de Janeiro: Brasport, 1998.

FOWLER, Martin. 2011. Disponível em <http://martinfowler.com/articles/newMethodology.html#FlavorsOfAgileDevelopment>. Acesso em 16/01/2011.

GREGORY, J.; CRISPIN, L. What Is Agile Testing Anyway? Disponível em: <http://www.informit.com/articles/article.aspx?p=1316250&seqNum=2>. Acesso em: 02 Jul 2010.

GUIMARÃES, L.; VILELA, P. Comparing Software Development Models Using CDM. Disponível em: <http://portal.acm.org/citation.cfm?id=1095714.1095793&coll=ACM&dl=ACM&CFID=97314516&CFTOKEN=34318538>. Acesso em: 05 Jul 2010.

JONES, T.Capers. Patterns of Software System Failure and Success,,ed. Intl Thomson Computer Pr, 1995. 292p.

MAIDASANI, Dinesh. Software Testing, 1ª Edição. 2007. 389 p.

MARICK, B. New Models for Test Development. *Proceedings of Quality Week.* 1999.

MARTIN, Robert C. Agile Processes. 2010. Disponível em <www.objectmentor.com/resources/articles/agileProcess.pdf>. Acesso em 15/12/2010.

NETO, Arilo C. Dias. Introdução a Teste de Software. Engenharia de Software Magazine. Rio de Janeiro, 54-60, 2007.

PÁDUA, Filho Wilson. *Engenharia de Software:* Fundamentos, Métodos e Padrões. 3ª Edição. LTC Editora, 2009. 1248 p.

PFLEEGER, S. L., "Engenharia de Software: Teoria e Prática", Prentice Hall- Cap. 08, 2004.

PRESSMAN, R. S., "Software Engineering: A Practitioner's Approach", McGraw-Hill, 6th ed, Nova York, NY, 2005.

PYHÄJÄRVI, M.; RAUTIAINEN, K. Integrating Testing and Implementation into Development. *Engineering Management Journal*. v. 16, n. 1, p. 33-39, Mar. 2004. Disponível em: <https://umdrive.memphis.edu/cdndnvll/www/testing/papers/pyhajarvi.and.Raoutiainen.(2004).pdf>. Acesso em: 29 Jun 2010.

ROCHA , A. R. C., MALDONADO, J. C., WEBER, K. C. et al., "Qualidade de software – Teoria e prática", Prentice Hall, São Paulo, 2001.

SCHWABER, Ken; Agile Project Management with Scrum, ed. Microsoft Press, 2004. 163p.

WATKINS, J., Agile Testing: How to Succeed in an Extreme Testing Enviroment. 1ª Edição. Cambridge, 2009. 338 p.

WELLS, Don. Extreme Programming: *A Gentle Introduction*. Disponível em: <http://www.extremeprogramming.org>. Acesso em: 02 Jul 2010.

WILLIAM, W. XP123. Exploring Extreme Programming. Disponível em: <http://www.xp123.com>. Acesso em: 05 Jul 2010.